Patterns for Documenting Frameworks – Part III

Authors **Ademar Aguiar, Gabriel David**
FEUP & INESC Porto, Universidade do Porto
E-mail: ademar.aguiar@fe.up.pt, gtd@fe.up.pt
version: 25 September 2006

Good design and implementation are necessary but not sufficient pre-requisites for the successful reuse of object-oriented frameworks. Although not always recognized, good documentation is crucial for effective framework reuse and comes with many issues. Defining and writing good quality documentation for a framework is often hard, costly, and tiresome, especially when not aware of its key problems and the best ways to address them. This document presents patterns from a set of related patterns that describe proven solutions to recurrent problems of documenting object-oriented frameworks. The pattern language they all form together aims at helping non-experts on cost-effectively documenting object-oriented frameworks. The patterns here presented complements the subset of artefact patterns under development (two presented at VikingPLoP'2005, and more two at EuroPLoP'2006) and address the problems of *describing the customizable points of the framework* and *how such customization is supported*, respectively the patterns "CUSTOMIZABLE POINTS" and "DESIGN INTERNALS".

Introduction

Object-oriented frameworks are a powerful technique for large-scale reuse capable of delivering high levels of design and code reuse. As software systems evolve in complexity, object-oriented frameworks are increasingly becoming more important in many kinds of applications, new domains, and different contexts: industry, academia, and single organizations.

Although frameworks promise higher development productivity, shorter time-to-market, and higher quality, these benefits are only gained over time and require up-front investments. Before being able to use a framework successfully, users usually need to spend a lot of effort on understanding its underlying architecture and design principles, and on learning how to customize it, which all together implies a steep learning curve that can be significantly reduced with good documentation and training material.

This paper contributes with two patterns to the work in progress of writing a pattern language that focus on problems of documenting frameworks [1][2][3], some of the several technical, organizational, and managerial issues that must be well managed in order to employ frameworks effectively.

Pattern language

The pattern language comprises a set of interdependent patterns that aim at helping developers on becoming aware of the typical problems they will face when documenting object-oriented frameworks. The patterns were mined from existing literature, lessons learned, and expertise on documenting frameworks, based on a previous compilation about framework documentation [4].

The pattern language describes a path commonly followed when documenting a framework, not necessarily from start to end to achieve effective results. In fact, many frameworks are not documented as completely as suggested by the patterns, due to different kinds of usage (white-box or black-box) and different balancing of tradeoffs between cost, quality, detail, and complexity. One of the goals of these patterns is precisely to expose such tradeoffs in each pattern, and to provide practical guidelines on how to balance them to find the best combination of documents to the specific context at hands.

According to the nature of the problems addressed, the patterns are organized in *artefact patterns* (*which kinds of documents to produce? what should they include? how to relate them?*), to which belong the patterns here documented, and *process patterns* strictly related with the process of cost-effectively documenting frameworks (*how to do it? which activities, roles and tools are needed?*), which are included as an appendix.

Artefact patterns

Artefact patterns address problems related with the documentation itself, here seen as an autonomous and tangible product independent of the process used to create it. They provide guidance on choosing the kinds of documents to produce, how to relate them, and what to include there.

Similarly to other technical documentation, the overall quality of framework documentation is complex to determine and assess, and this is perhaps the first issue. Documentation must have quality, that is, it must be easy to find, easy to understand, and easy to use [6]. Task-orientation, organization, accuracy, and visual effectiveness are among all documentation quality attributes, the most difficult ones to achieve on framework documentation [4].

From the reader's point of view, the most important issues are on providing accurate task-oriented information, well-organized, understandable, and easy to retrieve with search and query facilities. From the writer's point of view, the key issues are on selecting the contents to include, on choosing the best representation for the contents, and on organizing the contents adequately, so that the documentation results of good quality, while easy to produce and maintain.

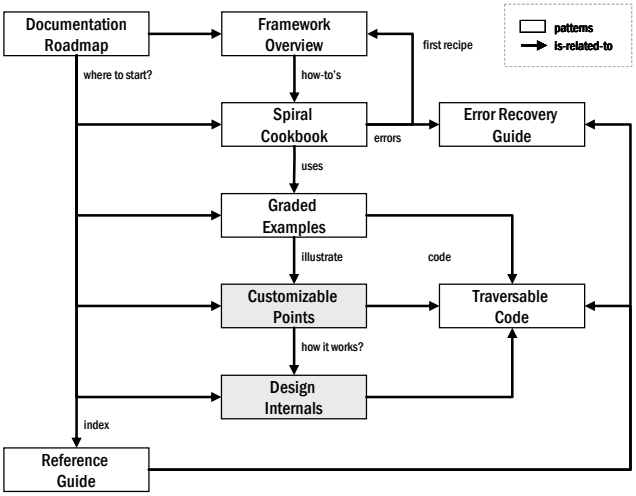

Figure 1 - Documentation artefact patterns and their relationships.

Patterns overview

To describe the patterns, we have adopted the Christopher Alexander's pattern form: *Name-Context-Problem-Solution-Example* [7]. Before going to the detail of each pattern, we will overview the pattern language with a brief summary of each pattern's intent. For contextual purposes, all the artefact patterns are overviewed below and depicted in Figure 1 highlighting the two patterns described in this paper.

Documentation Roadmap helps on deciding what to include in a first global view of the documentation that can provide readers of different audiences with useful and effective hints on what to read to acquire the knowledge they are looking for [1].

Framework Overview suggests providing introductory information, in the form of a framework overview, briefly describing the domain, the scope of the framework, and the flexibility offered, because contextual information about the framework is the first kind of information that a framework user looks [1].

Cookbook & Recipes describes how to provide readers with information that explains how-to-use the framework to solve specific problems of application development, and how to combine this prescriptive information with small amounts of descriptive information to help users on minimally understanding what they are doing [2].

Graded Examples describes how to provide and organize example applications constructed with the framework and how to cross-reference them with the other kinds of artefacts (cookbooks, patterns, and source code) [2].

Customizable Points describes how to provide readers with task-oriented information with more precision and design detail than cookbooks and recipes, so that readers can quickly identify the points of the framework (hot-spots) they need to customize and get a quick understanding about how they are supported (hooks).

Design Internals explains how to provide detailed design information about what can be adapted and how the adaptation is supported, by referring the patterns that are used in its implementation and where they are instantiated.

Reference Guide suggests what to include as reference information and how to structure the documentation to make it the most complete and detailed as possible to assist advanced users when looking for descriptive information about the artefacts and constructs of the framework.

