# A Model for Structuring User Documentation of Object-Oriented Frameworks Using Patterns and Hypertext

Matthias Meusel, Krzysztof Czarnecki, and Wolfgang Köpf
Daimler-Benz AG
Research and Technology
Ulm, Germany
E-mail: m.meusel@str.daimler-benz.com
{czarnecki, koepf}@dbag.ulm.DaimlerBenz.com

**Abstract.** Adequate documentation of an object-oriented framework is the prerequisite to its success as a reusable component. The overall design of a framework and its intended method of reuse are not obvious from the source code and thus have to be addressed in the documentation. Most importantly, the documentation of a framework has to be structured in such a way that it guarantees the adequate support of three major audiences: users selecting a framework, users learning to develop typical applications based on the selected framework, and users intending to modify its architecture.
This paper presents a model for structuring the documentation of an object-oriented framework. The model integrates existing approaches such as patterns, hypertext, program-understanding tools, and formal approaches into a single structure that is geared towards supporting the three audiences. The model will be illustrated using HotDraw, a Smalltalk framework for drawing editors, as an example. We also give a preliminary evaluation of the model.

## 1 Introduction

Object-oriented frameworks are reusable designs that leverage simple code reuse. One of the advantages of frameworks is the high degree of adaptability that is achievable through specialization. Unfortunately, experience has shown that both the development of a framework and its specialization are difficult. Specialization of a framework is difficult since it usually requires the modification of the framework (e.g. through subclassing), and this kind of reuse presupposes detailed knowledge of its design. We refer to this as the *framework understanding problem*. On the other hand, frameworks represent a state-of-the-art technology which can be used to develop components for easy bottom-up integration (e.g. [Tra95]). Such components are more self-contained than framework classes and avoid framework usability problems—yet often at the cost of lower adaptability.
The overall design of a framework and its intended method of reuse are not obvious from the source code. For this reason, adequate framework documentation is a necessary prerequisite to successful framework reuse. The biggest challenge, however, to documenting a framework is that it must support users with varying backgrounds and different levels of experience in using the framework. We observed this problem with the documentation of in-house frameworks developed at Daimler-Benz. The need for adequate documentation is even more urgent if the frameworks are intended to be reused as standard components throughout the corporation. In this context, a framework has to be viewed as a product.

This paper presents a model for documenting object-oriented frameworks that was developed to address the needs of adequate documentation of standard frameworks within the corporation. The model integrates existing approaches such as patterns [Joh92], hypertext, program understanding tools, and formal approaches into a single structure which is geared towards supporting different types of users. We also give a preliminary evaluation based on a case study that used this model to document HotDraw [Bra95], a moderately-sized Smalltalk framework.
This paper is organized as follows: Section 2 defines the requirements for user documentation of object-oriented frameworks. Section 3 introduces "the pyramid principle", a text structuring approach which our model is based on. The documentation model is presented in Section 4. Section 5 lists the related work, and Section 6 reports on some preliminary evaluation of the model and indicates directions for future research.

## 2 Requirements for User Documentation of Object-Oriented Frameworks

The general problem of *program understanding* is defined as the process of developing a mental model of a software system's intended architecture, meaning, and behavior [Mül96]. Methods to support program understanding include *documentation* as well as *program understanding* and *reverse engineering tools* such as static and dynamic analyzers, browsing and visualization tools that extract knowledge directly from the actual system.
*User documentation* is the type of documentation that is packaged with the final product. Some information for user documentation is extracted from the *development documentation* which is maintained throughout the project. But as a rule, user documentation has a different focus than development documentation. Development documentation records all the information pertinent to the developed product and its development process. User documentation, on the other hand, should contain only the information that enables the user to make the best use of the final product. Therefore, user documentation requires special effort to assure its value to the end-user.
According to [SAN88] (see also [AS90]), documentation adequacy is constituted by its *accuracy, completeness, usability*, and *expandability*. In the context of structuring user documentation, we are chiefly interested in usability. This can be further divided into *logical traceability* and *understandability*:

- **Logical traceability** is defined as the ability to follow the logical train of thought in the documentation through all of the pertinent parts, regardless of whether the parts are contiguous or not [SAN88]. A user must be able to find everything he needs, making adequate *search facilities* and *references* an absolute necessity.
- **Understandability** (also known as *Comprehensibility* or *Logical Readability*) is defined as the ease with which the documentation can be comprehended [SAN88]. User documentation is written for people. For this reason, the available knowledge of the process by which people construct mental models of the world has to be taken into account when structuring documentation. This knowledge includes issues such as the way people organize objects and the typical capability of short-term memory. In terms of text structuring, factors influencing understandability include *modularity*, *conciseness*, and *redundancy appropriateness* (i.e. when to repeat an item of information and when to use a reference). Appropriate writing

style, term consistency, and physical readability (i.e. format, print, etc.) are also important but are not considered in this paper.

