Patterns for Consistent Software Documentation

Filipe Figueiredo Correia
Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
filipe.correia@fe.up.pt

Hugo Sereno Ferreira
INESC Porto
Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
hugo.sereno@fe.up.pt

Ademar Aguiar
INESC Porto
Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
ademar.aguiar@fe.up.pt

Nuno Flores
Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
nuno.flores@fe.up.pt

ABSTRACT

Documentation is an important part of the captured knowledge of a software project, providing a flexible and effective way of recording informal contents. However, maintaining documentation's consistency requires a considerable effort. Existing solutions encompass different tools and approaches that support the process of creating, evolving and using documents and other artifacts derived from the software development process. Based on existing literature and expertise, we have identified the key problems and solutions for documentation consistency. In concrete, four distinct patterns and their relations were identified, which are here described — Information Proximity, Co-Evolution, Domain-Structured Information and Integrated Environment.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Documentation; D.2.11 [**Software Architectures**]: Patterns

1. INTRODUCTION

The artifacts created and evolved during from software development process are forms of captured knowledge. They are of different natures and capture several types of information. Some of them are more structured and formal, and thus specialized; others are more flexible and may be used to express virtually any intended topic.

Despite being useful for any software project, the value of software documentation depends on its ability to convey accurate information. It is therefore imperative to assure that it remains consistent. Yet, current production of software documentation still typically focuses on capturing informal, unstructured, human-oriented information. Consequently, ensuring its consistence is a hard to automate process, and therefore highly dependent upon human intervention.

Also, software systems evolve frequently, implying changes in code artifacts along with their related documentation (e.g. requirements, architecture and design documents). In fact, one of the highest costs of maintaining documentation for a large system is to ensure that it is kept in-sync within itself and among its related artifacts, a practice that may require continuous review.

In this context, inconsistencies essentially occur when particular information evolves independently, without the evolution of other related parts. Among other reasons, this may happen because: (a) the author lacks a global knowledge of all artifact dependencies; (b) a particular change cascades into multiple other changes, thus making harder the task of manually tracking them; or (c) as a deliberate way of reducing the maintenance effort.

It is important to note that documents with different subjects, target audiences, and frequencies of use are likely to require different degrees of accurateness. This means that deliberately allowing the documentation to become outdated may be a reasonable choice in some circumstances. For example, some types of documents are useful only within a specific time period, and there may be no value in updating them beyond it. For some projects, it is therefore advisable to take an agile approach towards documentation, producing and evolving it *just enough* and *when needed*, to better satisfy the project at hands.

Patterns addressing the topic of software documentation have previously been documented. The book *"Agile Documentation: A Pattern Guide to Producing Lightweight Documents for Software Projects"* [18] introduces a set of patterns covering a wide scope of concerns in the production of software documentation, and the pattern language *"Patterns for Documenting Frameworks"* [1, 3, 4, 5, 6] has focused on framework documentation in particular. Ambler's work on agile documentation and modeling is also very relevant to this topic [9, 8].

Although having some commonalities with the aforementioned works, the patterns presented in this paper address software documentation from a consistency standpoint, keep-

PLoP'09 August 28-30, Chicago, Illinois, USA

ing other important issues in view. They are meant to support teams on the selection of documentation-related tools, and to help tool developers to implement the most appropriate techniques to support documentation consistency.

2. PATTERNS OVERVIEW

An overview of the patterns, and how they relate to each other, is depicted in **Figure 1**.

Since the same information may exist, partially, or totally, in more than one document, there are implicit relations between those contents. With no easy way of recovering these relations, the effort of maintaining consistency increases, as information may be duplicated and scattered over several documents. INFORMATION PROXIMITY focuses on establishing and using explicit relations among different artifacts.

Software artifacts change to better respond to new needs, and documentation is required to accompany this evolution. However, due to the aforementioned intrinsic relations between different parts of documentation, it is common that locally introduced changes may render other documentation parts inconsistent or obsolete. CO-EVOLUTION focuses on strategies to update documentation while maintaining its consistency.

DOMAIN-STRUCTURED INFORMATION deals with structuring contents, with the main objective of automating the process of assessing consistency.

Different types of artifact may require different types of authoring tools. INTEGRATED ENVIRONMENTS articulate the use of different tools and allow them to be used uniformly.

