Patterns for Effectively Documenting Frameworks

Ademar Aguiar and Gabriel David

INESC Porto, Departamento de Engenharia Informática,
Faculdade de Engenharia da Universidade do Porto,
Rua Dr. Roberto Frias, 4200-465 Porto, Portugal
{ademar.aguiar,gtd}@fe.up.pt

Abstract. Good design and implementation are necessary but not sufficient prerequisites for successfully reusing object-oriented frameworks. Although not always recognized, good documentation is crucial for effective framework reuse, and often hard, costly, and tiresome, coming with many issues, especially when we are not aware of the key problems and respective ways of addressing them. Based on existing literature, case studies and lessons learned, the authors have been mining proven solutions to recurrent problems of documenting object-oriented frameworks, and writing them in pattern form, as patterns are a very effective way of communicating expertise and best practices. This paper presents a small set of patterns addressing problems related to the framework documentation itself, here seen as an autonomous and tangible product independent of the process used to create it. The patterns aim at helping non-experts on cost-effectively documenting object-oriented frameworks. In concrete, these patterns provide guidance on choosing the kinds of documents to produce, how to relate them, and which contents to include. Although the focus is more on the documents themselves, rather than on the process and tools to produce them, some guidelines are also presented in the paper to help on applying the patterns to a specific framework.

Keywords: Patterns, object-oriented frameworks, documentation.

1 Introduction

Software reuse is the activity of using existing software artifacts in the development of new software systems. Different reuse techniques usually use different software artifacts, differing in scale, abstractness and complexity, which range from concrete source-code components to highly abstract architectures. Firmly in the middle of this range, we find object-oriented frameworks, a powerful technique for large-scale reuse capable of delivering high levels of design and code reuse.

An object-oriented framework can be defined as a reusable, semi-defined application that can be specialized to produce custom applications. Through design and code reuse, frameworks help developers to achieve higher productivity, shorter time-to-market and improved compatibility and consistency. When combined with components and patterns, frameworks are considered the most promising current technology supporting large-scale reuse. Perhaps the best evidence of the power of object-oriented

J. Noble et al. (Eds.): TPLOP II, LNCS 6510, pp. 79–124, 2011.

frameworks is reflected on the well-known success of many examples of popular frameworks, such as: Model-View-Controller (MVC), MacApp, ET++, Microsoft Foundation Classes (MFCs), IBM's SanFrancisco, several parts of Sun's Java Foundation Classes (RMI, AWT, Swing), many implementations of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA), Microsoft .NET framewirks, and Apache's frameworks (Cocoon, Struts). Despite the existing difficulties of reusing frameworks, all the above examples of frameworks are playing, directly or indirectly, a very important role in contemporary software development.

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. All of this together implies a steep learning curve that can be significantly reduced with good documentation and training material.

This work contributes with a comprehensive set of patterns focusing on problems of documenting frameworks [2][3][4][5], some of the several technical, organizational, and managerial issues that must be well managed in order to employ frameworks effectively.

1.1 Pattern Language

The pattern language for documenting frameworks comprises a set of interdependent patterns that aim at helping developers to adopt a systematic way of solving, or simply becoming aware of, the typical problems they will face when documenting object-oriented frameworks.

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, activities and tools to the specific context at hands.

The patterns here presented were mined from existing literature, lessons learned, and expertise on documenting frameworks, including a previous compilation of the authors about framework documentation [1].

This document focuses on patterns closely related to the documentation itself, here seen as an autonomous and tangible product independent of the process used to create it. The patterns provide guidance on choosing the right documents to produce, how to relate them, and which contents to include [2][3][4].

From the framework documentation reader's point of view, the most important issue is 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 include: identifying the documentation needs, selecting the contents, choosing the best representation for the contents, and organizing the contents adequately, so that the documentation results of good quality, valuable for the target audience, while easy to produce and maintain.

1.2 Pattern Thumbnails

To describe the patterns, we started from the Christopher Alexander's pattern form [6] and adapted it to include the following sections: Name, Context, Problem, Forces, Solution, Examples, and Known Uses.

Before starting detailing each pattern, we will overview the patterns included in this document by summarizing their intents. Figure 1 shows all the patterns and their relationships.