The adequacy of documentation has to be evaluated in the context of its user and his goals. In particular, user documentation has to be structured so that it is useful to users with different levels of experience in using the software. Johnson differentiates among three types of audiences which the documentation of a framework needs to address [Joh94]:

1. **users deciding which framework to use:** These users should be able to quickly asses the scope of the framework and to tell if the framework is adequate for the application to be built. No information on how to use the framework is needed. Quick and accurate evaluation can prevent unnecessary costs.
2. **users wanting to build a typical application:** This audience needs to learn how to use the framework in order to develop a standard application. This involves customizing the appropriate "hot spots" [Pre95] by configuring concrete classes, filling in contracts by implementing abstract methods, etc. The amount of reading that is required for the user to be able to build simple applications has to be minimized.
3. **users wanting to go beyond the typical use:** These users intend to add new components and features, modify the architecture, etc. They need the detailed descriptions of the framework design and the algorithms. This type of documentation has to explicitly include the design assumptions underlying the framework architecture. This type of user needs to be able to find very specific information quickly.

## 3 Pyramid Principle

As already stated, the way people build mental models of the world needs to be taken into account when structuring documentation. In fact, this requirement applies to any document whose purpose is to communicate something useful to the reader. An ex-

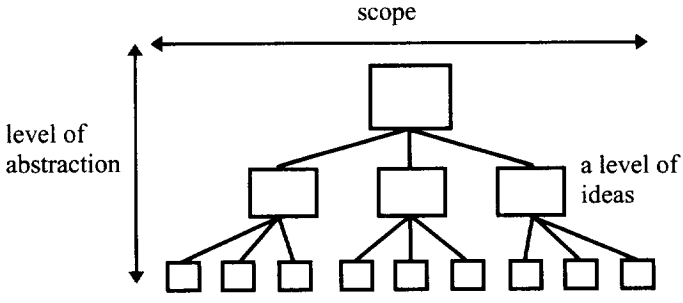



**Fig. 1.** Pyramid structure of a document

ample of a document-structuring approach that follows this requirement is the *pyramid principle* [Min91]. This approach has been developed primarily for structuring business documents. It is based on the concept that any grouping of ideas is easier to comprehend if this grouping is pre-sorted into a logical, top-down structure. This structure reassembles a pyramid with a main idea at the top and the remaining ideas

arranged into levels (see Figure 1). Each idea in the pyramid raises questions that are then answered at the lower level. The relationships between statements across two levels are called vertical relationships, whereas the relationships within one level are horizontal relationships. The latter can be further divided into deductive or inductive relationships.

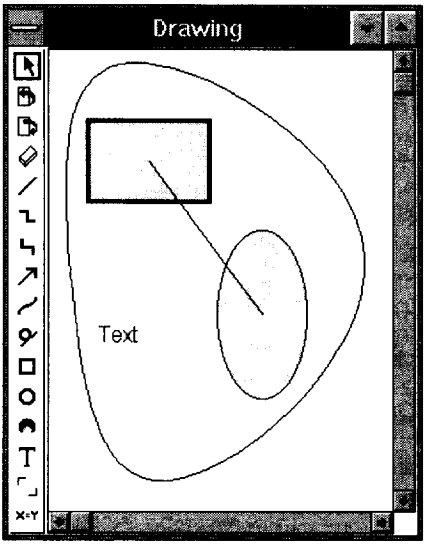



**Fig. 2.** Sample HotDraw application

The pyramid shape corresponds to the bottom-up abstraction process, i.e., to the process of clustering ideas to come up with more general ideas, which is typical for any development and discovery process. However, the presentation of ideas should always proceed in a top-down fashion since it focuses the reader's attention. Presenting an idea will create questions in his mind that have to be answered at the lower level. This process should be similar to carrying on a conversation.
The shape of the pyramid is determined by "The Magical Number Seven Plus or Minus Two". This famous number, introduced by Miller, indicates the typical number of things that a person can store in his or her short-term memory [Mil67]. If more than seven ideas at one level support an idea at the higher level, then it will most probably become necessary to insert a new level grouping the supporting ideas into logical classes between the other two levels.
The concept of abstraction represents the way people think and lies at the heart of hierarchical software [Par72]. Documentation needs to be structured in the same modular and hierarchical way.