Traversable Code provides hints on how to organize and present source code, both of the examples and the framework itself, when desired, to make it easy to browse and navigate, from, and to, other software artefacts included in the overall documentation, namely models and documents.

Error Recovery Guide explains how to help users on understanding and solving the errors they encountered when using the framework.

Pattern **Customizable Points**

You are documenting a framework to provide application developers with prescriptive and descriptive information capable of helping them on customizing the framework the way they need to satisfy the requirements they have at hands.

Problem To help application developers being effective on customizing a framework, the documentation should be organized in a way that can help readers on quickly obtaining detailed information, both prescriptive and descriptive, about the framework parts strictly required to customize, and how to customize them, in order to implement the specific features of the application at hands.

Although examples, cookbooks and recipes are good at providing prescriptive information, they might not be sufficient to allow customization of specific parts or in specific situations not predicted in other forms of documentation.

How to help readers on knowing which framework parts are customizable?

How to help readers on learning in detail how to customize a specific part of a framework?

Forces **Task-orientation.** Readers want to learn in detail how to use a certain customizable part of the framework, so the documentation must focus on customization tasks imposed by the framework, which users really need to perform, as perceived in the recipes of the framework's cookbook.

Balancing Prescriptive and Descriptive information. To be effective, the documentation about how to customize a specific part of a framework must achieve a good balance between the level of detail of the instructions provided to guide the usage of that framework's part, and the level of detail and focus used to communicate how it works, i.e. its design internals.

Different Audiences. An application developer is a software engineer who is responsible for customizing a framework to produce the application at hands. In a first place, application developers want to identify which customizations are needed to produce the desired application, and to know how to implement them, instead of understanding why it must be done that way. The application developer thus needs prescriptive information capable of guiding her on finding out which hot spots must be used, which set of classes to subclass, which methods to override, and which objects to interconnect. It must be expected that the application developer possibly is not knowledgeable on the application domain and not an experienced software developer.

Completeness. Readers appreciate complete information, i.e. that all possible customizations are mentioned with all the possible detail, which is not always feasible as it largely depends on the reader's point of view and the tasks to support.

Easy-to-use. Independently of the level of completeness and detail, the resulting documentation must be easy to use.

Solution Provide a list of the framework's *customizable points*, also known as *hot-spots*, i.e., the points of predefined refinement where framework customization is supported, and, for each one, describe in detail the *hooks* it provides and the *hot-spot subsystem* that implements its flexibility.

To allow easy retrieval, provide lists of customizable points ideally organized by different criteria, being probably the following the most important ones:

- *by kind of framework functionality*, to provide a black-box reuse-oriented view; especially useful when looking for possibilities of customization related with a set of features in mind;
- *by framework parts and modules*, to provide a white-box reuse-oriented view; especially useful when looking for possibilities of customization related with a specific framework part or module.

Hot-spot. Customization is supported at points of predefined refinement, called *hot-spots*, using general techniques, such as, abstract classes, polymorphism and dynamic binding. A hot spot usually aggregates several hooks within it and is implemented by a hot-spot subsystem that contains base classes, concrete derived classes and possibly additional classes and relationships.

Hook. Hooks present knowledge about the usage of the framework and provide an alternative view to design documentation [5]. Hooks provide solutions to very well-defined problems. They detail how and where a design can be changed: what is required, the constraints to follow, and effects that the hook will impose, such as configuration constraints.

A hook description usually consists of a name, the problem the hook is intended to solve, the type of adaptation used, the parts of the framework affected by the hook, other hooks required to use this hook, the participants in the hook, constraints, and comments. Hooks can be organized by hot spot; as said before, a hot spot tends to have several hooks within it. The usage of hooks can be semi-automated with the help of wizards, for example.

Hot-spot subsystem. The hot-spot subsystem supports variability either by inheritance or by composition. The variability is often achieved by the dynamic binding of a template method *t()*, an operation from a class *T*, that calls a hook method *h()*, an abstract operation from a base class, via a polymorphic reference typed with the class of the hook pointing to an operation *h'()*, from a subclass of *H,* that overrides *h()*. With inheritance, the polymorphic reference is attached to the hot-spot subsystem; with composition the reference is contained in it. Figure 2 below shows an example of both kinds of hot-spot subsystems.

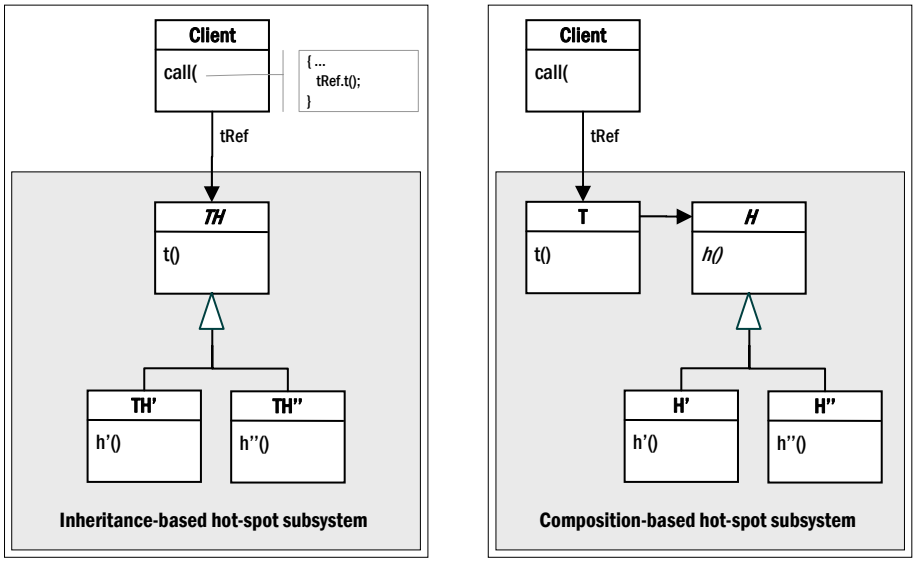

Figure 2 - Inheritance-based and composition-based hot-spot subsystems.

Examples Despite providing an organized list of customization points being of great value in terms of documentation completeness, they are not so frequently used as examples, cookbooks and recipes in the documentation of the most popular frameworks, namely those we have been referring so far in these patterns. We discuss below how these customizations are documented in some well-known frameworks.

JUnit. The major kind of reuse that JUnit was designed for is very simple and consists only on writing and organizing tests, so its documentation is mostly targeted to explain how to do these tasks, which is simply and perfectly documented as cookbooks and recipes in the document "JUnit Cookbook" document [14].