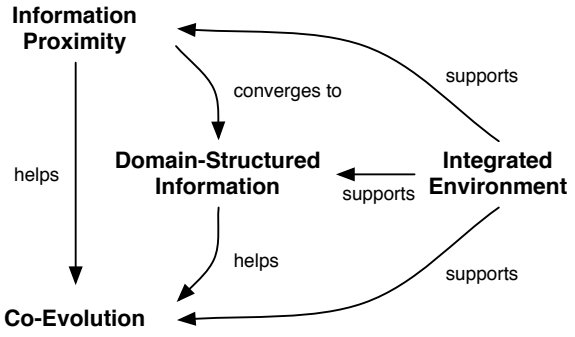

Figure 1: Overview of the patterns and their relationships.

3. INFORMATION PROXIMITY

Software documentation can be captured as a set of documents of different types and purposes. Thus, they may sometimes address the same information from different perspectives. However, as documentation evolves, the effort of keeping them consistent rises due to the proliferation of duplicate and closely related contents.

3.1 Problem

How to preserve documentation consistency when fragments of related content are scattered across documents?

Having related information fragments in *close proximity* of each other, and having *multiple uses* of the same information, across a set of different documents, are common needs in software documentation.

There is value in a well established *separation of concerns* among different artifacts, as it allows them to be reused more easily, but there may exist the need to tailor them to specific contexts, as to maintain a *high fitness for purpose*.

3.2 Solution

Keep related information fragments easily accessible from each other, using a single source, links, views, or transclusion, for example, so that it may be easier to assess if they are in-sync.

The concrete approach to keep related information fragments close to each other greatly depends on the purpose of the document being produced. Four different alternatives are considered here and further described in the following sections.

The use of **links** is a good choice if the related contents are not meant to be part of the same document, and **single-source**, **transclusion** and **views** may be used otherwise. **Single-source** may be a good choice if the related contents can be made part of the same artifact (e.g. source-code and API documentation are both frequently expressed as text in the same file), and **transclusion** or **views** should be used when the related contents already exist in other artifacts. A **view** may be seen as a special case of **transclusion**, in which all contents already exist in other artifacts, while, with **transclusion**, the document being authored has contents of it's own, and only parts of it are used from other documents.

3.2.1 Variant: Links

Use explicit relations between different resources, so that related contents are kept separated and readers may easily travel between them.

Creating links between contents allows to explicitly relate them, while keeping them as separate entities from both the authors' and readers' viewpoint. Links allow readers to quickly reach related pieces of information, hence making easier the process of maintaining them consistent.

The following consequences should be considered when applying this technique:

Web of Documents. LINKS leads to the creation of relations among the existing contents, forming a web of related documents.

Reuse. Although not a reuse technique per se, links reduce the need to duplicate contents.

Reachability. Even if information is created, stored and presented separately, one may easily reach related contents. However, it is worth noting that the achieved proximity between contents is not necessarily bidirectional — linking the artifact A to artifact B makes B closer to A, but the opposite isn't necessarily true, e.g. if backlinks are not supported.

Effort. Removing duplicated contents requires an additional effort to detect common information fragments and to restructure them accordingly.

3.2.2 Variant: Single Source

Capture related information fragments in the same artifact, so that they may be easily maintained close to each other.

Although this isn't possible for all kinds of information, some can be captured together in the same artifact. Doing so allows related information fragments to be kept consistent since the author can more easily travel between them. This is thus a form of *physical information proximity* as the contents are stored together with the goal of being presented together to readers and authors.

The following consequences should be considered when applying this technique:

Single artifact. The reader is presented with a document based in a single artifact, although including different types of information.

Reuse. Capturing different information fragments in the same artifact makes them less modular, and thus more difficult to reuse.

Reachability. Related information is stored and presented to the reader close to each other.

Effort. The flow of creating documentation is better than if these contents were kept separately.

3.2.3 Variant: Transclusion

Import the contents of an information fragment into a document by using a reference to it.

Isolating fragments of information as individual units eases their use for different purposes. Transclusion consists of creating references to information fragments on a document in such a way that they are presented to the reader as part of the document itself. Documents can be composed this way to fit the author's intent. Transclusion is a form of *virtual information proximity* since the contents are stored separately.

The following consequences should be considered when applying this technique:

Document-Oriented. Although leading to the creation of individual information fragments, the final result is a *document* tailored to a specific purpose.