## 4 Model of User Documentation for Object-Oriented Frameworks

Our model of framework documentation is based on the pyramid principle. In fact, every documentation has this structure whereby the structuring units are usually

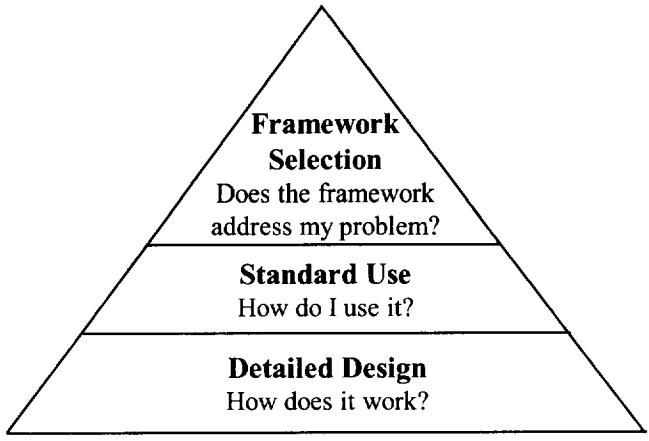



**Fig. 3.** Framework Documentation Pyramid

books, chapters, sections, paragraphs, etc. In our model, however, the ideas within the documentation pyramid are represented, as far as possible, in the form of patterns [GHJV95]. The pyramid is implemented using hypertext to capture the references between patterns.

In his seminal paper [Joh92], Johnson used patterns to document HotDraw. HotDraw is a Smalltalk framework for developing structured drawing editors, such as the one shown in Figure 2 [Bra95]. HotDraw is a moderately sized framework. The implementation of HotDraw described in [Bra95] has about 90 classes. We decided to use HotDraw as a case study for our documentation model, as it is an extremely useful framework, and it is not difficult to find users who would like to use HotDraw. Our hope was to receive feedback on the documentation from a variety of users.

In our model, the documentation pyramid has three main levels that correspond to the three different audiences defined in Section 2. At the top of the pyramid, there is the *Framework Selection Level* that addresses the scope of the framework using a *catalog pattern* (see Figure 3). The transition from top level to the *Standard Usage Level* is initiated by the question: *How do I use the framework?* The Standard Usage Level uses a system of *application patterns* to answer this question. Finally, the question *How does the framework work?* is addressed at the *Detailed Design Level.*

We will now discuss the three documentation levels using the HotDraw documentation as an example.

### 4.1 Framework Selection: Catalog Pattern

The *catalog pattern* is a concise, structured description indicating the framework's name, type, its application domain, main features, scope (illustrated by citing examples of typical applications), the main ideas behind its design, and related frameworks. The catalog pattern, as any pattern, has a problem-solution format and can be used as an entry in a component repository. In fact, its structure reassembles the structure of so-called design records [Tra95] that are used to describe components in a repository. Figure 4 shows the structure of the catalog pattern for the HotDraw example. Our hypertext implementation uses HTML frames to present the structure of a

pattern and the selected section at the same time (as in Figure 8). A more elaborate format for catalog patterns can be found in [SW96].

| Framework name | HotDraw |
|---|---|
| Type | application framework, white-box with some black-box elements<br><*other possible entries are* support framework *or* framework toolkit> |
| Keywords | object-oriented semantic-based graphic editor, diagrams, drawings, Smalltalk, white-box framework |
| Problem description | <*short description of the application domain*> |
| Solution | <*main features and design concepts of the framework*> |
| Examples | net editors, diagram editors, drawing editors, two-dimensional object visualization <*list of typical applications*> |
| Documentation | <*short description of the parts of documentation, possible and suggested entry points for reading*> |
| Other frameworks | SmallDraw, UniDraw for C++ |

**Fig. 4.** Catalog pattern for HotDraw

When documenting a framework toolkit, i.e., a set of related frameworks, it is necessary to create at least two levels of catalog patterns. For example, the ACE toolkit [Sch96] is a framework toolkit consisting of a number of support frameworks for distributed systems. The Framework Selection Level for ACE would consist of one main ACE catalog pattern and a number of other catalog patterns, each describing one of the ACE frameworks. As a general rule, depending on the complexity of the software which is documented, any of the three main pyramid levels can contain multiple sublevels.

### 4.2 Standard Usage: Application Patterns

Patterns have been proposed as a documentation technique which minimizes the amount of reading in order to solve a problem [Joh92]. The pattern approach to documentation corresponds to a cookbook and cookbooks have been successfully used to document e.g. MacApp [App] or VisualWorks [PPD]. We refer to patterns for documenting frameworks as *application patterns* as opposed to e.g. design patterns [GHJV95], since they describe how a framework is used to develop a typical application and not how the framework was developed. Application patterns are basically cookbook recipes with a certain standard format, and co-relationships. A system of application patterns describes the *standard application design space* of a framework. This design space can be thought of as determined by the framework's "hot spots" [Pre95].

There are many different pattern formats that usually depend on their area of application (e.g. design, architecture, system analysis, programming, planning and management, etc.; some example formats can be found in [CS95]). The format we used for the HotDraw application patterns is shown in Figure 5.