However, some more customizations can be done with JUnit, such as test runners, and test decorators, but information about these and other less used customizable points is only briefly mentioned in the "JUnit FAQ" document [15] and in the low-level Javadoc documentation. Figure 3 shows an enumeration of other possible customizations of JUnit (version 3.8.2) described in its accompanying documentation. How such customizations are implemented, i.e. their hot-spot subsystems, are not documented and only identifiable by direct source code inspection.

Swing. When compared with JUnit, Swing is a very large framework providing a huge number of possible customization points, which are organized in its documentation in a simple and easy to browse manner that uses different levels of depth and detail. The most intuitive list is probably the one provided by the "Visual Index to the Swing Components" (see Figure 4). A good and more complete alternative to the visual index to learn what can be customized in the Swing framework is the list that enumerates how-to use each of the key components (Figure 5-left), which gives access to more detailed lists of possible customizations of each component (Figure 5-right). Even more detailed information about how the flexibility is supported in each customization point

although is not explicit in the documentation, but left to the reader to explore by herself probably using the Javadoc API and source code inspection.

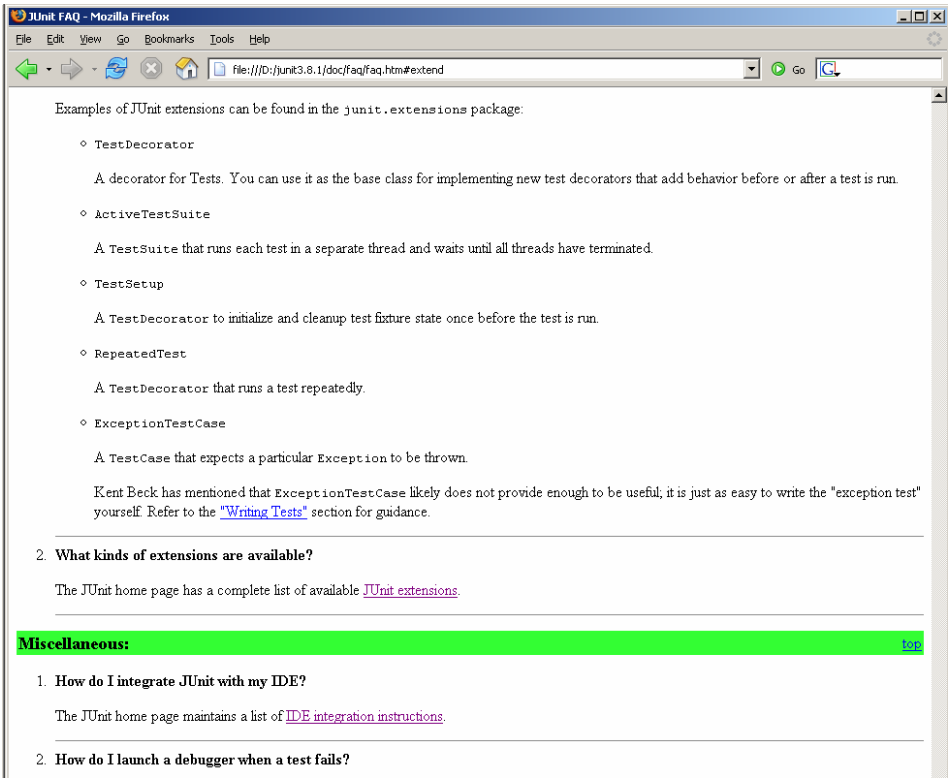

Figure 3 JUnit: hot-spots are only implicitly mentioned in the FAQ.

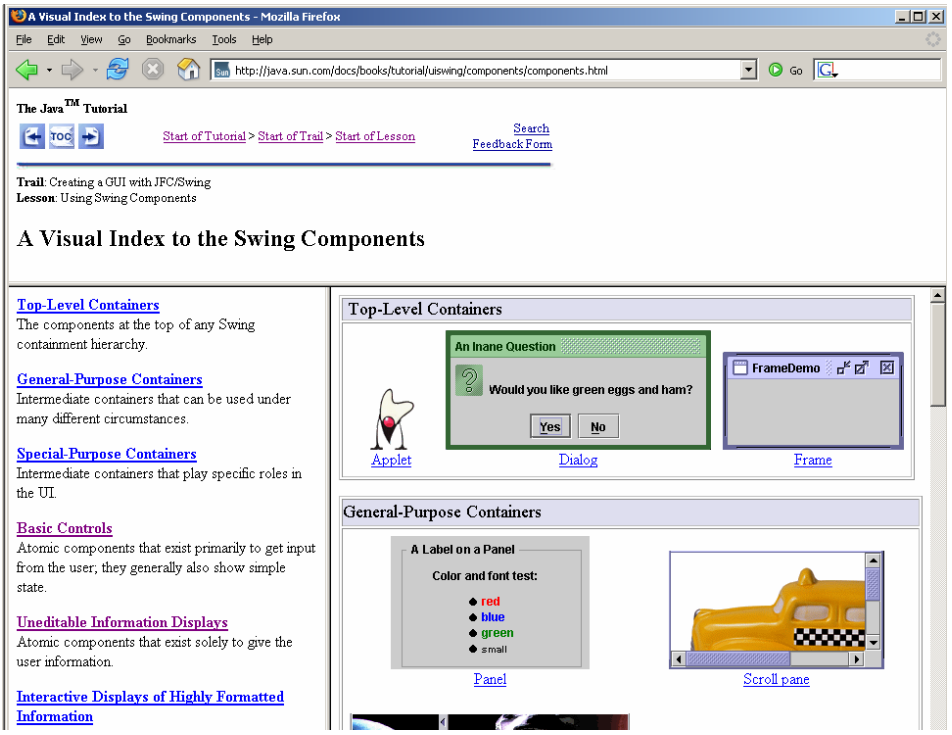

Figure 4 "A Visual Index to the Swing Components."