Reuse. Abstracting information into individual units also allows them to be reused more effectively.

Reachability. Information that may me created and stored separately is presented to the reader near each other. However, this proximity may be unidirectional — transcluding the artifact A into artifact B makes A closer to B, but the opposite isn't necessarily true.

Effort. The flow of creating documentation may be hindered when authors are faced with the need to abstract existing information into new distinct units.

3.2.4 Variant: Views

Create a virtual document, composed by different individual fragments of information.

Views may be called *virtual documents*, as they have no content of their own. Instead, they filter, transform and combine contents according to a desired format into a single document. It is thus a form of *virtual information proximity* in the sense that contents are stored separately.

The following consequences should be considered when applying this technique:

Document-Oriented. The final result is a *document* tailored for a specific purpose, although contents may be woven together from several sources.

Reuse. Weaving contents into a view is an effective way of reusing them.

Reachability. Information that may be created and stored separately is presented to the reader near each other. However, this proximity may be unidirectional — the contents may be *close* to one another on the context of a given view, but it may not be possible to reach one from the other outside of this context.

3.2.5 Creating Heterogeneous Documents

Using artifacts of different types to create a document gives rise to a heterogeneous document. This may be achieved by using techniques, like **single-source**, **transclusion** or **views**, but different types of information may require different authoring tools, making information fragments more difficult to combine. From these techniques, **single-source** in particular is restrained to formats that may be combined into the same file, while **transclusion** and **views** may more easily be used with different types of content.

3.3 Related Patterns

This pattern helps Co-Evolution, as keeping related contents near each other helps to change them together. Moreover, the creation of explicit relations frequently implies conferring more structure to the contents, which may converge to Domain-Structured Information.

As with the other patterns in this paper, Information Proximity greatly benefits from an appropriate tool support, which may be leveraged by an Integrated Environment.

Wikis [18] address the use of **links** but goes beyond the creation of explicit and navigable relations between resources, addressing the collaborative nature of this kind of systems.

Single-source is an approach similar to the one taken by the Code-Comment Proximity pattern [18], but goes beyond source code and comments, not restricting itself to any particular type of information.

Transclusion is similar to the Import by Reference pattern [18], although it focuses on consistency maintenance.

3.4 Known Uses

Hypertext-based systems in general, of which wikis are a good example, allow to establish **links** between related resources. The term **transclusion** appeared initially in the context of hypertext-based systems. For example, Mediawiki, the wiki engine powering Wikipedia, uses this concept to allow the inclusion of repetitive blocks of content. XSDoc [7, 2] is a wiki engine oriented for software development that uses **transclusion** to weave together heterogeneous artifacts, thus giving origin to heterogeneous documents.

Using the technique of *Code Annotations* (based on **single source**), documentation (or parts of it) can be generated from a unified representation of textual descriptions and source code. It is primarily used in the creation of API documentation and is supported by several tools: Javadoc [15] is one of the first known uses of the technique, as is Autoduck [10], a tool supporting code annotations in C++. The .NET framework uses XML in code annotations to produce compendiums of API documentation (CHM, HTML, etc.), in-editor assistance, and code-completion.

Views are frequently the product of an automatic generation process, in which several contents are combined according to a pre-established document form — some tools exist that support this approach [11].

Literate Programming (LP) [17] combines textual descriptions and source code in a **single source file**, and provides the mechanisms to extract such different contents to different artifacts whenever required. The LP tool set *dotNoweb* [20] further allows to combine textual descriptions and source code with diagrams expressed using the *dot* language. LP systems also usually provide a form of **transclusion**, by allowing the creation of information fragments — *chunks* – which can then be (re)used multiple times across several documents.

Elucidative Programming [21] is a documentation technique that relies on the creation of **links** between source code and documentation, allowing to mutually navigate between them.

Several office software suites, such as Microsoft Office and OpenOffice, allow combining different kinds of artifact in a same document, also resulting in heterogeneous documents. Some uses of Literate Programming, such as VDMTools, directly parse and write *.rtf* documents which have native support for images.

4. CO-EVOLUTION

Software documentation can be captured as a set of documents of different types and purposes. Thus, they may sometimes address the same information from different perspectives. However, as documentation evolves, the effort of keeping them consistent rises due to the proliferation of duplicate and closely related contents.