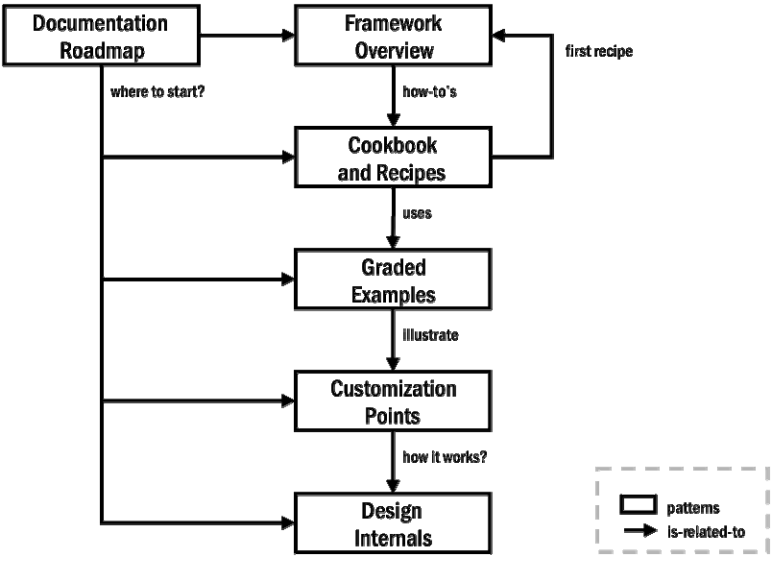


Fig. 1. Documentation patterns and their relationships

DOCUMENTATION ROADMAP helps on deciding what to include in the documentation overview to provide readers of different audiences with useful and effective hints on what to read to acquire the knowledge they are looking for.

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 needs.

COOKBOOK & RECIPES explains how to provide readers with information that explain 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 understanding what they are doing.

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).

CUSTOMIZABLE POINTS describes how to provide readers with task-oriented information with more precision and more design detail than cookbooks and recipes, so that readers can quickly identify the customizable points of the framework (hot-spots) and to understand how they are supported (hooks).

DESIGN INTERNALS explains how to provide detailed 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.

1.3 Pattern-Based Documentation Process

To produce the documentation, there is a life cycle typically organized in five basic activities: configuration, production, organization, usage, and maintenance (Figure 2). Each activity poses their specific problems, closely related to the activities, roles and tools needed to cost-effectively produce the documents [5].

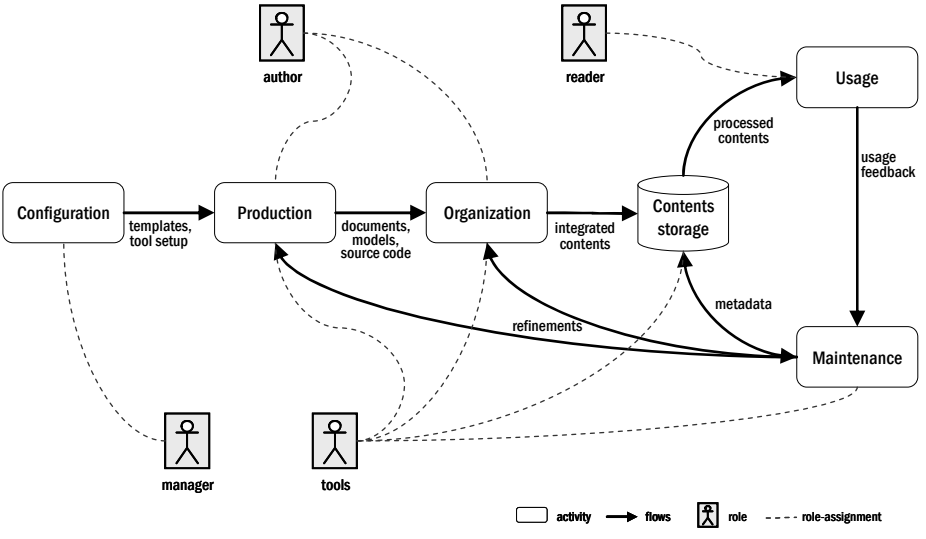


Fig. 2. Typical activities and roles of documenting frameworks

Examples of these problems are: identifying the target audiences, organizing and creating documents, linking related documents, preserving the semantic consistency of duplicated contents, choosing the right documentation tools, and publishing the documents.