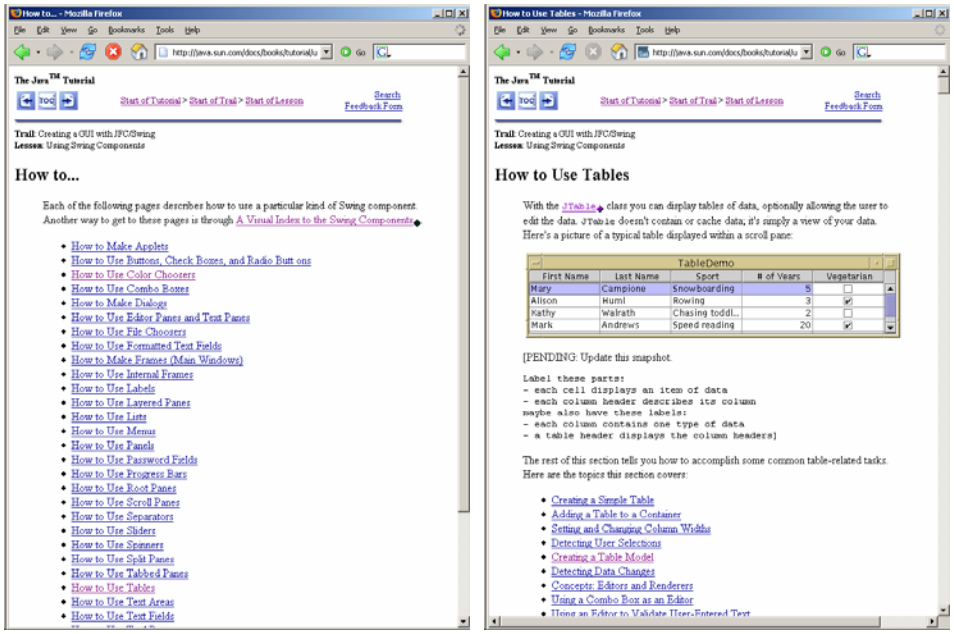

Figure 5 List of the most frequently used customizations possible with Swing and Swing Tables.

Consequences

By providing framework users with an organized and exhaustive list of all the predefined customization points, or at least, the most important and frequently used, readers can evaluate faster if the framework is applicable to the problems at hands, and therefore to decide with more confidence to reuse it or not.

After knowing the points to customize, whether the knowledge was gathered from own experience, others knowledge, or documentation (e.g. "CUSTOMIZATION POINTS", "GRADED EXAMPLES", or "COOKBOOKS AND RECIPES"), framework users can then start learning which tasks must be carried on to customize them properly, possibly supported by the prescriptive information provided by the "COOKBOOKS AND RECIPES" related with those customizable points. In addition, they can use the descriptive information provided for each "CUSTOMIZATION POINT" to learn more about how its flexibility is supported, and the information about its "DESIGN INTERNALS" to know in detail how the framework is designed.

Although adding some possible redundancy, lists of "CUSTOMIZATION POINTS" are easy to use and browse and provides a good balance between prescriptive and descriptive information thus being a good complement to the prescriptive information of "COOKBOOKS AND RECIPES" and the descriptive information of "DESIGN INTERNALS".

Pattern **Design Internals**

Information explaining in detail how a framework was designed and implemented can be of great value for potential users willing to get a better understanding in order to reuse it in more advanced ways.

Problem Framework instantiation for a particular application often consists on customizing hot spots in a way planned by framework designers. Typical instantiations can be often achieved simply by plugging in concrete classes selected from an existing library that customize the hot spots to the needs of the application at hands, also known as black-box reuse. Other instantiations can be achieved by extending framework abstract classes in a way planned by framework designers. The instantiation requires matching of interfaces and behaviors, and the writing of code to implement new behaviors, also known as white-box reuse.

Not all instantiations of a framework are simple to achieve, but they can't be all documented exhaustively and in enough detail, especially those more advanced customizations, or those not initially planned by framework developers.

To cover these advanced instantiations, but also other advanced kinds of reuse, such as flexing, composing, evolving or mining a framework, it is thus important to provide framework users with detailed information about how a framework and its flexibility was designed and implemented so that they can figure out which and how to develop the customizations.

How to help framework users on quickly grasping the design and implementation of a framework to support them on achieving customizations not typical, advanced, or not specifically documented?

Forces **Different Purposes.** In addition to the framework purpose and usage instructions, the framework documentation must also provide information to help framework users on understanding the underlying principles and the basic architecture of the framework so that they can develop not only trivial and planned but also advanced applications that are conformant to the framework.

Balancing Prescriptive and Descriptive information. Although programmers can use a framework without completely understanding how it works, such as when following a set of instructions, a framework is much more useful for those who understand it in detail. To be effective, the documentation must achieve a perfect balance between the level of detail of the instructions provided to guide the usage of the framework, and the level of detail and focus used to communicate how the framework works, i.e. its design internals.

Minimizing design information complexity. To communicate complex software designs is challenging. Frameworks derive their flexibility and reusability from the use (and abuse) of interfaces and abstract classes, which, together with polymorphic methods, significantly complicate the understanding of the run-time architecture. The design information to communicate can include not only the different classes of the framework, but also the strategic roles and collaborations of their instances,

and rules and constraints, such as cardinality of framework objects, creation and destruction of static and dynamic framework objects, instantiation order, synchronization and performance issues.

Solution Provide concise but detailed information about the design internals of the framework by describing the framework hot-spots at a meta-level using *meta-patterns*, and by describing the roles of framework participants using *design patterns* and *design pattern instantiations*.

Design patterns. A pattern names, abstracts, and identifies the key aspects of a design structure commonly used to solve a recurrent problem. Succinctly, a *pattern* is a generic *solution* to a recurring *problem* in a given *context* [7]. The description of a pattern explains the problem and its context, suggests a generic solution, and discusses the consequences of adopting that solution. The solution describes the objects and classes that participate in the design, their responsibilities and collaborations. The concepts of pattern and pattern language were introduced in the software community by the influence of the Christopher Alexander's work, an architect who wrote extensively on patterns found in the architecture of houses, buildings and communities [7]. Patterns help to abstract the design process and to reduce the complexity of software because patterns specify abstractions at a higher level than single classes and objects. This higher-level is usually referred as the *pattern level*.

A design pattern is thus a specialization of the pattern concept for the domain of software design. Design patterns capture expert solutions to recurring design problems. As design patterns provide an abstraction above the level of classes and objects, they are suggested as a natural way for documenting frameworks [10]: to describe the purpose of the framework, the rationale behind design decisions, and to teach them to their potential users.

Design patterns are particularly good to document frameworks because they capture design experience at the micro-architecture level and enclose meta-knowledge about how to incorporate flexibility [16][21]. In fact, design patterns are capable to illuminate and motivate architectures, preserve design decisions made by original designers and communicate to future users, and provide a common vocabulary that improves design communication, and to help on the understanding of the dynamics of control flow.

The concepts of frameworks and patterns are closely related, but neither subordinate to the other. Frameworks are usually composed of many design patterns, but are much more complex than a single design pattern. In relation to design patterns, a framework is sometimes defined as an implementation of a collection of design patterns.

