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

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

Ademar Aguiar
Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
ademar.aguiar@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 raises several issues. The present pattern language describes complementary solutions for managing and ensuring the consistency of software documentation, by focusing on different tools and approaches which support such activities. Ten distinct patterns and their relations are described — Views, Transclusion, Links, Single Source, Heterogeneous Document, Synchronous Co-Evolution, Time-Shifted Co-Evolution, Auditable Document, 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

Artifacts derived from the process of software development are forms of captured knowledge. They are of different natures, they capture several types of knowledge. 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.

Documentation is as valuable as 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. Therefore, ensuring its consistency reveals to be a process both hard to automate, and highly dependent upon human intervention.

Also, it's usual for software systems to be frequently evolving, thus requiring changes in code artifacts along with related documentation (e.g. requirements, architecture and design documents). In fact, one of the highest costs of maintaining documentation for a large system is ensuring it's kept in-sync with its related artifacts, a practice that may require continuous review.

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

Some patterns which address the topic of software documentation have already been documented. The book *"Agile Documentation: A Pattern Guide to Producing Lightweight Documents for Software Projects"* [14] introduces a set of patterns, covering a wide scope of concerns in the production of software documentation. The pattern language *"Patterns for Documenting Frameworks"* [1, 3, 4, 5, 6] has focused on framework documentation in particular.

Although having things in common with the aforementioned works, the patterns presented in this paper address software documentation from a consistency standpoint, while not disregarding other important issues. They are meant to help both the documentation producers to select the right tools, and tool developers to implement the most appropriate techniques.

2. OVERVIEW

Figure 1 depicts an overview of the pattern language, presenting each pattern and the relationships between them. They are organized in groups which reflect similarities between both their contexts and the problems they address.

2.1 Information Proximity Patterns

Software documentation may include the same information in different documents, produced for different purposes. However, the relations among these dependent pieces of informa-

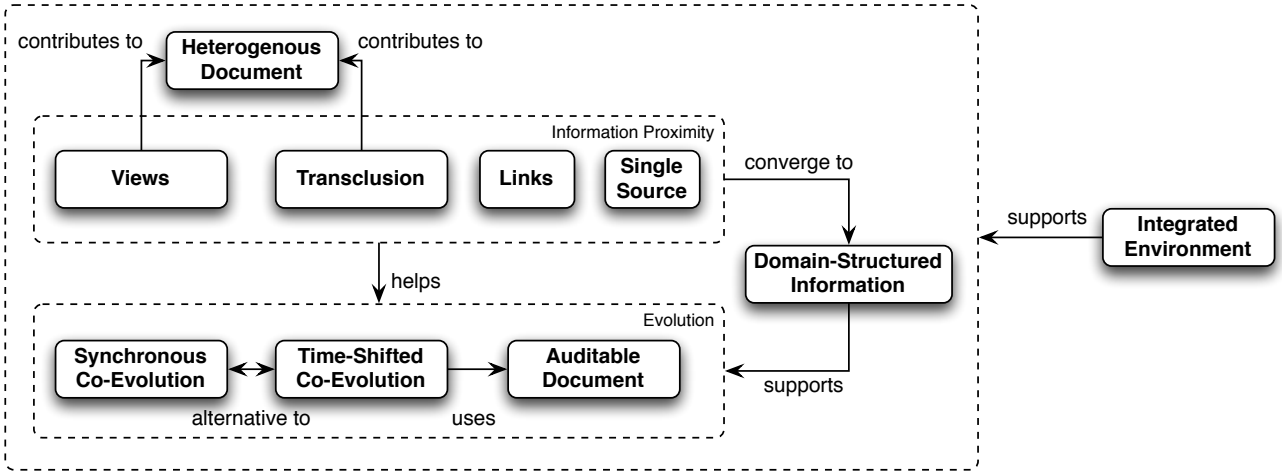

Figure 1: Overview map of consistency patterns and their relationships.

tion only exist implicitly. This may happen because (a) the process hasn't yet led the authors to explicitly capture such contents, (b) they may have appeared at different times or (c) they were produced using different tools. With no way of recovering these relations, the effort of maintaining consistency increases, as information is duplicated and scattered over several documents.

The class of patterns that keep related contents at close range, in order to facilitate their maintenance, is called *Information Proximity Patterns*. While all these patterns focus on establishing relations among different artifacts, they may have to be brought into balance with the overall goal of documentation: to capture concise and targeted information.