Although implicit in the problem-solution pairs of each pattern, there is a very simple method subjacent in all patterns that promote a systematic approach for documenting frameworks. The method aims at being flexible, almost neutral, and easy to adapt to different projects and development environments.

Process Goals

The key goals of the process underlying the application of these patterns are the following:

- **To provide easy-to-follow guidelines.** Although guidelines cannot guarantee quality documentation, they help improve individual quality attributes in a measurable and repeatable way. The goal is to provide some guidelines in the solution section of the patterns, so that writers, especially novices, can find guidance when documenting a framework.
- **To be cost-effective.** To be useful in practice, the cost of documentation must not significantly outweigh its benefits. The goal is to help writers on finding the right balance of the forces, explicitly mentioned in each pattern, for their specific situations.
- **To be user-centered.** Documentation is a means to communicate knowledge to the users, not an end. The goal is to center the process on the user: to analyze user's needs, to design the documentation, to write, test, and refine, until it is ready for usage. The guidelines presented in pattern form suggest a problem-driven approach, thus user-oriented.
- **To support reuse.** To improve documentation productivity, it is important to reuse previous knowledge and artifacts. The goal is to promote the reuse of artifacts (templates, conventions, rules, etc.) and processes (guidelines, tools, integrations). These patterns do not impose any specific template or guideline, but their systematic application helps on finding commonalities that can lead to

Following these goals, a few roles, techniques, activities, and guidelines were defined and embedded in the patterns.

Roles

The documentation is written by different kinds of people, in different phases of the process, which must cooperate and collaborate during the documentation process. There are three roles suggested by the patterns:

- **Technical writers** are responsible to structure, guide, review, and conclude the documentation; this role is especially important for patterns **DOCUMENTATION ROADMAP**, **FRAMEWORK OVERVIEW**, and **COOKBOOK & RECIPES**;
- **Developers** are responsible for content creation mostly during development; this role is especially important for patterns **GRADED EXAMPLES**, **CUSTOMIZABLE POINTS**, and **DESIGN INTERNALS**;
- **Documentation managers** are responsible for configuring and maintaining the documentation base; this role is equally important for all patterns.

Activities

Since the production of framework documentation is closely related with framework design and usage, ideally, these activities should be done in parallel, possibly

side-by-side, if we want to obtain documentation that is understandable, consistent, and easy-to-maintain. The key activities suggested by the patterns are the following:

- **configuration** of the documentation base; this activity is present in all patterns but more explicit in **DOCUMENTATION ROADMAP**;
- **creation and cross-referencing** of the various kinds of contents; this activity is present in all patterns;
- **normalization, integration and storage** of contents; this activity is not considered in the patterns, since it is orthogonal to all of them and would constrain its flexibility;
- **publishing and presentation** of contents to target audiences; this activity is not considered in the patterns, for the same reason as above.

After a configuration phase, the production of framework documentation starts with the creation of the various kinds of contents, and their cross-referencing. Upon creation, the different kinds of contents are normalized, integrated and stored in a repository from where they will be retrieved, transformed, published and presented to target audiences.

What to write	Why	By who	To whom	When
Framework documentation				
Framework overview	to give an overview of the framework: domain, customizations, documentation	writer, developer	all audiences	first release
Cookbooks, recipes	to explain how to use the framework: the points of customization and the steps required to perform	developer, writer	framework users	development
Examples	to provide concrete examples of usage of the framework	developer	framework users	development
Hot spot	to provide information about the points of the framework that accept customization	developer	framework users, framework developers, framework maintainers	development
Hook	to provide detailed design information about how to customize the framework in a specific point of flexibility	developer	framework users, framework developers, framework maintainers	development
Patterns	to provide design information about how the framework works to offer a specific functionality: classes, methods, roles, collaborations	developer, writer	framework users, framework developers, framework maintainers	design
Pattern instances	to provide detailed design information about how the framework was implemented to offer the flexibility and extensibility	developer	framework users, framework developers, framework maintainers	development

Fig. 3. Typical authors for the different types of documents

Guidelines

In addition to the roles and activities presented above, the patterns implicitly provide some guidelines to help writers on effectively producing good quality framework documentation.