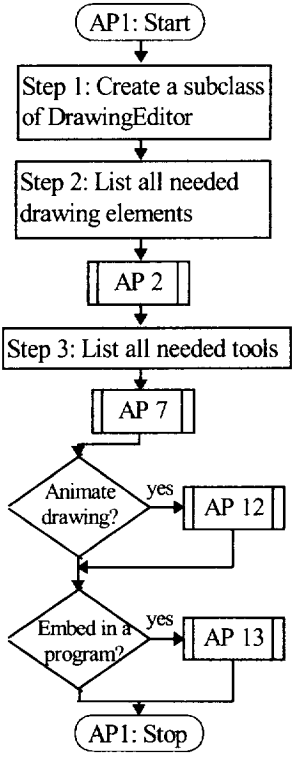
<table>
<tr><th colspan="2">Application Pattern 1: Creating a Semantic Graphic Editor</th></tr>
<tr><td>Context</td><td>HotDraw is a framework for structured, two-dimensional drawing editors. The elements of the drawing can have constraints between them, they can react to user commands, and they can be animated. The editors can be the complete application, or they can be embedded into a larger application.</td></tr>
<tr><td>Problem</td><td>How can a drawing editor be created using HotDraw?</td></tr>
<tr><td>Solution Flowchart</td><td>Explanations</td></tr>
<tr><td> 
</td><td>Step 1:<br>HotDraw provides DrawingEditor as a standard superclass for all drawing editors. You will need to create a subclass of DrawingEditor to represent your editor, e.g.:<br><pre>DrawingEditor subclass: #MyDrawingEditor
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'MyCategory'</pre>Step 2:<br>Each drawing consists of a number of drawing elements, such as lines, circles, and rectangles. You will need to create a list of drawing elements that are needed for your problem. Use Application Pattern 2 to define the needed drawing elements.<br>Step 3:<br>Tools are used to create and manipulate drawing elements. They are located on the tool palette. You will need to create a list of tools that are needed for your problem. Use Application Pattern 7 to define the needed tools.<br>Decision 1:<br>If you need to animate your drawing, refer to Application Pattern 12.<br>Decision 2:<br>If you need to embed your drawing in other program, refer to Application Pattern 13.</td></tr>
<tr><td>Examples<br>(references to concrete implementation steps)</td><td>PERTChart<br>DrawingInspector</td></tr>
<tr><td>Design Information</td><td>HotDraw Architecture</td></tr>
</table>

**Fig. 5.** Example of an application pattern

Each application pattern has a flowchart presenting the steps for solving the problem. Conditional decisions can be annotated with forces, i.e. factors that the framework

user has to consider when deciding which branch to follow. Some steps in the flowchart are local to the application pattern, e.g. Step 1 in Application Pattern 1 (Figure 5). The explanations of the local steps may include diverse diagrams, e.g. class or interaction diagrams. Other steps represent "calls" to other application patterns, e.g. AP2. The boxes representing such steps can be hyperlinked to the corresponding application patterns. The complete "calling structure" for the HotDraw application patterns is shown in Figure 6. The pattern names were left largely unchanged compared to the original names in [Joh92]. Six patterns, however, had to be added and the bodies of the original patterns had to be updated to adequately represent the new HotDraw design as described in [Bra95].

Application Pattern 1 (AP1) is the initial pattern (see Figure 5 and Figure 6). The

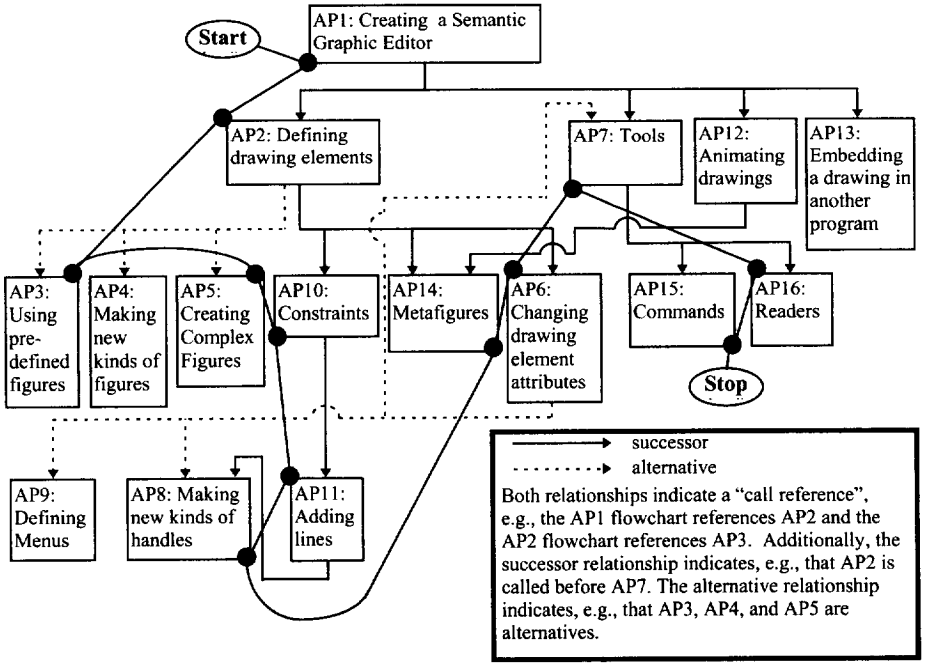



**Fig. 6.** System of application patterns for HotDraw and tutorial trail for PERTChart

original set of patterns in [Joh92] also had an initial pattern; however, we split it into the catalog pattern and the new initial pattern AP1, so that AP1 contains only the top level steps for creating a drawing editor.