2.2 Evolution Patterns

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 related pieces inconsistent or obsolete.

Evolution Patterns focus on the strategies to update related parts of documentation, considering the moment in which it should be done. While maintaining documentation consistency at all times ensures the value of documentation is kept, it might not always be cost-effective, and deferring these tasks may reveal to be the most appropriate option.

Furthermore, documentation may be written cooperatively by different producers (i.e. towards the same goal), often resulting in the lack of a global view of their contributions. Awareness of what contributions have been made, where, and by whom, allows to track changes and possible defer consistency maintenance tasks for a later time.

3. VIEWS

In software documentation, the description of artifacts can be captured using different documents, which have different purposes. Maintaining them consistent requires an non-neglectable effort. As documentation evolves, such effort rises due to the proliferation of duplicate and related contents.

This pattern considers that a document is being produced and **all of its contents** originate from external artifacts.

3.1 Problem

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

This issue appears from the need of having *multiple uses* of the same information, across a set of different documents, although there would be advantages in capturing a *single existence*, to ease maintenance.

Achieving an appropriate *separation of concerns* is also a relevant issue, in which case individual fragments of information could be reused more easily. However, the associated effort might not be negligible, specially when tailoring artifacts for different contexts while maintaining an *high fitness for purpose*.

3.2 Solution

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 combine contents by weaving them together according to a desired format. It is a form of *virtual information proximity* in the sense that contents are stored separately, even though they are presented together to the readers.

The following consequences should be considered when applying this pattern:

Document-Oriented. Although leading to the creation of individual information fragments, the intended result provides a cohesive *document* to the reader.

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

Information Proximity. Information that is created and stored separately is presented to the reader in close range of each other, simplifying consistency maintenance.

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

This pattern may be used to create an HETEROGENEOUS DOCUMENT, even though the view's contents would not necessarily be of different types.

As every other *Information Proximity* pattern, it helps *Evolution*, because keeping related contents at close range helps changing them altogether. Moreover, the creation of different information fragments frequently converges to DOMAIN-STRUCTURED INFORMATION, which also supports evolution, by providing a richer base of trackable information.

As with the other patterns in this pattern language, this pattern greatly benefits from an appropriate tool support, which may be leveraged by an INTEGRATED ENVIRONMENT.

3.3 Known Uses

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 [9].

4. TRANSCLUSION

In software documentation, the description of artifacts can be captured using different documents, which have different purposes. Maintaining them consistent requires an non-neglectable effort. As documentation evolves, such effort rises due to the proliferation of duplicate and related contents.

This pattern considers that a document is being produced and **portions** of its contents are imported from other documents.

4.1 Problem

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

This issue appears from the need of having *multiple uses* of the same information, across a set of different documents, although there would be advantages in capturing a *single existence*, to ease maintenance.

Achieving an appropriate *separation of concerns* is also a relevant issue, in which case individual fragments of information could be reused more easily. However, the associated effort might not be negligible, specially when tailoring artifacts for different contexts while maintaining an *high fitness for purpose*.

4.2 Solution

Include the contents of an information fragment in a document by using a reference to it.

By isolating fragments of information as individual units, one eases their use for different purposes. Transclusion consists in 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.

The following consequences should be considered when applying this pattern:

Document-Oriented. Although leading to the creation of individual information fragments, the intended result provides a cohesive *document* to the reader.

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

Information Proximity. Information that is created and stored separately is presented to the reader in close range of each other, simplifying consistency maintenance.

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

This pattern may be used to create an HETEROGENEOUS DOCUMENT, even though the view's contents would not necessarily be of different types.

As every other *Information Proximity* pattern, this pattern helps *Evolution*, as keeping related contents at close range helps changing them altogether. Moreover, the creation of different information fragments frequently converges to DOMAIN-STRUCTURED INFORMATION, which also supports evolution, by providing a richer base of trackable information.

As with the other patterns in this pattern language, this pattern greatly benefits from an appropriate tool support, which may be leveraged by an INTEGRATED ENVIRONMENT.

This pattern is similar to IMPORT BY REFERENCE [14], although it focuses on consistency maintenance.

4.3 Known Uses

The concept of TRANSCLUSION appeared in the context of hypertext-based systems, and it's still used nowadays. For example, Mediawiki, the wiki engine powering Wikipedia, uses this concept to allow the inclusion of repetitive blocks of content.