The table in Figure 3 lists the major types of documents suggested by the patterns and provides answers to the why's, who's and when's of each type of document. Considering a particular type of document, the meaning of each column of the table is the following:

- **why**, defines its purpose, the goal that it helps to achieve;
- by **who**, defines who are the authors typically responsible for writing it;
- to **whom**, defines who benefits from it;
- **when**, defines the development phase considered the best for writing it.

The list is not exhaustive and is presented only as indicative information, not prescriptive.

1.4 Pattern Application: A Typical Sequence

To better illustrate when and how the patterns above can be applied, let's consider a typical case of an existing framework about which there isn't any kind of information other than its source code.

When new application developers, other than the original framework developers, start trying to reuse the framework, the need for written information becomes evident and the framework team decides to start documenting the framework.

One of the first documents to write is usually for explaining the purpose of the framework and its application domain (pattern **FRAMEWORK OVERVIEW**).

In a second moment, it is important to provide new framework users with concrete examples, from simple to complex ones, which can be used, understood, and learnt by framework users (pattern **GRADED EXAMPLES**).

As documentation grows, the need to keep it organized and easy to access usually leads to the creation of a documentation overview capable of guiding readers directly to the information they look for (pattern **DOCUMENTATION ROADMAP**).

Additional documents can then be written, as needed, to explain how to do the most typical customizations of the framework, in a very pragmatic and prescriptive way (pattern **COOKBOOK & RECIPES**).

When more detailed information is required about the framework's configuration points and their implementation, specialized documentation should be provided at a level of abstraction above the source code. This can be provided either from an external point of view (pattern **CUSTOMIZABLE POINTS**), enumerating all known framework points specifically designed to support customization, or from an internal perspective (pattern **DESIGN INTERNALS**), revealing as much as possible its design internals, most notable the design tradeoffs and decisions made.

The sequence of pattern instantiation above outlined is probably the most common in agile projects, which are very restrictive with documentation effort, suggesting that documentation should be lean and lightweight. In agile projects, documentation is mostly produced collaboratively, when needed, and published publicly, using simple tools.

Other sequences are obviously possible, and should be defined according to the project needs. Independently of the sequence followed, these patterns help on adopting a systematic approach to document frameworks, driven by the specific documentation needs at hands, and accommodate well to different project documentation constraints and needs: completeness, time, cost, target audience, and process discipline and agility.

1.5 Paper Outline

After this introduction to the overall pattern language and its underlying concepts, Section 2 presents a concise review of the key issues of documenting frameworks and related work. The six patterns are then described in the following sections. To illustrate the patterns, examples and known uses are described using different well-known frameworks: JUnit, as a small example; and Java Foundation Classes, .NET, and Eclipse as examples of large frameworks. Although several other frameworks could be used for this purpose, since the patterns are present in several other frameworks, the criteria were to select frameworks widely available, highly popular, and with different budgets. The paper concludes with final considerations about the contribution of these patterns.

2 Key Issues of Documenting Frameworks

By nature, frameworks are much more difficult to document than object-oriented applications or class libraries, being its generativity aspect one of the major reasons. Frameworks are a software product specifically built to create other software products, and therefore they are designed to be flexible and extensible, which makes them hard to explain how to use, and how they work, especially to new users.

There are several issues associated with framework documentation, both technical and non-technical [1] (Figure 4). They include fundamental issues of technical documentation, such as quality assessment, common issues of software documentation, and several issues specific to framework documentation, related with the documentation product itself, the process used to create it, and the tools used to support the process.