Each application pattern references a concrete implementation step which is part of a tutorial covering the implementation of a complete application. A sample application for HotDraw is PERTChart. PERTChart is a simple editor for nets representing real time events and timing relationships between them (see [Bra95]). The PERTChart editor can be developed by applying twelve HotDraw patterns. The implementation steps for the PERTChart example are shown as a trail traversing the application pat-

terns in Figure 6. A framework documentation may require more then one tutorial, so that each application pattern references at least one concrete implementation step.
Ideally, the examples can be run out of the tutorial. A configuration management system, such as Envy [OTIa], can also be used to store each implementation step as a new version of the sample application. This way, one can browse through the source code of the application whose degree of completion corresponds to the currently viewed implementation step. We used a simple HTML server running in the Smalltalk environment in order to to invoke class browsers and other Smalltalk applications by clicking on a hyperlink in a HTML browser.
Tools for visualization of the examples, such as in [LN95], could be used at this level. In fact, so called *exemplars* have been proposed as a top-down approach to framework specialization [GM95]. In this approach, the application engineer starts with an exemplar that is similar to the intended application and visually customizes the framework by selecting classes for the hotspots.
If the documented framework contains large class categories for its hotspots, a class retrieval tool, such as ClassExpert [CHEK96], is a useful addition. ClassExpert deploys a similarity-based retrieval schema that supports reuse by both specialization and modification.

### 4.3 Detailed Design: Design Patterns and More

A system of application patterns describes the standard application configuration space of a framework. Frameworks, however, also offer reuse opportunities that go beyond the standard usage. This type of reuse usually involves some modification of the framework itself, such as modification of its architecture by including new kinds of components, altering some standard behavior, etc., and requires detailed knowledge of the framework design. The documentation of the detailed design corresponds to what is typically called *technical documentation* and is also essential for maintenance and re-engineering purposes.
A possible approach to documenting the design of a framework is the utilization of design patterns [GHJV95]. A set of design patterns for deriving HotDraw is presented in [BJ94]. The overall architecture of a framework can be documented using *architectural patterns* [BMR+96]. For example, the architecture of HotDraw is best described using the Model-View-Controller pattern [BMR+96, p. 125].
It might be difficult and time-consuming to present all the design information in the form of patterns. Therefore, some of the information could be presented simply as a set of related design documents that describe some selected topics (similar to the online help of Visual C++).
The design documentation has links to a class reference guide that describes the interfaces of framework classes. If class interfaces are properly commented, a complete class reference guide can be automatically generated from the source code. This approach is based on the idea of *literate programming* (see [Knu92]). In fact, commercial tools which generate HTML from Smalltalk code, for example, are available (e.g. Envy/QA [OTIb]), and some environments (e.g. Smalltalk/X [STX] ) generate such interface descriptions dynamically, thereby keeping the documentation synchronized with the environment.
As Johnson suggested, formal approaches are most appropriate at the detailed design documentation level. Examples of formal approaches for documenting frameworks include *contracts* [HHG90], *structural relationships* [PPSS95], and *ObjChart*

[GM95]. Such approaches could support the automatic verification that certain architectural assumptions are not violated by the client code. Another useful approach to combat this problem is *reuse contracts* [SLMD96].
Since the framework documentation can achieve only a limited degree of detail, program understanding and reverse engineering tools which extract design knowledge directly from the source code are extremely useful at this documentation level. Such tools include class and instance browsers, cross reference tools, debuggers, interaction diagrammers (e.g. [KM96]), method invocation coverage tools, path coverage tools, slicing tools (e.g. [TCFR96]), etc. These tools can also be hyperlinked with the documentation.

### 4.4 Overall Structure

The overall structure of the framework documentation is shown in Figure 7. The Framework Selection level contains a catalog pattern. The Standard Use Level consists of an application pattern system and tutorials (there are two tutorials in Figure 7). The Detailed Design Level contains a set of design patterns and design articles. The documentation should be supplemented with a glossary of domain concepts. Examples of domain concepts for HotDraw are *drawing elements* and *constraints*.
The list of contents on the initial page of the documentation should include the entry points indicated in Figure 7. These entry points are appropriate for a systematic top-down strategy to program understanding. To support other strategies, such as opportunistic strategy or iterative hypotheses refinement [TPS96], a search engine has to be provided. A search engine could also dynamically synthesize documentation pages that match certain user profiles in response to a query.

