Beyond Generated Software Documentation – A Web 2.0 Perspective

Philipp Schugerl[1], Juergen Rilling[1], Philippe Charland[2]
[1]*Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada*
{p_schuge, rilling}@cse.concordia.ca
[2]*System of Systems Section, Defence R&D Canada – Valcartier Québec, Canada*
philippe.charland@drdc-rddc.gc.ca

Abstract

Over the last decades, software engineering processes have constantly evolved to reflect cultural, social, technological, and organizational changes, which are often a direct result of the Internet. The introduction of the Web 2.0 resulted in further changes, creating an interactive and community driven platform. However, these ongoing changes have yet to be reflected in the way we document software systems. Documentation generators, like Doxygen and its derivatives (Javadoc, Natural Docs, etc.) have become the de facto industry standards for creating external technical software documentation from source code. However, the interwoven representation of source code and documentation within a source code editor limits the ability of these approaches to provide rich media, internationalization, and interactive content. In this paper, we combine the functionality of a web browser with a source code editor to provide source code documentation with rich media content. The paper presents our fully functional implementation of the editor within the Eclipse framework.

1. Introduction and Motivation

With millions of lines of source code written every day, the importance of good documentation cannot be overstated. Software components that are well-documented are easier to comprehend and consequently easier to maintain and reuse. This becomes especially important in large software systems [1].

Typical software system documentation consists of different types of artifacts, ranging from source code and low-level models to high-level documentation such as requirements and use cases. Good software documentation usually provides multiple views of a system (static, dynamic, external, internal) at different abstraction levels and using different formats (e.g., text, pictures, and video) [2].

Given that programmers typically lack appropriate tools and processes to create and maintain documentation, the documentation task has been widely considered as an unfavorable and labor-intensive task within software projects [3]. Documentation generators are among the tools that are most widely accepted and used.

In this research, we present a novel approach that combines the functionality of a web browser with a typical source code editor. In comparison to existing documentation generators, our approach provides support for WYSIWYG style documentation, internationalization, and interactive content embedded directly within the source code. The main objectives of our approach can be summarized as follows:

Support for program comprehension: In many software projects, the maintainer of a component is not the original author, and the original programmer is no longer accessible. As a result, maintainers have to rely on the existing documentation as one of the main sources for the comprehension of a software system. Well-documented software consists of a multitude of resources that are often accessible through a web browser. Resources may contain static and dynamic content (e.g., user comments within a help site or wiki) and rich media (e.g., pictures and videos). This is different from documentation generators like Javadoc [4] that are restricted to static content and do not support dynamic and/or frequent updates of the documentation. Additionally, websites are often available in different languages. Given the ability to integrate information from other artifacts (e.g., history information from bug tracking or version control system) and websites as part of source code documentation, a truly ambient and pervasive documentation can be established.

Maintaining documentation: Software documentation is created and updated throughout all stages of the software life cycle. For example, Javadoc uses comments and tags in order to document classes,

methods, and variables. Although such comments are fairly easy to create, the formatting using HTML and correct usage of tags represent an additional burden to most Java developers. Javadoc itself supports a multitude of tag types (e.g., @link, @see, etc.). Most programmers only use a small subset of the available tags, which is symptomatic of the difficulty programmers have in using the notation. Using the rich editing facilities provided by Wikis can eliminate the need for tags and ease both the creation and maintenance of documentation.

The remainder of this paper is organized as follows: Section 2 explains the background of this research and compares existing solutions to this work. We provide implementation details in Section 3 and summarize the main ideas and results in Section 4.

2. Software Documentation

Software documentation consists of a multitude of artifacts that can range from requirements documents to design models and source code comments. In this paper, we focus on system API documentation, which is a low-level documentation describing a software system at a class and method levels.

Literate and Elucidative Programming: Literate programming [5] was suggested in the early 1980's by Donald Knuth in order to combine the process of software documentation with software programming. Its basic principle is the definition of program fragments directly within software documentation. Literate programming tools can further extract and assemble the program fragments as well as format the documentation. The extraction tool is referred to as `tangle` while the documentation tool is called `weave`. In order to be able to differentiate between source code and documentation, a specific documentation or programming syntax has to be used.

In contrast, elucidative programming [6] does not require such a reorganization of source code. Instead, it provides a mutual navigation between program source files and sections of the documentation. This can take place within a web browser (e.g., through a two-framed layout) or within a programming IDE. Elucidative programming is superior in this aspect to literate programming, as it does not use one complex and often unreadable combined literate document. Moreover, elucidative programming does not interfere with compilers, IDEs, and source code analysis tools that often cannot deal with literate programming files. Nevertheless, elucidative programming still requires a documentation specific language and lacks the integration within mainstream programming environments.