General documentation issues	
Technical	Quality assessment
Software	semantic consistency, appropriate documentation environments, knowledge acquisition, programmer's attitude
Framework-specific documentation issues	
Product	Combining prescriptive and descriptive information, managing consistency and redundancy, good organization of contents, graded presentation of contents, several entry points and traversal paths, multiple views and outputs, contents integration
Process	Lack of standards, guidelines, cost, user-centered process, documentation reuse
Tools	Authoring tools, contents synchronization, extraction tools, browse and retrieval facilities

Fig. 4. Enumeration of framework documentation issues [1]

2.1 Quality Assessment Issues

Good quality technical documentation is the ultimate goal of documentation writers, and what system users definitely look for. But to produce good quality documentation comes with many issues, being perhaps the first obstacle the difficulty of defining and assessing its quality.

The book *Producing Quality Technical Information* (PQTI) [31] contain one of the earliest comprehensive discussions about the multidimensional nature of quality documentation. In the revised edition of PQTI, *Developing Quality Technical Information* (DQTI) [30], these dimensions are refined and expanded to nine dimensions organized into three overriding categories: easy to use, easy to understand, and easy to find (Figure 5). DQTI is targeted for general use and already takes in account online information (help, tutorials, etc.).

Quality dimension	Description
Easy to Use	
Task Orientation	Helps users complete tasks related to their work by using the product.
Accuracy	Contains no mistakes or errors, truthful and factual.
Completeness	Includes all essential parts (but only these parts).
Easy to Understand	
Clarity	Contains no ambiguity or obscurity.
Concreteness	Contains no abstractions; including appropriate examples, scenarios, and metaphors.
Style	Uses correct and appropriate writing conventions and word choice.
Easy to Find	
Organization	Organizes material coherently in a way that makes sense to the user.
Retrievability	Presents information in a way that lets users find information quickly and easily.
Visual Effectiveness	Uses layout, illustrations, color, type, icons, and other graphical devices to enhance meaning and attractiveness.

Fig. 5. Quality dimensions for technical information [30]

When addressing problems of software documentation, it is important to be aware of the general issues of technical documentation. However, they are well beyond the scope of the specific topic of framework documentation.

2.2 Software Documentation Issues

Although not always recognized, documentation plays a central role in software development. Most of the development effort is spent on formalizing information, that is, on reading and understanding requirements, informal specifications, drawings, reports, memos, electronic messages, and other informal documents, in order to produce formal documents, such as source code files, models and specifications.

The general issues associated with software documentation become even more important in the context of reusable assets, such as frameworks. These issues include the preservation of the semantic consistency between different documents (such as informal documents and code), the lack of appropriate documentation environments, the knowledge acquisition problem, and the attitude problem of developers in respect to

the priorities of documentation. All these issues have a strong impact on the cost of producing, evolving, and maintaining software documentation.

2.3 Product Issues

The first category of framework documentation issues is related with the documentation itself, seen as an autonomous and tangible product independent of the process used to create it.

The difficulties more specific to framework documentation, as considered by the authors, are related with the following attributes: task-orientation, organization, retrievability, accuracy, and visual effectiveness.

From an external perspective, 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 the help of search and querying facilities. From an internal perspective, 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, and easy to produce and maintain. Here is a quick overview of such issues [1]:

- **Combining prescriptive and descriptive information.** Framework users want practical information (prescriptive information, concrete examples), explaining how-to-use the framework. However, related descriptive information (patterns, contracts, etc.) is also important to be available, to explain how the framework works.
- **Managing consistency and redundancy.** Readers' comprehension is fast when information is provided with accuracy, without inconsistencies between their different kinds of contents (text, source code, models).
- **Good organization of contents.** The external organization is how the contents fit together, as perceived by the reader. A good and easy to understand organization of information helps readers be more effective when using the documentation, because the perceived organization enables them to predict the information they will find in certain parts of the documentation.
- **Graded presentation of contents.** Contents must be organized in a graded, or spiral way, emphasizing main points first, and subordinating secondary points for later. This helps minimize the amount of information needed to read and understand at once.
- **Several entry points and traversal paths.** Documentation readers should be able to follow their preferred learning strategy, such as hierarchical-based or example-based strategies. Due to the potential huge amount of contents, it is important that readers are able to locate framework parts starting from different documentation entry points (e.g. overviews, examples, patterns, etc.).
- **Multiple views and outputs.** The documentation must be configurable to provide a variety of document views (static, dynamic, internal, external, etc.) to effectively support the several kinds of tasks that different people, from different audiences, and with varying levels of experience want to perform.