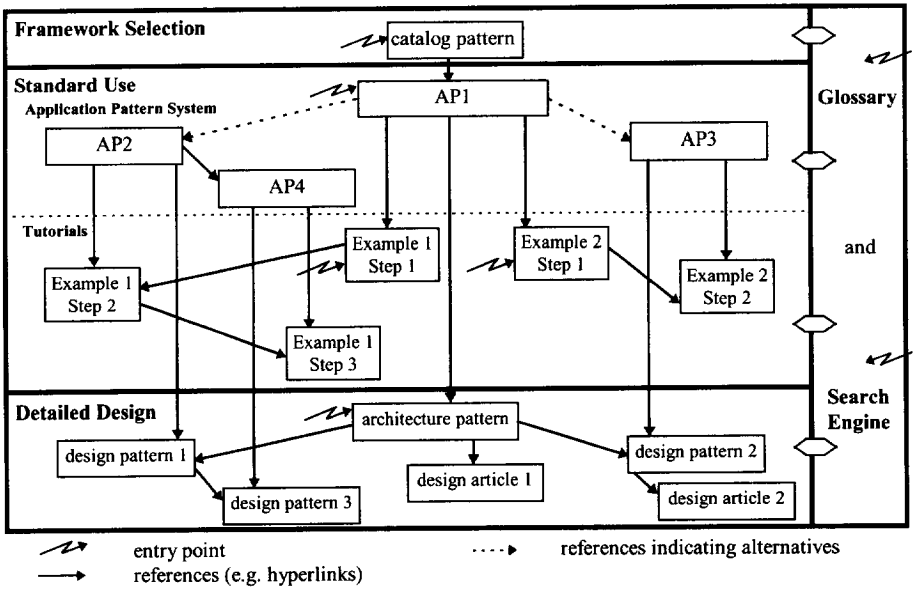



**Fig. 7.** Overall structure

In the foreseeable future there will be the need for a hard copy, therefore an appropriate facility for flattening of the hypertext for hard copy is required. The pyramid structure of the documentation guarantees its printability as a sequential document. Hyperlinks can be replaced by text references.
It is important to note that the completeness of the information in the lower parts of the pyramid can quickly lead to a "scope explosion". In order to adequately support the typical end-user, it is essential to adhere to the systematic, pyramid structure at the Framework Selection Level, Standard Usage Level, and in the upper part of the Detailed Design Level. This goal is achievable since the scope of the pyramid at these levels is still manageable. In large pyramids, the details in the lower parts of the Detailed Design Level are usually best accessed through a search engine.

## 5 Related Work

Johnson used patterns to document the standard use of HotDraw [Joh92], and Beck and Johnson used design patterns to document the design of HotDraw [BJ94]. Our work builds on both approaches and integrates them into a single model using hypertext. Our model distinguishes clearly between three documentation levels and defines the vocabulary for the documentation elements.
In [SW96], Silva and Werner propose patterns and hypermedia for packaging reusable components. The patterns described in their paper correspond to the catalog patterns in our model and are implemented using ToolBook, a hypermedia toolkit.
The types of documents that are usually packaged with object-oriented libraries or frameworks (such as VisualWorks) include the user's guide, cookbook, tutorials, technical documentation, and class reference guide. Our model contains all of these document types except for the user's guide, the contents of which are distributed over the three documentation levels.
The framework documentation techniques presented in [CI93] include documenting the class hierarchy, protocols, control flow, synchronization, entity relationships, and configuration constraints of the framework. These techniques can be classified into the Detailed Design Level of our model.
Probably the most ambitious approach to documenting frameworks are active cookbooks [PPSS95]. In the active cookbook approach, implementing steps are carried out automatically. This approach is complementary to our model. In fact, as indicated in the following section, active guidance and some bookkeeping functionality in the framework documentation system are desirable. However, the application of expert system technology to the documentation problem certainly needs more research. Our prototypical HTML implementation of the documentation model for HotDraw demonstrates the practicality of the model that is ready for pilot trials in industrial projects.
There has been a considerable amount of research on class retrieval techniques (see [CHEK96]). Such approaches clearly do not solve the framework understanding problem; yet, as indicated in Section 4.2, they are a useful addition to the documentation of a framework that contains very large class categories.

## 6 Discussion and Future Work

A complete evaluation of our documentation structuring model requires an elaborate empirical evaluation in an industrial environment that would enable a statistically valid conclusion to be drawn. Such an evaluation was beyond the scope of our project. Nevertheless, we have distributed our HotDraw documentation (in German) over the World Wide Web (see Figure 8).[1] A variety of users have viewed and used this documentation. Their subjective evaluation was basically positive even though there were some reports of problems.