To document the design internals of a framework in relation with the patterns it implements we must first know, or recognize, the patterns in the framework design, and to match them against the many popular design patterns already documented, such as the catalogues known as GoF patterns [16] and POSA patterns [18]. However, more contextualized design patterns are very likely to not being yet published or documented, due to its specificity, either in terms of

applicability or organization dependency. In these situations, it is required to spend the effort to mine and write the patterns considered important to explain the underlying framework design. A good source of knowledge for those willing to learn how to write patterns is [19], itself documented under the form of a pattern language.

Design pattern instances. Searching, selecting and applying design patterns are the necessary steps of the cognitive process for assigning the roles defined in a pattern, to concrete classes, responsibilities, methods and attributes of the concrete design. This process is generally called pattern instantiation [22].

Documenting pattern instances is important because it will help other developers on better understanding the resulting concrete classes, attributes and methods, and the underneath design decisions. This provides a level of abstraction higher than the class level, highlighting the commonalities of the system and thus promoting the understandability, conciseness and consistency of the documentation. At the same time, the documentation of pattern instances will help the designer instantiating a pattern, to certify that she is taking the right decision. In general, this results in better communication within the development team and consequently on less bugs.

To more formally document a pattern instance we must describe the design context, justify the selection of the pattern, explain how the pattern's roles, operations and associations are mapped to the concrete design classes, and to state the benefits and liabilities of instantiating the pattern, eventually in comparison with other alternatives.

Meta-patterns. Frameworks are designed to provide their flexibility at hot spots using two essential constructs: templates and hooks. The possible ways of composing template and hook classes in the hot spots of a framework were catalogued and presented under the form of a set of design patterns, which were called meta-patterns. Although meta-patterns can be used to document the roles of framework participants, the level of detail is too fine to be useful, but extremely useful to document the roles of the participants involved in a design pattern.

Examples Frameworks are usually composed of many design patterns, being sometimes defined as an implementation of a collection of design patterns. Design patterns are thus commonly used in many frameworks to explain the global architecture of the framework, and how it was designed. We will illustrate here with examples of how design patterns are used to document popular frameworks, such as JUnit, Swing, J2EE and .NET, and also the classical HotDraw framework.

HotDraw. The first paper that mentions the advantages of using patterns to document a framework is authored by Ralph Johnson [10], which presents a pattern language to document the HotDraw framework, comprising a set of patterns, one for each recurrent problem of using the framework. In that work, patterns are not only used to document the design of the framework, but also as a way of organizing the documentation, similarly as a cookbook does with the recipes (pattern "COOKBOOK AND RECIPES"), where each pattern provides a format for each recipe.

JUnit. The document "A Cook's Tour" [28], devoted to explain how JUnit was designed, includes a pattern-by-pattern tour to the design internals of JUnit. Figure 6 presents an extract from this document that shows the design patterns used in the architecture of JUnit, which describe in more detail JUnit's internal design. In concrete, it informally enumerates the design patterns instantiated by the major abstractions of JUnit.

Figure 7 presents on the left another extract from this document informally explaining, using natural language, models, and fragments of source code, how the class `TestCase` instantiates the Template Method design pattern. Figure 7 on the right presents an extract from the documentation relative to the Template Method pattern [17] that shows the structure of the solution proposed by the pattern, the participants involved and their roles, and the consequences of instantiating the pattern.

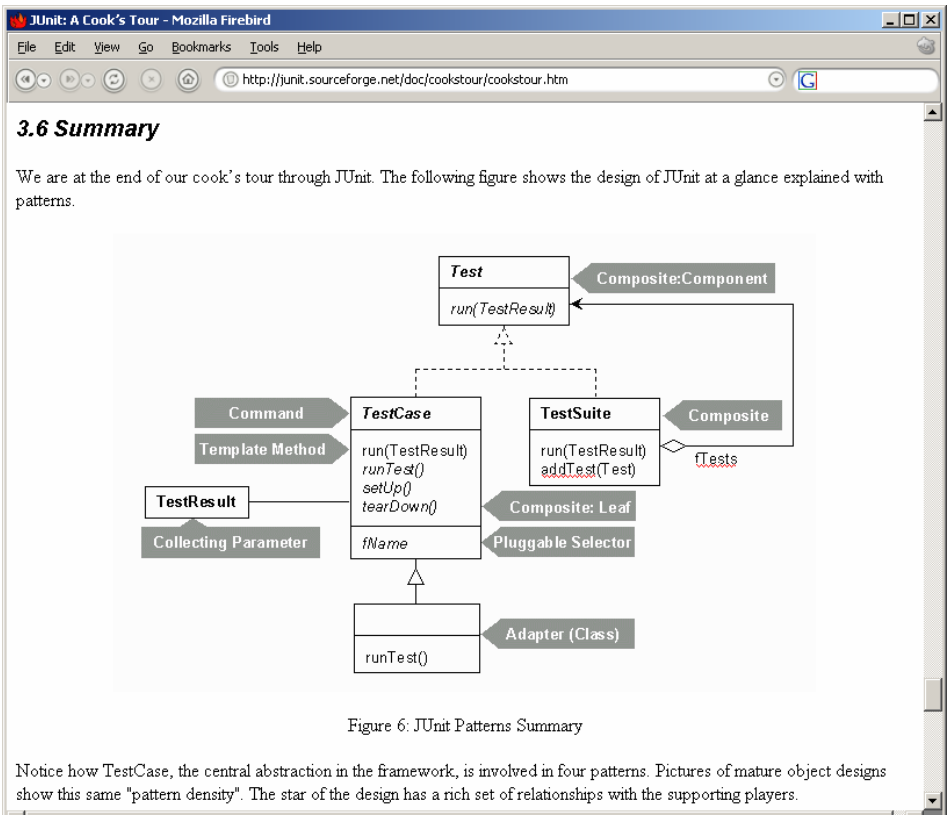

Figure 6 Example of using design patterns to document the design of JUnit.