Another example is that of Literate Programming (LP) [13]. These approaches provide means of creating information fragments — *chunks* – which can then be (re)used multiple times across several documents.

5. LINKS

In software documentation, the description of artifacts can be captured using different documents, which have different purposes. Maintaining them consistent requires an non-neglectable effort. As documentation evolves, such effort rises due to the proliferation of duplicate and related contents.

This pattern considers that a document is being produced and its contents are related with other contents, though they are **not meant to be part of the document being authored**.

5.1 Problem

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

This issue appears from the need of having *multiple uses* of the same information, across a set of different documents, although there would be advantages in capturing a *single existence*, to ease maintenance.

Achieving an appropriate *separation of concerns* is also a relevant issue, in which case individual fragments of information could be reused more easily. However, the associated effort might not be negligible, specially when tailoring artifacts for different contexts while maintaining an *high fitness for purpose*.

5.2 Solution

Use explicit relations between different resources, so that related contents are kept separated, but readers may easily navigate 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 documentation users to quickly reach related pieces of information, thus easing the process of maintaining them consistent.

The following consequences should be considered when applying this pattern:

Web of Documents. This pattern 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 provide the ability to reduce the need to duplicate contents.

Information Proximity. Even if information is created, stored and presented separately, one may easily reach related contents.

Effort. If links are used to remove duplicated information, there will be an additional effort of abstraction. Nonetheless, the existence of explicit relations represents a new concern of maintenance as documentation evolves.

As every other *Information Proximity* pattern, this pattern helps *Evolution*, as keeping related contents at close range helps changing them altogether. Moreover, the creation of explicit relations frequently converges to Domain-Structured Information, which also supports evolution, by providing a richer base of trackable information.

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

Although Wikis [14] make heavy use of hypertext, this pattern's scope goes beyond the creation of explicit and navigable relations between resources, addressing the collaborative nature of this kind of systems.

5.3 Known Uses

Hypertext-based systems in general, of which wikis are a good example, allow to establish links between related resources.

Elucidative Programming [16] is a documentation technique that relies on the creation of links between source code and documentation, allowing to mutually navigate between them.

6. SINGLE SOURCE

In software documentation, the description of artifacts can be captured using different documents, which have different purposes. Maintaining them consistent requires an non-negligible effort. As documentation evolves, such effort rises due to the proliferation of duplicate and related contents.

This pattern considers related contents are being produced, and **they may be made part of the same document, if required**.

6.1 Problem

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

Both having related information fragments at *close range* of each other, and having *multiple uses* of the same information, across a set of different documents, are common needs in software documentation. However, there is also value in a well established *separation of concerns* among different artifacts, as it allows them to be reused more easily, even if that implies tailoring them for different contexts while maintaining an *high fitness for purpose*.

6.2 Solution

Capture related information fragments in a same artifact, so that one may be easily reached from the other.

Although not 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, as the author may more easily reach them. This is thus a form of *Physical Information Proximity* as the contents are stored together, with the objective of being presented together to the author.

The following consequences should be considered when applying this pattern:

Single artifact. The reader is presented with a document based in a single artifact, but that includes different types of information.

Reuse. Capturing different information fragments in the same artifact allows them to be reused less effectively.

Information Proximity. Related information is stored and presented to the reader at close range of each other, simplifying consistency maintenance.

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

As an *Information Proximity* pattern, this pattern helps *Evolution*, as keeping related contents at close range helps changing them altogether. Moreover, organizing different types of contents within a single artifact frequently converges to Domain-Structured Information, which also supports evolution, by providing a richer base of trackable information.

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

This pattern is similar to Code-Comment Proximity [14], as both use the same base approach. However, Single-Source goes beyond source code and comments, and doesn't restrict itself to a particular type of information.

6.3 Known uses

Literate Programming 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 Literate Programming tool set *dotNoweb* further allows to combine textual descriptions and source code with diagrams expressed using the *dot* language.

Using the technique of *Code Annotations*, 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 be several tools: Javadoc [12] is one of the first known uses of the technique, as is Autoduck [8], a tool supporting code annotations in C++. The .NET framework uses XML in code annotation to produce both compilations of API documentation (CHM, HTML, etc.) and in-editor assistance, as code-completion.

7. SYNCHRONOUS CO-EVOLUTION

Evolving information in the documentation usually requires other documents and artifacts to be updated, in order to maintain them consistent.