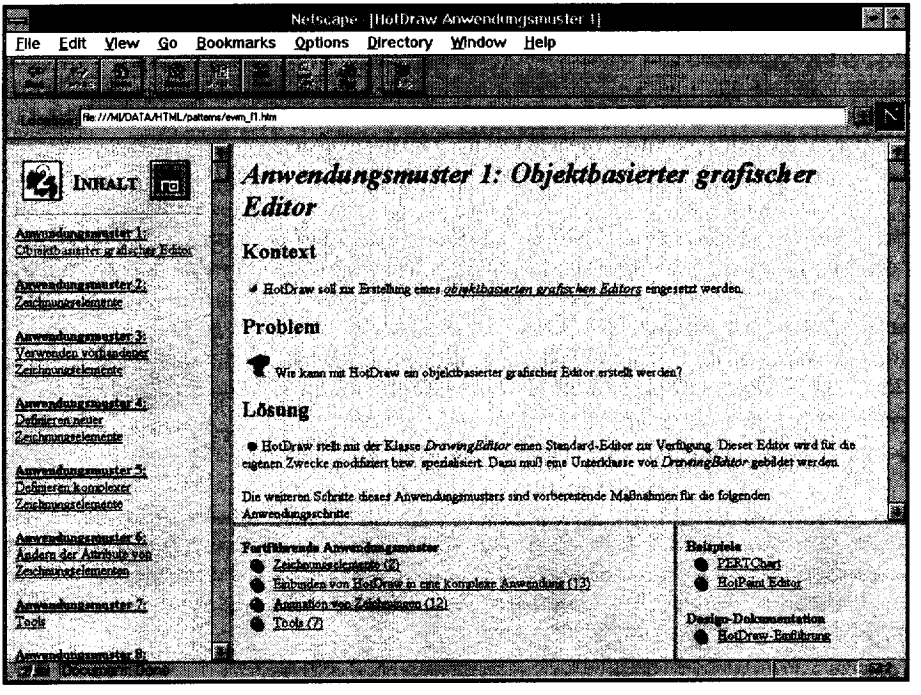



**Fig. 8.** Application Pattern 1 viewed in a HTML browser

Most of the users complained about the cognitive burden that results from keeping track of the path which was traversed through the system of application patterns while applying them. For example, after completing the step corresponding to the application pattern AP2 (in Figure 6), the user has to go back to the "calling" application pattern AP1 and resume at Step 3 (in Figure 5), which he has to remember. In order to solve this problem, a facility must be provided for keeping track of the executed steps. One possible scenario would be to let the user mark a step by clicking on an associated checkbox. The active cookbooks approach [PPSS95], however, seems to be a better solution.

[1] The HotDraw documentation is available in German at http://nero.prakinf.tu-ilmenau.de:80/~czarn/doku.html.

We also learned that it is important for hypertext documents to be largely self-contained in order to minimize the need for jumping back and forth in the "hyperspace", although the required degree of self-containment depends on the user's background. Research on *adaptive hypertext* (e.g. [Bru96]) might provide an adequate solution to this redundancy appropriateness problem in the long term.

An area that we did not cover is the problem of integrating our model into the industrial software development cycle. As one of the authors is currently involved in an industrial project to document a framework, we hope to derive the requirements for such integration in the near future.

Documentation affects the overall quality of a component and has a substantial impact on the component's success as a reusable component. The preparation of documents requires enormous effort which is often viewed as a burden, especially by the developer. This effort is frequently underestimated in a project. Some information for the user documentation (especially for the technical part) can be extracted from development documentation, although user documentation has a different focus than the development documentation. For this reason, our position is that user documentation has to be written by a documentation expert. At the heart of component-based development lies the principle that developing the components costs more than assembling them, so that the extra cost for documenting reusable components pays off in the reuse part of the software life cycle. Unfortunately, in many cases, we observed a lack of willingness to invest this extra amount of effort in documenting in-house frameworks. This situation manifests some cultural problems that are symptomatic of component-based development in large corporations.

## Acknowledgments

The authors would like to express their thanks to Ralph Johnson, John Brant, and others for making HotDraw publicly available over the Internet and thus providing an extremely useful tool as well as a case study for the research community. We also thank Frances Paulisch and Ulrich Eisenecker for providing us with valuable comments on an earlier version of this paper.

## References

[App] Apple Computer. *MacApp Programmer's Guide*. 1986

[AS90] J.D. Arthur and K.T. Stevens. Document Quality Indicators: A Framework for Assessing Document Adequacy. Technical Report, No. TR90-60, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, 1990

[BJ94] K. Beck and R. E. Johnson. Patterns Generate Architectures. In *Proceedings of the ECOOP'94*, Springer Verlag, Berlin, 1994, pages 139-149

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996

[Bra95] J.M. Brant. HotDraw. Master's Thesis, University of Illinois at Urbana-Champaign, 1995. HotDraw is available at http://st-www.cs.uiuc.edu/users/brant/HotDraw/HotDraw.html

[Bru96] P. Brusilovsky. Adaptive hypermedia, an attempt to analyze and generalize. In *Multimedia, Hypermedia, and Virtual Reality*, P. Brusilovsky, P. Kommers, and N. Streitz, Eds., vol. 1077, Springer-Verlag, Berlin, 1996, pages 288-304