4.1 Problem

When to update a related piece of information in documentation?

Changes are made by the authors having the introduction of added value in view. However, changes required to ensure consistency don't always provide *immediate benefits*, and may shift the author's main *focus*.

Furthermore, the primary *goal* of the project will not always be the same. For example, during an inception phase, the *change rate* at which documented artifacts evolve is usually high. This means that changing just enough of the related information fragments might be the best choice. On the other hand, deployment phases may benefit from producing documentation with a higher level of *detail*.

Finally, *tracking* all the required changes may be difficult to carry out without any kind of *auxiliary support*, since it is easy to disregard global consequences during local modifications.

4.2 Solution

When a change is introduced, update the related information parts.

If all the related pieces aren't updated at the same time, they may grow harder to resync as time passes. Two variants to the co-evolution of contents are considered here and are further described on the following sections.

Synchronous co-evolution is a good option when it is important that documentation is kept consistent at all times, or if the effort of recovering consistency at a later time is high. **Time-shifted co-evolution** may be used when the effort of recovering consistency is reasonable. This may happen when it is not difficult to assess the existence of relations between contents and the presence of inconsistencies between them.

4.2.1 Variant: Synchronous Co-Evolution

Whenever a change is introduced, update every related piece of information.

Although the quantity of information to be updated may be considerable, the most reliable way of ensuring consistency is to update all related information at the same time. Changes are made in small increments, in order to reduce the risk of forgetting to update something.

The following consequences should be considered when applying this technique:

State. Documentation is always in a consistent state.

Focus. The focus of the author on the task at hand is harder to maintain, as some of the changes she is required to do are not directly related with her main goal.

Effort. Introducing a change to a document carries a *higher up-front cost* — it may take more time than expected, as all the related contents will have to be updated at the same time.

Efficiency. If a particular fragment has several others that depend on it, and it has a high rate of change, it may be inefficient to keep consistency at all times.

4.2.2 Variant: Time-Shifted Co-Evolution

Whenever a change is introduced, provide mechanisms to track the pending related changes, and update the most relevant pieces of information only when needed.

Related contents don't need to be updated simultaneously if the changes that are made are in some way recorded. Authors will be able to, at a later time, assess which are the pending related changes, and evolve documentation to a consistent state as soon as they are addressed.

For example, using the concept of auditable document (see section 4.2.3) authors may gain more awareness of the required modifications, facilitating the detection of changes that are still to be applied.

The following consequences should be considered when applying this technique:

State. Consistency is not kept at all time.

Focus. The author may focus solely on the task at hand, leaving related changes for later.

Effort. Only the changes that bring *short term benefits* are required to be made, and related changes may be deferred to a later time.

Efficiency. The task of updating documentation is distributed across the development process, as documentation may be updated only when necessary. However, the author may be faced with the additional effort of tracking which information needs to be updated, even if tools that support this task may exist.

4.2.3 Creating Auditable Documents

An auditable document makes it possible to assess at any time who, how, why, and what has been produced, by tracking information regarding the authoring process.

Being able to follow and understand how a document is evolved makes the entire process more *transparent* and *traceable*. However, it is important to note that the tracking mechanisms may increase the *complexity* of authoring the document, and the extra information that is recorded may increase the *storage space consumption*. Furthermore, for heterogeneous documents, tracking the evolution as a whole may involve tracking different types of artifact.

4.3 Related Patterns

Domain-Structured Information supports Co-Evolution, since making richer information available allows tracking the information that needs to be co-evolved in greater detail. Information Proximity helps this pattern too, since having related contents easily reachable from one another assists in determining which contents are affected by a particular change.

Some patterns already describe the use of auditable documents in more concrete scenarios, namely Document History [18] focuses on maintaining a list of past versions of a document, and Annotated Changes [18] provides a way to directly record, inside a document, which of its parts have recently been modified.

4.4 Known Uses

Literate Programming and Code-Annotations, such as Javadoc, may be regarded as a way of supporting **synchronous co-evolution**, as providing Information Proximity helps to co-evolve related information parts simultaneously.

Solutions that allow auditable documents to be produced support **time-shifted co-evolution**. Wiki engines and version control systems are good examples of such solutions, which allow to track how documents evolve and support assessing which changes are required to maintain consistency.