7.1 Problem

When to update a related piece of information in documentation?

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

Furthermore, the approach to a given task may vary between a depth-first or a breath-first approach, depending on the desired *goal*. For example, during an inception phase, the *change rate* at which documented artifacts evolve is usually high, thus changing just enough in every piece of related information would be regarded as more productive. In opposition, deployment phases of development may benefit from an high level of detail about a single piece of information.

Finally, *tracking* all the required simultaneous changes may be difficult to carry out by *memory* alone, since it is easy to forget or disregard global consequences during local changes.

7.2 Solution

Whenever a change is introduced, update every related piece of information, leaving the details for later.

When there are related pieces of information, changing one may render others inconsistent. If we don't update them at the same time, and we don't record what needs to be updated, they grow harder to resync as time passes. Although the quantity of information to be updated may be considerable, the most reliable way of ensuring consistency is to synchronously update all related information, leaving out details for later improvement.

The following consequences should be considered when applying this pattern:

Memory. Because all the required changes are introduced simultaneously, the risk of forgetting to update every relevant piece of information is reduced. With the proper support of tools, this risk can be rendered almost non-existent.

Objective. By lowering the level of detail (and depending on the goal of the current task), the effort of a simultaneous update can be reduced.

Incremental. Updating every document to reflect every change may hinder both experimentation and focus on earlier phases of development, where new ideas are introduced and old ideas change very frequently. Keeping documentation fully consistent in these phases doesn't provide substantial benefits, while making an incremental approach more difficult. Using this pattern is recommended on later phases.

Concentrated effort. When introducing a change which triggers the need to update several pieces of information and documents, that change can be said to carry an *high up-front cost*. However, it is frequent that only the original change will provide short term benefits.

Using Domain-Structured Information supports Synchronous Co-Evolution, since making richer information available allows to more easily track which information needs to be co-evolved.

All of the *Information Proximity* patterns help this pattern in a similar way, since having related contents easily reachable from one another allows to determine which contents are affected by a particular change.

TIME-SHIFTED CO-EVOLUTION is a direct alternative to this pattern, and the choice on which to use will greatly dependent upon how easy it is to decide which contents are related with each other and how easy it is to track, at at later time, which changes were introduced.

7.3 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.

8. TIME-SHIFTED CO-EVOLUTION

Evolving information in the documentation makes it necessary to update related documents and artifacts in order to maintain them consistent.

AUDITABLE DOCUMENTS are being produced; allowing to track how documentation is evolving.

8.1 Problem

When to update a related piece of information in documentation?

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

Furthermore, the approach to a given task may vary between a depth-first or a width-first approach, depending on the desired *goal*. For example, during an inception phase, the *change rate* at which documented artifacts evolve is usually high, thus changing just enough in every piece of related information would be regarded as more productive. In opposition, deployment phases of development may benefit from an high level of detail about a single piece of information.

8.2 Solution

Whenever a change is introduced, update only the most relevant piece of information, and provide mechanisms to track the related required changes.

When there are related pieces of information changing one may render others inconsistent. If we don't update them at the same time, and we don't record what needs to be updated, as time passes they grow harder to resync. Using AUDITABLE DOCUMENTS one may keep track of which changes were introduced, thus facilitating the detection of pending changes in related pieces of information, in order to keep documentation consistent.

The following consequences should be considered when applying this pattern:

Memory. Because all the required changes are introduced at different times, there is the risk of forgetting to update the relevant pieces of information. Using AUDITABLE DOCUMENTS reduces this risk.

Objective. By providing only the necessary detail in a document, or just updating the most relevant artifacts, the author may better focus her attention.

Incremental. By updating only the required information for that point in time, one may take an incremental approach to the production and evolution of contents.

Concentrated effort. The effort of applying this pattern 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.

Using DOMAIN-STRUCTURED INFORMATION supports TIME-SHIFTED CO-EVOLUTION, since making richer information available allows to more easily track which information needs to be co-evolved.

All of the *Information Proximity* patterns help this pattern in a similar way, since having related contents easily reachable from one another allows to determine which contents are affected by a particular change.

This pattern depends on the ability to track how the information has evolved, which may be achieve through an AUDITABLE DOCUMENT.

SYNCHRONOUS CO-EVOLUTION is a direct alternative to this pattern, and the choice on which to use will greatly dependent upon how easy it is to decide which contents are related with each other and how easy it is to track, at at later time, which changes were introduced.