Single/Multi-Source Documentation: Single-source documentation falls into the category of documents with an interwoven representation. Contrary to the literate approach, documentation is added to source code in the form of comments that are ignored by compilers. Tools such as Doxygen [7] or its Java derivative Javadoc [4] are examples of such a documentation approach. A generator (similar to the weave tool within literate programming) can extract these comments and transform the corresponding documentation in a variety of output formats, such as HTML, Latex, PDF, etc. Most tools also provide specific tags within comments that influence the format of the documentation produced or the way documentation pages are linked.

Multi-source approaches maintain source code and documentation separately. This is often the case for high-level documentation such as architectural or requirements documents. Most multi-source documentation tools require that links between source code and documentation are established manually. However, these links can break over time due to changes performed to the artifacts. Furthermore, due to the physical separation of artifacts, multi-source documentation is not always well suited for documentation that is directly linked to source code (e.g., system API documentation and design documents).

Proprietary Systems: Web-based system documentation can be found among many commercial software products such as Microsoft Windows (through its Microsoft Developer Network [8]) or Sun Java [9]. Web-based documentation has the advantage of being easily accessible by a large number of developers around the world at a relatively low cost. Information can be kept up-to-date and querying facilities can be provided. Search engines, such as Google, index the documentation and make it searchable through their portal. Most documentation found online also comes in a downloadable (offline) version and printable format.

Wikis: Current wiki systems have evolved to become an important aspect of today's software documentation environment. Due to their ease of use, collaborative nature, and flexibility, they provide an incentive for developers to document quickly and access documentation easily. Wikis are frequently used by the software engineering community to produce informal software documents such as drafts of design and implementation, or to gather requirements.

3. Implementation

This section describes implementation details of our Software Engineering Editor (SE-Editor) and discusses how our approach supports the substitution of traditional documentation through embedded wikis.

3.1. Web Content Java Editor

Considering the advantages and disadvantages of existing documentation approaches, we have implemented a prototype (SE-Editor) that has the capability to display rich web content within a Java source code editor using the Eclipse framework [10], which offers easy extensibility through plug-ins.

Our SE-Editor enriches the standard JDT source code editor by adding support for web-based content that is displayed directly within the editor through a Standard Widget Toolkit (SWT) browser control. One of the main requirements for the implementation of the SE-Editor is that it remains fully compatible with the existing JDT Java editor. Syntax completion, code highlighting, and underlining of errors are some of these essential productivity tools within development environments which need to be preserved in order to ensure the acceptance by programmers. In our approach, we extended JDT `CompilationUnitEditor`. A web browser cannot be directly displayed within the text control of this editor. Instead, it is necessary to add a SWT `StyleRange` placeholder to the control and then render a SWT `Browser` control on top.

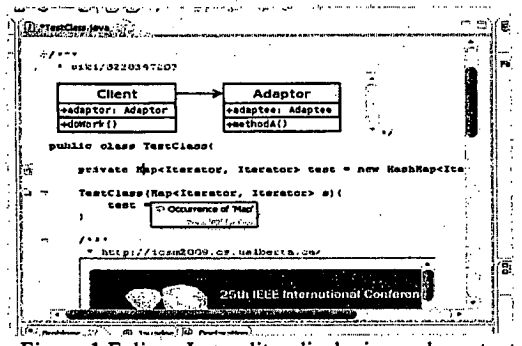

Figure 1 Eclipse Java editor displaying web content

A `PaintObjectListener` then repositions the browser whenever its `StyleRange` is drawn, providing a seamless integration to users. The position for browsers is marked by a source code comment with one slash and three asterisks (/***) that contains a URL. This is compatible with compilers that do not understand Javadoc, as they will interpret the syntax as a standard Java comment (/*), and those that do not understand our three asterisks, as they will interpret the syntax as a Javadoc comment (/**).

Figure 1 shows an example of our SE-Editor displaying embedded rich media web content. In this example, a relative URL (pointing to a wiki with an UML diagram) is resolved against a pre-configured base URL. This allows for different types of content hosting (e.g., locally or remotely hosted content). Other applications of our SE-Editor are playing embedded videos of relevant tutorials (e.g., use of certain design patterns) or play recorded brainstorming sessions. However, in this paper we focus on the integration of wikis as system API integration as explained in the following section.

3.2. Using Wikis as Documentation

In this section, we illustrate and discuss how our rich media content SE-Editor can be applied to replace existing Javadoc documentation with web-based documentation. We consider two scenarios.