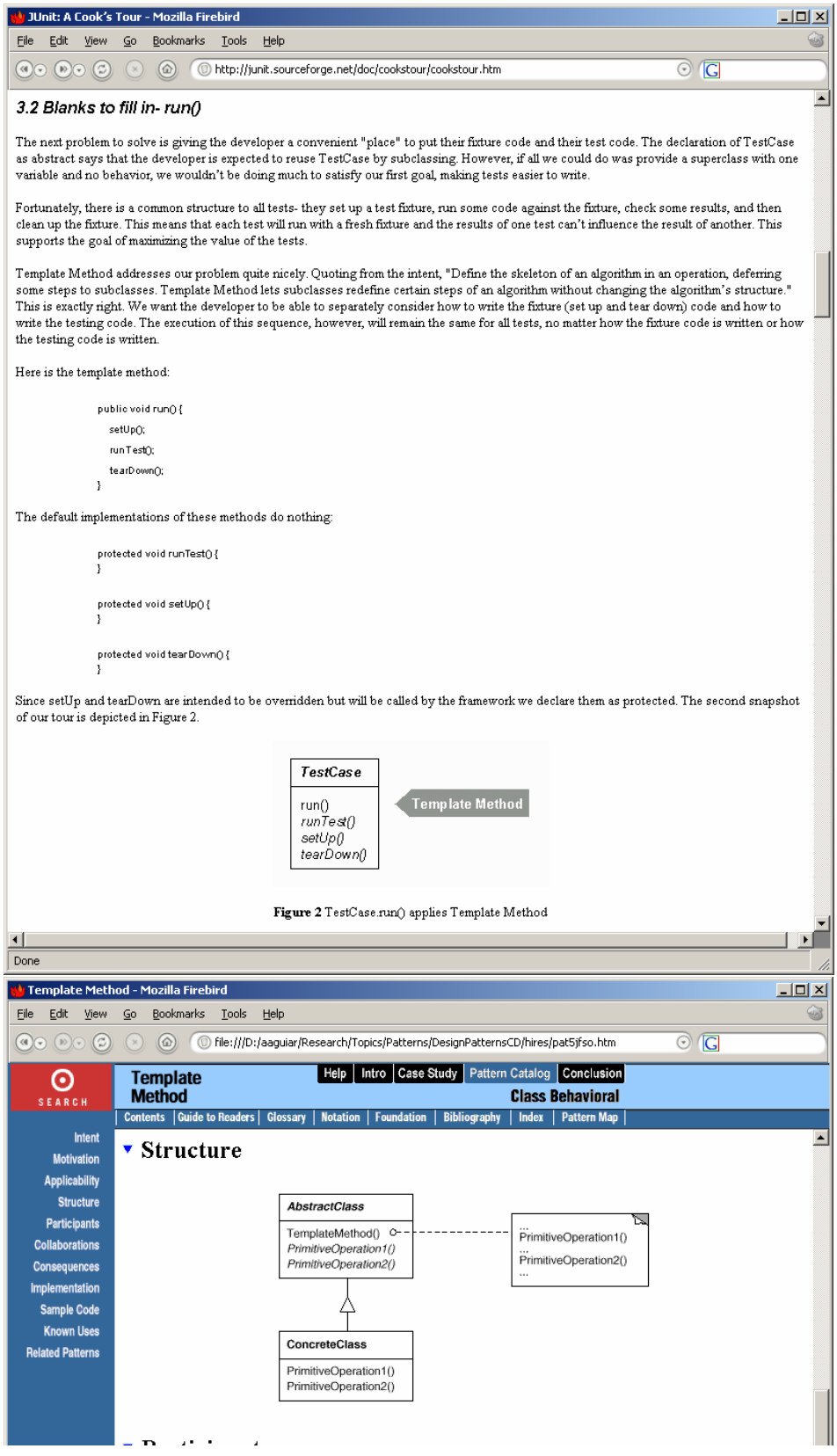

Figure 7 Template Method: being instantiated by the TestCase class (left); Template Method Pattern (right).

Known Uses

Swing. The much more complex Swing framework instantiates many more patterns (e.g. Observer, Composite, Decorator, Visitor, etc.) but its accompanying documentation doesn't use pattern instances as explicitly and exhaustively as we can observe in JUnit, probably due to the cost of doing it.

Figure 8 shows an extract from an overview of the Swing architecture, where we can learn about the foundational design principles of Swing, concretely the model-view-controller architectural pattern (MVC) and its instantiation in Swing classes.

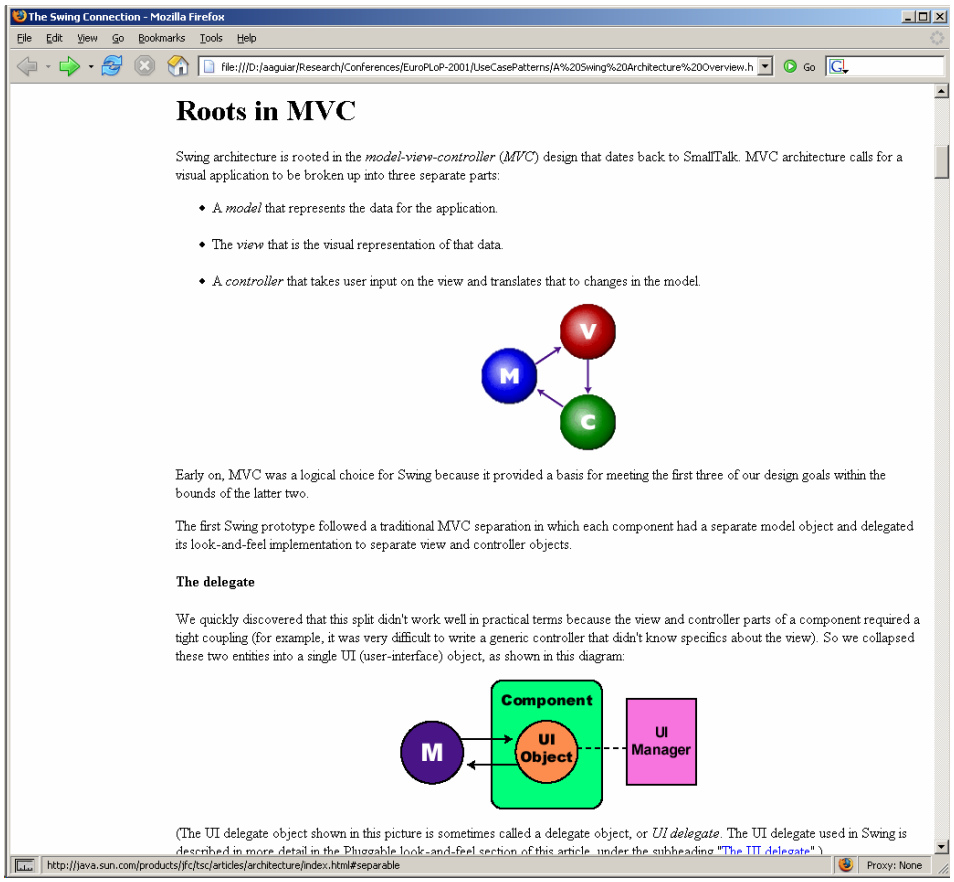

Figure 8 An extract from "A Swing architecture overview" showing MVC and its instantiation in Swing.