8.3 Known Uses

Solutions that allow AUDITABLE DOCUMENTS to be produced support TIME-SHIFTED CO-EVOLUTION. Wiki engines and version control systems are a good examples of such solutions, which track how a documents evolve and support the decision of what changes are required to maintain consistency.

9. AUDITABLE DOCUMENTS

Documents are produced and consumed by different actors, who work cooperatively (i.e. towards the same goal) but there isn't a global view of the contributions made by each of them. Namely, it's not possible to know when each contribution was introduced and by whom.

9.1 Problem

How to increase the transparency of the process by which the documents are evolved?

Being able to follow and understand how a document is created makes the authoring process more *transparent*. Such transparency is proportional to the level of *traceability* achieved, this is, to the level of detail recorded about every step in a process chain.

Tracking the evolution of an *heterogeneous* document as a whole may not be easy, depending on the types of artifacts that are used. Furthermore, the introduced tracking mechanisms may increase the *complexity* of authoring the document, and the extra information that is recorded may increase its *storage space consumption*.

9.2 Solution

Make it possible to assess at any time who, how, and what has been produced, by tracking information regarding its creation and evolution process.

All these extraneous information (meta-information) directly related to the process, may be recorded. Namely, information about which individual contributions were made to a given document along it's evolution, who has done each of them and at which point in time they were made, among others.

The following consequences should be considered when applying this pattern:

Transparency. By recording these meta-information, and by making it accessible, authors and readers may understand how the document is being evolved. This allows authors to perceive how they may play a part in that process, and increase readers' trust in the document, by tracking document changes. It becomes possible for them to assess which updates were made to the documentation and if inconsistencies might have been introduced.

Tool support. New tools may have to be developed that are able to efficiently track different types of artifacts.

Space Consumption. Additional storage space is required to accommodate the additional contextual information and each of the applied changes. If space consumption becomes critical, either: (a) use a strategy where only enough information about the change is stored (i.e. keeping only the differences), or (b) discard older data, though taking into account the potential loss of transparency and traceability.

Using Domain-Structured Information makes richer information available, thus providing more feedback on which information needs to be co-evolved.

Time-Shifted Co-Evolution uses this pattern to support determining the information that may need to be updated.

Other related patterns include Document History [14], which focuses on maintaining a list of past versions of a document, and Annotated Changes [14], that provides a way to directly record inside a document which of its parts have recently been changed.

9.3 Known Uses

It is common for text processors to possess 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.

Document management systems frequently track the entire production context of documents, and the several versions that a document has gone through. Wiki engines also track changes made to their pages, and allow to later query the differences between different versions.

Version Control Systems (VCS), specifically those that use global versioning (such as *Subversion*, as opposed to *CVS*), are widely used to maintain consistency during the process of evolution in software development.

10. HETEROGENEOUS DOCUMENT

Several types of documentation artifacts are produced, each fulfilling a distinct purpose. During this process, different facets of the overall information need to be addressed and combined into the same document.

10.1 Problem

How to express different types of information in the same document?

The *simplest* way of authoring and combining different types of information is to express them all as the same type of artifact — documentation is commonly for the most part expressed as text — although different subjects may be better conveyed using *different types of artifacts*.

There are *mismatches* between different types of artifacts. Using different formats, and having been designed as standalone units, and requiring different authoring tools, they are most often not easy to combine.

10.2 Solution

Allow the coexistence of different types of artifacts in the same document while maintaining them as separate entities.

The underlying format of the document should be able to deal with different types of artifacts relating distinct pieces of information, and weaving them to result in an heterogeneous document, without losing the identify of each artifact it is composed by. Information Proximity patterns may be used as a general approach to combine these interrelated parts of information while preserving overall consistency. Since information recorded as different types of artifacts pose additional challenges, mainly due to their different syntactic format and potential lack of a formal semantics, they may not trivially support being combined. A specification of how these different artifacts should be weaved allows them to remain self-contained, despite being part of a larger document. The process of weaving these artifacts together may be done automatically by supporting tools.

The following consequences should be considered when applying this pattern:

Fitness for purpose. By creating an heterogeneous document (i.e. combining different artifacts) authors may decrease the effort in preserving consistency among related artifacts, since they are closely presented. Authors are also able to address the particular goal of each document, since different artifacts convey complementary information, and allow to better express the intended ideas.