[CHEK96] K. Czarnecki, R. Hanselmann, U. W. Eisenecker, and W. Köpf. ClassExpert: A Knowledge-Based Assistant to Support Reuse by Specialization and Modification in Smalltalk. In *Proceedings of the 4th International Conference on Software Reuse*, M. Sitamaran, ed., IEEE Comp. Soc. Press, 1996, pages 188-193

[CI93] R.H. Campbell and N. Islam. A Technique for Documenting the Framework of an Object-Oriented System. In *Computing Systems*, vol. 6, no. 4, Fall 1993, pages 363-389

[CS95] J. Coplien and D. Schmidt, eds. *Pattern Languages of Program Design.* Addison-Wesley, 1995

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995

[GM95] D. Gangopadhyay and S. Mitra. *Understanding Frameworks by Exploration of Exemplars.* In *Proceedings of the International Workshop on Computer-Aided Software Engineering (CASE'95),* 1995

[HHG90] R. Helm, I.M. Holland, and D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *Proceedings of the OOPSLA/ECOOP'90*, SIGPLAN Notices, vol. 25, no. 10, 1990, pages 169-180

[Joh92] R. E. Johnson. Documenting Frameworks using Patterns. In *Proceedings of the OOPSLA'92*, ACM SIGPLAN Notices, vol. 27, no. 10, October 1992, pages 63-76

[Joh94] R. E. Johnson. Documenting Frameworks. In Frameworks Digest, vol. 1, no. 13, Oct. 26, 1994, available at ftp://st.cs.uiuc.edu/pub/FWList/v1n13

[KM96] K. Koskimies and H. Mössenböck. Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs. In *Proceedings of the 18th Int. Conf. on Software Engineering*, IEEE Comp. Soc. Press, 1996, pages 366-375

[Knu92] D.E. Knuth. Literate Programming. Center for the Study of Language and Information, Stanford University, 1992

[LN95] D.B. Lange and Y. Nakamura. Interactive Visualization of Design Patterns Can Help in Framework Understanding. In *Proceedings of the OOPSLA'95*, ACM SIGPLAN Notices vol. 30, no. 10, October 1995, pages 342-356

[Mil67] G.A. Miller. The Magical Number Seven Plus or Minus Two. In *The Psychology of Communication: Seven Essays*, G.A. Miller, New York: Basic Books, 1967

[Min91] B. Minto. *The Pyramid Principle. Part One: Logic in Writing.* Pitman Publishing., London, 1991. First published by Minto International Inc. in 1987

[Mül96] H. A. Müller, Understanding Software Systems Using Reverse Engineering Technologies: Research and Practice. Tutorial Notes, 18th Int. Conf. on Software Engineering, 1996, page 2-12

[OTIa] Object Technology International Inc., *ENVY/Developer R3.01*. User Manual, 1995

[OTIb] Object Technology International Inc., ENVY/QA, see at: http:// www.oti.com/briefs/qa/qabrief.htm.

[Par72] D.L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. In *Communications of the ACM*, vol. 15, no. 12, December 1972, pages 1053-1058

[PPD] Parplace-Digitalk, Inc. *VisualWorks Cookbook*. 1995

[PPSS95] W. Pree, G. Promberger, A. Schappert, and P. Sommerlad. Active Guidance of Framework Development. In *Software—Concepts and Tools*, no. 16, 1995, pages 136-145

[Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995

[SAN88] K.T. Stevens, J.D. Arthur, and R. E. Nance. A Taxonomy for the Evaluation of Computer Documentation. Technical Report, No. TR88-38, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, 1988

[Sch96] D.C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. Available at http://www.cs.wustl.edu/ ~schmidt/ACE.html

[SLMD96] P. Steyaert, C Lucas, K Mens, and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proceedings of the OOPSLA'96*, ACM SIGPLAN Notices, vol. 31, no. 10, October 1996, pages 268-285

[STX] eXept Software AG, Smalltalk/X, see at: http://www.informatik.uni-stuttgart.de/stx/stx.html

[SW96] M.F. da Silva and C.M.L. Werner. Packaging Reusable Components Using Patterns and Hypermedia. In *Proceedings of the 4th International Conference on Software Reuse*, M. Sitamaran, ed., IEEE Comp. Soc. Press, 1996, pages 146-155

[TCFR96] F. Tip, J.-D. Choi, J. Field, G. Ramalingam. Slicing Class Hierarchies. In *Proceedings of the OOPSLA'96*, ACM SIGPLAN Notices, vol. 31, no. 10, October 1996, pages 179-197

[TPS96] S.R. Tillay, S. Paul, and S. B. Smith. Towards a Framework for Program Understanding. In *Proceedings of the 4th Workshop on Program Comprehension*, A. Cimitile and H.A. Müller, eds., IEEE Comp. Soc. Press, 1996, pages 19-28

[Tra95] W. Tracz. DSSA (Domain-Specific Software Architecture) Pedagogical Example. In *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 4, July 1995