It is common for text processors to provide a *track changes* feature, which is a form of Annotated Changes. This feature may be used by authors and readers to track the changes the document has recently gone through. Although this makes the document auditable to a certain point, it is usually very limited in time.

5. DOMAIN-STRUCTURED INFORMATION

Documentation usually follows a text-oriented structure, using elements such as titles, paragraphs, lists, tables, etc. Although it provides a lot of flexibility, the degree to which a document is relevant will depend on how well it serves its purpose and accurately expresses the intended ideas. Moreover, the same piece of information may be better conveyed using different perspectives, intrinsically related to each other.

The main reason why maintaining documentation requires continuous review is that relations between documentation parts aren't explicitly formalized. This decreases the capability to automatically process it, i.e. in order to automatically assess its consistency.

5.1 Problem

How to structure the information in documentation?

As mentioned before, textual documentation is a *flexible* way of capturing knowledge. While this flexibility is an important asset, *formalizing* the content itself makes information less subject to multiple interpretations, and allows it to be automatically processed.

However, the mechanisms used to allow a degree of formalization higher than that provided by simple textual descriptions may affect the *simplicity* in producing documentation.

5.2 Solution

Organize contents according to their domain, so that the information form directly relates to domain concepts.

Textual documentation doesn't provide the mechanisms to formally express the relations between the concepts being documented. Structuring the contents around the domain concepts provides the support to automatically assess the existence of inconsistencies, and prevents the introduction of new ones.

The following consequences should be considered when applying this pattern:

Flexibility. Some flexibility is lost whenever information has to follow a predefined structure.

Automation. The use of a domain-oriented structure with well defined semantics makes information less open to different interpretations, and allows it to be processed by computers.

5.3 Related Patterns

The individual information units often required by Information Proximity tend to converge to Domain-Structured Information, as the advantages of organizing the contents around domain concepts emerge. This pattern also supports Co-Evolution, as it provides a richer base of traceable information.

As with the other patterns in this paper, Domain-Structured Information requires appropriate tool support, and may benefit from the use of an Integrated Environment.

This pattern is similar to Structured Information [18], in that it also address how documents' contents are organized. However, Domain-Structured Information focuses on formalizing contents according to the information's domain, with the aim of automating consistency assessment, while Structured Information focuses mainly in structuring contents to ease the perception of the readers.

5.4 Known Uses

Code comments are a form of source code documentation. Code annotations, such as Javadoc comments [15], add an additional level of structure to source code comments, formalizing information elements with a lower granularity. Javadoc allows to describe elements such as method parameters, authors, creation dates and references, among others.

Semantic Wikis support DOMAIN-STRUCTURED INFORMATION, and some semantic wiki engines may automatically detect existing inconsistencies with the use of reasoners [13].

Some wiki engines allow templates to be applied for very specific purposes. Mediawiki allows the creation of sidebar templates, through which one may provide structured information.

Systems taking an object-oriented approach to documentation have also been use in the past [19, 12].

6. INTEGRATED ENVIRONMENT

Working with different kinds of artifacts frequently implies the use of specialized and independent tools for each of them. Although such artifacts are sometimes strongly related, these tools don't necessarily interoperate, making the artifacts more difficult to combine and confront, and the authoring environment heterogeneous and more difficult to use.

6.1 Problem

How to support the maintenance of consistency between independent artifacts with related content?

Tools that deal with a *wide range of artifacts* usually provide a more *homogeneous* and *interoperable* environment, although they tend to be not as *powerful* and *simple* as *specialized tools*.

6.2 Solution

Use an integrated environment, where several types of artifact and their relations may be maintained uniformly.

An integrated environment goes beyond the capabilities that general purpose tools possess. It supports handling several types of artifact, providing specialized features for each of them and an infrastructure through which they interoperate.

This supports strategies of documentation maintenance that focus on bridging related information parts regardless of their nature.

The following consequences should be considered when applying this pattern:

Specialization. Integrated environments strike a balance between a generic approach, in which tools may handle several types of artifacts with a basic level of functionality, and a specialized approach, in which exists a deeper support for a selected set of artifact types.

Simplicity. While potentially making each tool more complex individually, their overall simplicity is increased by providing a more homogeneous usage.

Interoperability. An integrated environment coordinates the several tools it provides, and supports their interoperability.