New wiki based documentation: For a software developer, the integration of new wiki based software documentation within the source code is a straightforward task. The programmer only has to add a comment (using our three star comment syntax) specifying the URL of a wiki page. In cases where the wiki author does not want to use the class name as part of the URL, a random identification number is created. The wiki content can be further customized through the use of parameters in the URL.

Re-documentation: In this scenario, the basic objective is to allow maintainers to enrich an existing Javadoc documentation with additional content allowing them take advantage of the collaborative nature of wikis. In order to transform Javadoc documentation into a wiki, we use a web service interface (offered by most wikis) in conjunction with Sun's Javadoc Doclet API. A custom doclet allows us to iterate over the existing Javadoc documentation and create wiki pages for each class and paragraphs for each method. The created wiki pages with random URLs are stored in a file, together with their classes and methods they are linked to. In a second step, the Java source files are re-parsed to replace the existing Javadoc comment with the new URLs.

3.3. Overcoming Limitations

Availability. Documentation ideally resides directly within the source code and should remain available without the need for network access. As a result, any interactive content, published as part of wiki pages,

should remain usable even when no Internet connection is available. Similar to Microsoft's Developer Network, packaged information can be made available for offline browsing, while additional online information and content is shown if accessible. Some wikis which can run offline (e.g., through Google Gears) also re-establish interactivity and synchronize themselves once they are online.

Accessibility. Access to source code documentation is normally controlled through a version control system. A similar control structure can be provided within the presented approach for the documentation through the use of wiki systems. Wikis offer a complete login system that can be coupled with a versioning system in order to allow the same users to modify documentation and source code. An interesting aspect of this approach is that access rules could also be set differently in order to restrict who can view (privacy), modify, and extend the existing documentation apart from who can access the source code.

Synchronization. As documentation and source code remain separate, there is a need to define how synchronization is performed (e.g., when changing the name of a class or when moving source code to another file). When copying source code from one file to another, the link between source code and documentation remains intact through the assigned unique URL. The renaming of variables, methods, etc. requires manual effort within the wiki at the moment but could be automated through an extension to the Eclipse refactoring mechanism at a later point.

Versioning. A multitude of parameters can be passed to the website that displays the documentation content. One of these parameters can be the version information of the source code (e.g., revision number) that can be used to display the documentation related to a particular revision or release.

Interoperability. The annotation should not affect the system requirements. Even without our SE-Editor being installed, the source code has to remain compileable by a development environment and easily readable by users. Our approach uses comments to associate a unique URL to a class or method. As the compiler does not interpret comments, our approach remains compatible with other development environments.

4. Conclusions and Future Work

In this paper, we have presented a novel approach to integrate rich web content within a source code editor. We have demonstrated the feasibility of our approach by implementing a fully functional prototype using the Eclipse framework. The same way Web 2.0 applications have supported the social, collaborative, and service-oriented nature of information sharing, our SE-Editor provides a more collaborative software engineering environment to documentation.

As part of our future work, we plan to further evaluate our ideas and collect additional feedback. We have made the SE-Editor Eclipse plug-in available on the Eclipse Plugin Central [11] and encourage the software engineering community to test and comment on it.

Acknowledgement

This research was partially funded by DRDC Valcartier (contract no. W7701-081745/001/QCV).

References

[1] M. M. Lehman and L. A. Belady, *Program evolution: processes of software change*. San Diego, 1985.

[2] M.-A. D. Storey, K. Wong, and H. A. Muller, "How do program understanding tools affect how programmers understand programs?," *Science of Computer Programming*, vol. 36, no. 2-3, pp. 183-207, 3/2000.

[3] R. Brooks, "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies*, vol. 18, pp. 543-554, 1983.

[4] Sun Microsystems, Inc. (2009, April) Javadoc Tool Home Page. [Online]. http://java.sun.com/j2se/javadoc/

[5] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, no. 2, p. 97–111, 1984.

[6] K. Normark et al., "Elucidative Programming in Java," in *Proceedings on the 18th annual Int. conference on Computer documentation*, 2000, p. 483–495.

[7] D. van Heesch. (2009, April) Doxygen. [Online]. http://www.stack.nl/~dimitri/doxygen/

[8] Microsoft Corporation. (2009, April) Microsoft Developer Network. [Online]. http://msdn.microsoft.com/

[9] Sun Microsystems, Inc. (2009, April) Java API Specifications. [Online]. http://java.sun.com/reference/api/

[10] The Eclipse Foundation. (2009, April) Eclipse.org home. [Online]. http://www.eclipse.org/

[11] Eclipse Foundation. (2009, April) Eclipse Plugin Central. [Online]. http://www.eclipseplugincentral.com