J2EE. The patterns underlying the design of the enterprise version of Java is documented in the core J2EE patterns catalog [20], which serve as a valuable source of knowledge to learn more about how J2EE is designed and how the applications based on J2EE should be designed. Figure 9 shows the index of all the core J2EE patterns.

.NET. Similarly to J2EE, there is a document that presents the patterns underlying Microsoft's .NET framework for enterprise applications. Figure 10shows the documentation of the MVC pattern, which includes an example of its instantiation in .NET.

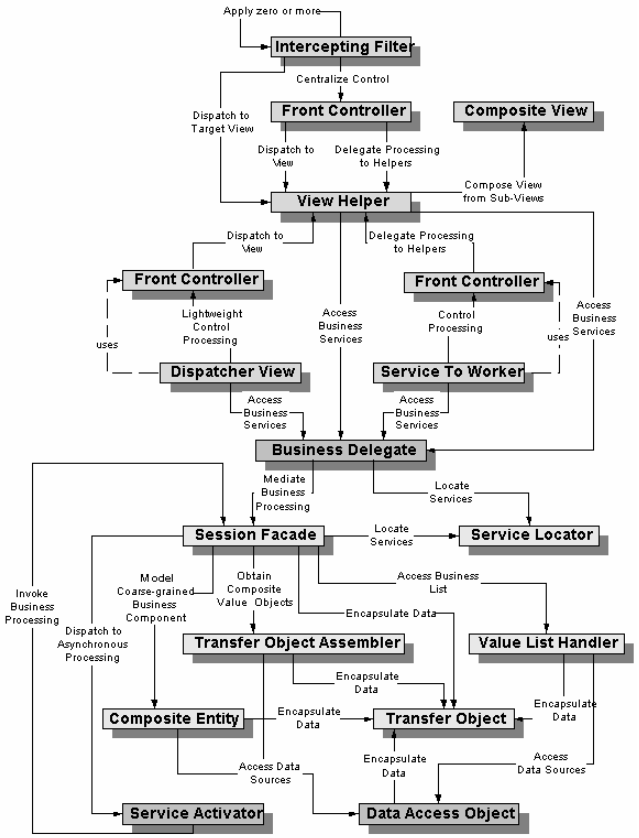

Figure 9 Core J2EE Patterns: patterns index.

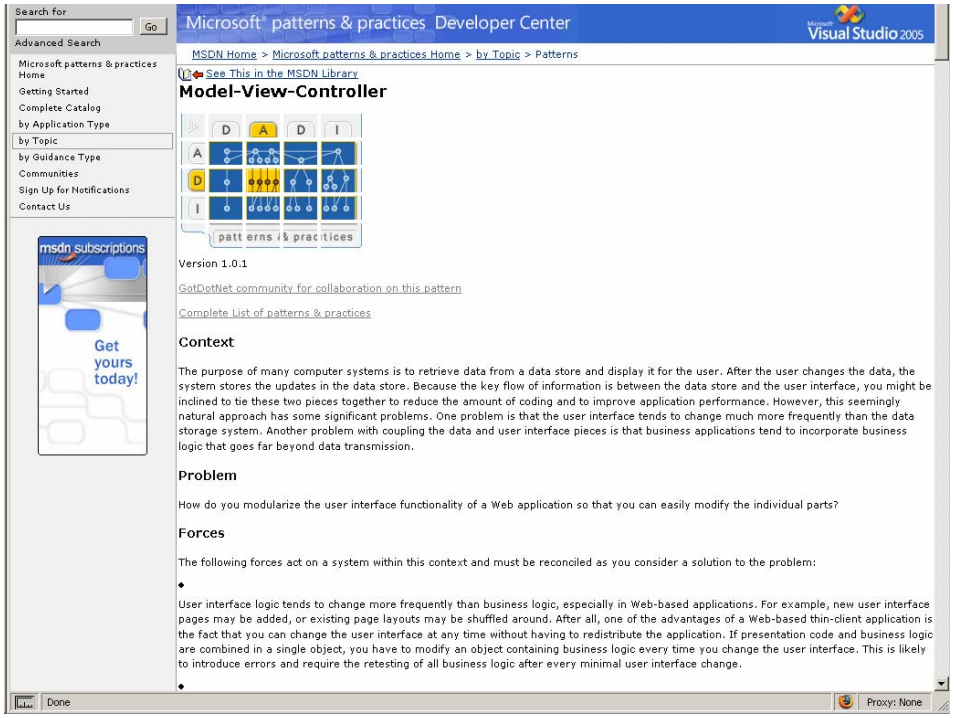

Figure 10 .NET enterprise solution patterns". showing MVC and its instantiation in .NET.

Consequences

By documenting the framework design internals, using patterns and pattern instances, namely, we provide framework users with additional knowledge that can help them better understand the underlying architecture and design principles of the framework, and therefore to enable more advanced customizations or simple but not documented customizations elsewhere in another form of documentation.

However, to document framework's specific patterns, not published, and to document pattern instances can be hard work, if not done at the right moment by the right people.

As one of the most complex kinds of object-oriented software systems, frameworks can be hard to understand and explain, but definitely patterns are a excellent mean to do that, as they provide a good balancing between simplicity of reading and richness of the information provided.

Credits

The authors would like to thank our shepherd Rosana Teresinha Vaccare Braga, and also Ralph Johnson, for the valuable comments and feedback provided during the shepherding of these patterns. We thank also Neil Harrison, Uwe Zdun, for shepherding previous patterns from this same set of patterns, and Eduardo Fernandez, Kevlin Henney, Klaus Marquardt, Sergiy Alpaev, Sami Lehtonen, Allan Kelly, Ian Graham, Alexander Füllebornand, Martin Schmettow, Michalis Hadjisimouand, and all the other participants of the writer's workshops at VikingPLoP'2005 and EuroPLoP'2006, for the motivation, comments and suggestions for improvement they provided.

References

[1] Aguiar, A., and David, G. (2005). Patterns for Documenting Frameworks – Part I. In Proceedings of VikingPLoP'2005, Helsinki, Finland (to be published).

[2] Aguiar, A., and David, G. (2005). Patterns for Documenting Frameworks – Part II. In Proceedings of EuroPLoP'2006, Irsee, Germany (workshopped).

[3] FEUP, doc-it project web site, http://doc-it.fe.up.pt/.

[4] Aguiar, A. (2003). A minimalist approach to framework documentation. PhD thesis, Faculdade de Engenharia da Universidade do Porto.

[5] Froehlich, G., Hoover, H. J., Liu, L., and Sorenson, P. G. (1997). Hooking into object-oriented application frameworks. In International Conference on Software Engineering, pages 491–501.

[6] Hargis, G. (2004). Developing quality technical information. Prentice-Hall, 2nd edition.