6.3 Related Patterns

INTEGRATED ENVIRONMENT directly contributes to the remaining patterns of this paper by orchestrating the several tools involved. It is also directly related to the pattern FEW TOOLS [18], which addresses the notion that supporting the creation of documentation with too many and unconnected tools may become a burden to authors.

6.4 Known Uses

Eclipse and Visual Studio are examples of integrated environments that combine different kinds of artifact and tools, supporting and articulating their work.

Trac [14] and Redmine [16] are Web-based environments that integrate different kinds of information, including textual descriptions supported by a wiki, source-code browsing, milestone management, issue-tracking, etc.

7. ACKNOWLEDGMENTS

We would like to thank our shepherd, Ralph Johnson, for his help on the process of improving this paper, as well as to all the participants of the writer's workshops where this paper was discussed, for their insightful comments and suggestions — the *Process Writers' Workshop* at *PLoP'09*, with Christian Crumlish, Pam Rostal, Robert Hanmer, Xiaohong Yuan and Zhen Jiang, and the *MiniPLoP* workshop at *OOPSLA'09*, with Dave West, Eduardo Fernandez, Joseph Yoder, Pam Rostal and Richard Gabriel. A special thanks to Bob Hanmer for several suggestions regarding the English language used on the paper.

We would also like to thank the *Portuguese Foundation for Science and Technology* and *ParadigmaXis, S.A.* for sponsoring this work through the doctorate scholarship grant SFRH / BDE / 33298 / 2008.

8. REFERENCES

[1] A. Aguiar and G. David. Patterns for documenting frameworks — Part I. Helsinki, Finland, Sept. 2005.

[2] A. Aguiar and G. David. WikiWiki weaving heterogeneous software artifacts. In *Proceedings of the 2005 international symposium on Wikis*, pages 67–74, San Diego, California, 2005. ACM.

[3] A. Aguiar and G. David. Patterns for documenting frameworks — Part II. Irsee, Germany, July 2006.

[4] A. Aguiar and G. David. Patterns for documenting frameworks — Part III. Portland, Oregon, USA, Oct. 2006.

[5] A. Aguiar and G. David. Patterns for documenting frameworks: customization. In *Proceedings of the 2006 conference on Pattern languages of programs*, pages 1–10, Portland, Oregon, 2006. ACM.

[6] A. Aguiar and G. David. Patterns for documenting frameworks - process. Recife, Brazil, May 2007.

[7] A. Aguiar, G. David, and M. Padilha. XSDoc: an extensible wiki-based infrastructure for framework documentation. Alicante, Oct. 2003.

[8] S. Ambler. Agile/Lean Documentation: Strategies for Agile Software Development. http://www.agilemodeling.com/essays/agileDocumentation.htm.

[9] S. Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. Wiley, 1st edition, Apr. 2002.

[10] E. Artzt. Autoduck user's guide. Technical report, 2000.

[11] J. Bayer and D. Muthig. A view-based approach for improving software documentation practices. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, page 10 pp., 2006.

[12] B. Childs and J. Sametinger. Literate programming and documentation reuse. In *Software Reuse, 1996., Proceedings Fourth International Conference on*, pages 205–214, 1996.

[13] B. Decker, E. Ras, J. Rech, B. Klein, and C. Hoecht. Self-organized reuse of software engineering knowledge supported by semantic wikis. In *Proceedings of the Workshop on Semantic Web Enabled Software Engineering (SWESE)*, Nov. 2005.

[14] Edgewall Software. The Trac Project — Integrated SCM & Project Management — http://trac.edgewall.org/ [accessed on 2009/12/01].

[15] L. Friendly. The design of distributed hyperlinked programming documentation. In *Proceedings of the International Workshop on Hypermedia Design*, Montpellier, France, 1995.

[16] Jean-Philippe Lang. Redmine — http://www.redmine.org/ [accessed on 2009/12/01].

[17] D. E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, 1984.

[18] A. Ruping. *Agile Documentation: A Pattern Guide to Producing Lightweight Documents for Software Projects*. John Wiley & Sons, Inc., 2003.

[19] J. Sametinger. Object-oriented documentation. *SIGDOC Asterisk J. Comput. Doc.*, 18(1):3–14, 1994.

[20] A. Sousa. dotNoweb User's Guide. Technical report, 2005.

[21] T. Vestdam and K. Nørmark. Aspects of internal program documentation-an elucidative perspective. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 43–52, 2002.