- **Contents integration.** The internal organization is how the contents are tied together during production. All kinds of framework documents should be easy to integrate in a unified whole, to be easy to share, manage and interchange. In particular, documents must be conveniently integrated with the program code, in order to guarantee semantic consistency and to be easy to navigate from code to documents and vice-versa.

2.4 Process Issues

When documenting a framework, besides deciding *what* to document, the main issues are relative to the documentation process itself: *when* to document, *who* should be responsible for writing particular documentation contents, and *how* to combine and organize the overall documentation contents in order to achieve good quality.

To be useful, framework documentation must include a lot of contents, gathered from different types of documents, at different moments of the development lifecycle, and produced by different kinds of people. To be cost-effective, the production of documentation must follow a well-defined process capable of orchestrating all participants, and promoting their cooperation.

Although there exists a set of documentation practices commonly accepted as useful for documenting frameworks, such as cookbooks, patterns, and examples, processes for documenting frameworks are still missing. The lack of well-defined processes is one of the most important issues of framework documentation processes. Here is a quick overview of such issues [1]:

- **Lack of standards.** The combination of patterns, hooks, hot spots, examples, and architectural illustrations, has proven to be very effective. However, the combination of different kinds of documents may become very difficult and expensive, both to produce and to maintain, due to the several setup efforts and the probable overlap, unless there is a commonly accepted method for documenting frameworks, or a standard for framework documentation, or at least, an infrastructure to help produce, combine and present the contents of framework documentation.
- **Guidelines.** Although checklists, requirements, and guidelines for writing documentation cannot guarantee the development of quality documentation, they help improve individual quality attributes in a measurable and repeatable way, and therefore contribute to improve the overall documentation quality. It is important to get from documentation experts such checklists and guidelines, so that writers, especially novices, can find guidance when documenting a framework.
- **Cost.** To be useful in practice, the cost of documentation must not significantly outweigh its benefits. Especially during framework maintenance, it is very important that the documentation is inexpensive and easy to evolve; otherwise it won't be updated at all, becoming inconsistent, and thus negating the benefits of producing framework documentation.
- **User-centered process.** In order to fine-tune the documentation to the needs of framework users, it is important to follow a user-centered documentation

process, monitoring the usage, getting user feedback, and then reflect the observations in future documentation releases.

- **Documentation reuse.** To improve documentation productivity, it should be reusable, thus allowing extending and modifying it, without making any changes to the original documentation. Documentation reuse can be achieved through reuse of artifacts (templates, conventions, rules, etc.) or reuse of process (guidelines, tools, integrations).

2.5 Tools Issues

One of the reasons for not documenting adequately is the lack of convenient tools supporting rich representations of the developer's mental information. With the goal of ensuring quality and reducing the typically high costs associated with the production and maintenance of framework documentation, it is mandatory to automate the process the best as possible, while retaining the process flexible and easy to adapt to different developers and environments. Examples of activities that would significantly benefit from automation are described below.

- Authoring tools. Attractive, interoperable, and easy to use tools for assisting writers during production and maintenance of documentation would be an important incentive for more and better documentation. The interoperability of authoring tools with development environments is both a standardization and technical issue.
- Synchronization of contents. The usage of tools or automatic mechanisms to synchronize source code, models, and documents is crucial for preserving the consistency between all the contents, especially in the presence of redundancy. Such synchronization would enable to keep design specifications always updated, that is, "alive", thus motivating for the importance of documenting while developing, instead of documenting after developing.
- Extraction tools. To reduce the effort of producing and using the documentation it is sometimes desirable to automatically produce pre-defined views through reverse engineering of formalized software artifacts, such as, source code, models, or formal specifications. Trivial examples are the generation of UML models directly from source code, or the reverse, currently supported in several tools, such as Borland's Together [Borland, 2003], and IBM's Eclipse [Eclipse, 2003]. More sophisticated automation examples include the identification and classification of the hot spots of a framework, or the mining of patterns (or meta-patterns) instantiated in a framework implementation.
- Browse and retrieval facilities. Due to the usual big amount of information contained in framework documentation, it is important to provide readers with good browsing and retrieval facilities, so that the reader don't become lost in the documentation. Such facilities include selection, filtering, searching, querying, navigation, and history mechanisms (automatic or user defined).