[7] Alexander, C., Ishikawa, S., and Silverstein, M. (1977). A Pattern Language. Oxford University Press.

[8] Krasner, G. E. and Pope, S. T. (1988). A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. Journal of Object-Oriented Programming, 1(3):27–49.

[9] Pree, W. (1995). Design Patterns for Object-Oriented Software Development. Addison-Wesley / ACM Press.

[10] Johnson, R. (1992). Documenting frameworks using patterns. In Paepcke, A., editor, OOPSLA'92 Conference Proceedings, pages 63–76. ACM Press.

[11] Lajoie, R. and Keller, R. K. (1995). Design and reuse in object-oriented frameworks: Patterns, contracts and motifs in concert, pages 295–312. World Scientific Publishing, Singapore. World Scientific.

[12] Froehlich, G., Hoover, H. J., Liu, L., and Sorenson, P. G. (1997). Hooking into object-oriented application frameworks. In International Conference on Software Engineering, pages 491–501.

[13] Apple Computer (1986). MacApp Programmer's Guide. Apple Computer.

[14] Beck, K. and Gamma, E. (2003b). JUnit: Cookbook. Available from http://junit.sourceforge.net/doc/cookbook/cookbook.htm.

[15] Clark, M. (2003). JUnit: FAQ - frequently asked questions. Available from http://junit.sourceforge.net/doc/faq/faq.htm.

[16] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995b). Design Patterns — Elements of reusable object-oriented software. Addison-Wesley.

[17] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995a). Design Patterns — Elements of reusable object-oriented software. Addison-Wesley, CD version edition.

[18] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). Pattern Oriented Software Architecture — a System of Patterns. John Wiley & Sons.

[19] Meszaros, G., and Doble, J. (1996). Metapatterns: A pattern language for pattern writing. In the 3rd Pattern Languages of Programming conference, Monticello, Illinois, September 1996.

[20] Alur D., Crupi, J., and Malks, D. (2001). Core J2EE Patterns: Best Practices and Design Strategies, Publisher: Prentice Hall / Sun Microsystems Press, ISBN:0130648841; 1st edition.

[21] Beck, K. and Johnson, R. (1994). Patterns generate architectures, volume 821, pages 139–149. Springer-Verlag. Berlin.

[22] Odenthal, G. and Quibeldey-Cirkel, K. (1997). Using patterns for design and documentation. In Akcsit, M. and Matsuoka, S., editors, ECOOP'97 — Object-Oriented Programming, 11th European Conference Proceedings, volume 1241 of Lecture Notes in Computer Science, pages 511–529. Springer-Verlag.

[23] Eckstein, R., Loy, M., and Wood, D. (1998). Java Swing. O'Reilly & Associates, Inc.

[24] Weinand, A., Gamma, E., and Marty, R. (1989). Design and implementation of ET++, a seamless object-oriented application framework. Structured Programming, 10(2).

[25] Gosling, J., Joy, B., and Steele, Jr., G. L. (1996). The Java Language Specification. Addison-Wesley. Also available online at URL http://java.sun.com/docs/books/jls/.

[26] Beck, K. and Gamma, E. (2003c). JUnit: Test infected: Programmers love writing tests. Available from http://junit.sourceforge.net/doc/testinfected/testing.htm.

[27] Schappert, A., Sommerlad, P., and Pree, W. (1995). Automated support for software development with frameworks. In ACM SIGSOFT Symposium on Software Reusability, pages 123–127.

[28] Beck, K. and Gamma, E. (2003a). JUnit: A cook's tour. Available from http://junit.sourceforge.net/doc/cookstour/cookstour.htm.

[29] Beck, K. and Gamma, E. (1997). JUnit homepage. Available from http://www.junit.org.

[30] Hansen, T. (1997). Development of successful object-oriented frameworks. In Addendum to the 1997 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (Addendum), pages 115–119. ACM Press.

Appendix

This appendix briefly presents the *process patterns* that complement the *artefact patterns* previously referred. They address problems and solutions strictly related with the process of cost-effectively documenting frameworks (*how to do it? which activities, roles and tools are needed?*).

Process Patterns

The patterns related with the process of cost-effectively documenting object-oriented frameworks are overviewed below and depicted in Figure 11.

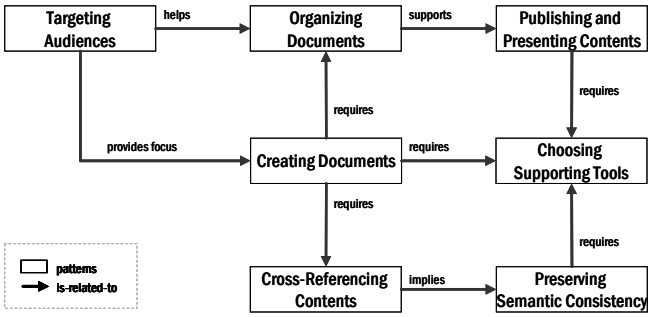

Figure 11 - Documentation process patterns and their relationships.

Targeting Audiences describes one of the first activities in the overall process of documenting a framework, which is to define and prioritize the audiences intended to be addressed by the documentation. Having defined the audiences on target, the contents can be properly created and organized so that they can be presented through the most appropriate views and formats for those audiences.

Creating Documents provides hints on the main activity of documentation. It explains how to streamline the creation of documentation artefacts (documents, models, source code fragments, etc.) both by developers and technical writers, to yield a good quality and cost-effective documentation.

Cross-Referencing Contents addresses the problem of linking and relating different documentation artefacts (e.g. examples, patterns, source code), to provide good navigability between all the contents involved, and therefore to minimize the obstacles to learning strategies that readers spontaneously adopt.

Preserving Semantic Consistency suggests ways of coping with the difficulties of preserving the semantic consistency between related software artefacts (source code, models, and documents) during development to enable their continual review and modification throughout the lifecycle and thus to preserve its accuracy and value for the readers.

Organizing Documents provides hints on how to keep all the contents consistent, well structured, integrated, easy to browse, and easy to maintain.

Publishing and Presenting Contents describes the ultimate activity of documentation, the reason why it is produced and organized. The pattern addresses issues on using documentation, not only to read contents in a presentation format, but also to browse, search, select, and navigate through the contents, what sometimes requires processing of contents (transformations, filtering, composition, etc.), to present them in a format convenient for the user.

Choosing Tool Support addresses the problem of ensuring quality and reducing the typically high costs associated with the production and maintenance of framework documentation. The pattern suggests automating the documentation process the best as possible, while retaining the flexibility and adaptability to different developers and environments.