Mismatches. Artifacts are weaved together through a weaving specification, working around the difficulty of bridging them using their own (often syntactically irreconcilable) formats alone.

Heterogeneous Tools. The use of distinct tools increases the effort of authoring content.

Simplicity. Authoring simplicity may be lost, as the author now need to support the multiple ways of interacting with each of the several supported content types.

This pattern may be applied by making use of VIEWS or TRANSCLUSION when the use of different types of artifacts is possible. It greatly benefits from an INTEGRATED ENVIRONMENT since different types of authoring tools are usually involved.

10.3 Known Uses

XSDoc [7, 2] is a wiki engine oriented for software development in which pages weave together heterogeneous artifacts.

Several office software suites, such as *Microsoft Office* and *OpenOffice*, allow combining different kinds of artifact in a same document. This is a feature that unfortunately has not yet seen it's usage in popular Integrated Environments, such as Visual Studio or Eclipse. However, some uses of Literate Programming, such as *VDMTools*, directly parse and write *.rtf* documents which have native support for images.

11. DOMAIN-STRUCTURED INFORMATION

While textual descriptions represent a very flexible way of capturing acquired knowledge, and thus are usually an important part of software documentation, the degree in which a document is relevant will vary depending on how well it serves its purpose and accurately conveys the intended ideas. Moreover, the same piece of information bay be better conveyed by using different perspectives, intrinsically related with each other.

Maintaining and assuring consistency requires continuous review. The major cause of it is the fact that relations between documentation parts aren't explicitly formalized, hence decreasing the capability of information to be automatically identified, processed and inferred. For this reason, these types of maintenance tasks affect the issue of consistency orthogonally.

11.1 Problem

How to create documentation so that procedures over the captured information may be automated?

As mentioned before, textual documentation is a *flexible* way of capturing knowledge. This flexibility is an important asset, but a higher level of *formalization* also brings benefits, such as being less subject to multiple interpretations, and allowing information to be automatically processed.

However, the mechanisms used to allow a degree of formalization higher than that provided by textual descriptions, may easily undermine the *simplicity* of producing documentation

11.2 Solution

Structure information according to its domain, so that the information form directly relates to domain concepts.

Textual documentation usually follows a text-oriented structure, using elements such as titles, paragraphs, lists, tables, etc. In order to automate tasks, infer relationships and preserve format consistency over it, an higher degree of structure is needed, requiring a formalization oriented to domain concepts and their underlying relations, specifically between different artifacts that convey different perspectives over the same information.

Extending the documentation with richer structure provides an infrastructure where not only consistency can become automatically assessed, but also prevent introduction of new inconsistencies.

The following consequences should be considered when applying this pattern:

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

Objectivity. The use of a structure with well defined semantics makes information less open to different interpretations.

The creation of structure and/or individual information fragments, frequently required by *Information Proximity Patterns*, tends to converge to DOMAIN-STRUCTURED INFORMATION. This pattern also supports *Evolution*, as it provides a richer base of trackable information.

As with the remaining patterns of this pattern language, DOMAIN-STRUCTURED INFORMATION requires appropriate tool support, and may benefit from the use of an INTEGRATED ENVIRONMENT.

This pattern is similar to STRUCTURED INFORMATION [14], 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 objective of automating consistency assessment, while STRUCTURED INFORMATION focuses mainly in structuring contents to ease readers' perception.

11.3 Known Uses

Code comments are a form of source code documentation. Code annotations, such as Javadoc comments [12], add an additional level of structure to source code comments, formalizing information elements of 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 [11].

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 [15, 10].

12. 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 with each other, making the development environment heterogeneous and more difficult to use.

12.1 Problem

How to maintain consistency between related information parts that are stored as independent artifacts?

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*.

12.2 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. They support the production and maintenance of several types of artifact, providing specialized features for each of them and an infrastructure through which they interoperate.

This supports strategies of documentation maintenance which focuses 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 complexifying each tool individually, their overall simplicity is increased by providing an homogeneous usage.

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

INTEGRATED ENVIRONMENT directly contributes the remaining patterns of this pattern language, by orchestrating the several tools involved. It is also directly related to the pattern FEW TOOLS [14], which introduces the notion that supporting the creation of documentation with too many and unconnected tools may become a burden, rather than a way to support the users.

12.3 Known Uses

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

13. ACKNOWLEDGMENTS

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

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

[9] 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.

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

[11] 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.

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

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

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

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

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