All the issues mentioned above are potential sources of complications to document frameworks that may result in powerful frameworks not being reused (i.e. being wasted) due to the difficulty of understanding them. Many of these issues are

interdependent and require proper balancing in order to find effective solutions. Two of these tradeoffs include usage with design information, and prescriptive with descriptive information.

2.6 Usage vs. Design Information

Using a framework and understanding how it works are issues that depend on each other. Framework usage documentation, i.e. documentation describing how a framework is supposed to be used, needs to cover both ways of using a framework: as a black-box system, by simply connecting components, or as a white-box system, by providing extensions to the framework [1].

Much of the work on framework documentation has focused more on ways of representing the design and architecture of frameworks, and less on defining effective ways of helping new users on quickly learning how to use a framework.

One of the reasons for such discrepancy in research effort is perhaps the fact that framework design documentation challenges the capabilities provided by object-oriented design methods and existing software documentation techniques. The design of a framework is usually complex, involving abstract classes and complex object collaborations, thus being best documented with the help of abstractions higher than classes, such as design patterns and role models. In addition, while in software documentation it is possible to clearly separate usage documentation (external view) from system documentation (internal view), with frameworks such separation needs to be fuzzy, as one way of using a framework is by extending it, a kind of usage that requires knowledge about the framework internals [1].

Another possible reason for the discrepancy in framework documentation research effort is that the issues of designing effective user-oriented documentation are typically investigated in the more general fields of technical documentation and computer documentation. In these fields a lot of research is carried on designing usable and understandable task-oriented documentation, which effectively can help users in doing their job, instead of communicating how the system works.

When documenting a software product, we often assume that a good product can be used without knowing how it works, and therefore we clearly separate the user documentation from the system documentation because they have different audiences.

But frameworks are different. The majority of frameworks can be used both as a black-box product and as a white-box product [1]. Therefore, framework usage documentation must mix information both from typical user documentation and system documentation, a situation that poses integration and organization difficulties.

Ralph Johnson authored with other colleagues the first important papers about frameworks [33][34][31][13]. However, Fayad, Johnson, and Schmidt author the most complete source of information about object-oriented frameworks technology, in a set of three volumes that compiles several articles and chapters from different authors [24][25][23]. Older work can be found in Taligent's publications [44][19].

2.7 Prescriptive vs. Descriptive Information

In the article "Documenting frameworks using patterns" [31] Johnson describes a documentation technique that organizes patterns in a pattern language, in a similar

way that recipes are organized in a cookbook. Each pattern describes a recurrent problem in the problem domain covered by the framework, and then describes how to solve that problem.

The primary goal of Johnson's patterns is to teach how to use the framework, and then to complement the task-oriented information with explanations about how the framework works, for those willing to know the details. This documentation technique is an attempt to combine prescriptive information (how-to-do) with descriptive information (how-it-works) in order to result effective for new framework users. The perfect balance between these two kinds of information is difficult to fine-tune to a large and heterogeneous audience, because it depends on the context of use, on the user's experience, and on user goals. To be equally effective for different framework users, the balance would require dynamic adjustment, or otherwise, to be intentionally set by the user.

Besides Johnson's work, other different techniques and styles were proposed, both prescriptive (cookbooks, recipes, etc.) and descriptive (design patterns, meta-patterns, architectural models, etc.), as previously described.

Greg Butler's work on framework documentation is perhaps the most complete and comprehensive existing theoretical work, in the opinion of the authors. Butler introduces the concept of reuse cases to catalog the properties of the existing documenting approaches [15]. Butler et al. define the concept of documentation primitive, an elementary unit of documentation, and define a "task-oriented framework for framework documentation" that relates documentation approaches with documentation primitives [16]. Butler also suggests combining a framework overview with examples, and a cookbook with recipes organized in a graded or spiral way. More descriptive information, such as contracts, architecture, or design patterns, might be also available, and accessible as related material. The work of Butler was used as a starting point for the underlying research of these patterns.

Research and experience have proved the effectiveness of some techniques even when used in isolation, namely: cookbooks in [35], patterns [31], and examples. Despite these empirical results, evidence suggests that framework documentation benefits from a combination of techniques in order to deliver complete reference information, detailed design information, effective usage information, and to result easy to use, easy to understand, and easy to find. This evidence is supported by some empirical investigation [36][35].

Meusel et al. [40] propose a model to structure framework documentation that integrates patterns, hypertext, program-understanding tools, and formal approaches into a single structure that is manipulated to address three different kinds of reuse: selecting, using, and extending a framework. The model is based on the pyramid principle and organizes the documentation into three levels of abstraction, one level for each different kind of reuse. The model supports both top-down and bottom-up learning strategies. The article doesn't mention if and how the consistency between source code and documents is achieved, and how the production of the documentation weaves in the development life cycle.

In practice, the lack of standards, common formats, and tools makes the combination of different kinds of documents difficult and expensive both to produce and maintain, due to the difficulties of managing the redundancy introduced, and ensuring its consistency. This concern with consistency is mentioned but not addressed in related work, with the exception of Demeyer et al [20] that reports on the use of open

hypermedia systems to keep framework cookbooks up-to-date and consistent with framework source code.

3 Pattern DOCUMENTATION ROADMAP

To completely satisfy all audiences and requirements, the documentation of a framework typically includes a lot of information, organized in different types of documents, namely framework overviews, examples of applications, cookbooks and recipes, design patterns, and reference manuals. These documents provide multiple views over the framework (static, dynamic, external, internal) at different levels of abstraction (architecture, design, implementation), which altogether help to grasp all kinds of information that readers may want to look for in framework documentation.

Problem

The complex web of documents and contents provided with a framework must be organized and presented in a simple manner to the different audiences, so that the readers don't become overwhelmed or lost when using the documentation. In other words, the overall documentation must be easy to use by all kinds of readers, so that each individual reader can quickly acquire the strict degree of understanding she needs to accomplish her particular engineering task (reuse, maintenance or evolution).

> **How to help readers on quickly finding in the overall documentation their way to the information they need?**

Forces

- **Different audiences.** Readers of different audiences have different needs and interests that must be addressed by the documentation.
- **Different kinds of reuse.** A framework can be reused in different ways, each requiring different levels of knowledge about the structure and behavior of the framework, and therefore pose different demands on the documentation.
- **Easy-to-use.** Despite its inherent potential complexity, the documentation must result easy-to-use.

Solution

Start by providing a roadmap for the overall documentation, one that reveals its organization, how the pieces of information fit together, and that elucidates readers of different audiences about the main entry points and the paths in the documentation that may drive them quickly to the information they are looking for, especially at their first contact.

The roadmap would help both when navigating top-down, from a main entry point of the documentation to concrete topics and subtopics, and when navigating bottom-up from a small piece of information to a bigger one to try to identify it as part of a whole, still unknown in a first contact.

To be effective, a documentation roadmap for a framework should be written in a task-oriented manner and to include the following aids:

- topics organized by audience, kind of task, and order of use, to help readers of different audiences quickly retrieval of the information they need to accomplish the tasks they have in mind;
- emphasis of the main entry points and subordination of the secondary ones, to improve visual effectiveness;
- overview of topics conveying how subtopics are related, to support non-linear readings;
- transitions and links to topics, and from them to the roadmap again, to improve the overall navigability around the roadmap.

Although all the above mentioned aids are important to include, in fact, the optimal level of importance assigned to each one depends on several factors: the *framework* being documented, which *audiences* are you willing to address in the overall documentation, which *kinds of reuse* to support explicitly in the documentation, and which *tasks* to emphasize and subordinate.

Independently of how these factors are balanced, the roadmap should be easy to use, easy to understand, well organized, and visually effective, a set of quality characteristics that suggest not to include everything in the roadmap, but only the entry points and topics more relevant for the most important tasks of the target audiences.

Examples

JUnit. The JUnit framework [9] provides a very simple documentation roadmap, where the audiences, kinds of reuse and tasks are implicitly mentioned in the names of the entry points in the documentation and their respective descriptions (Figure 6).

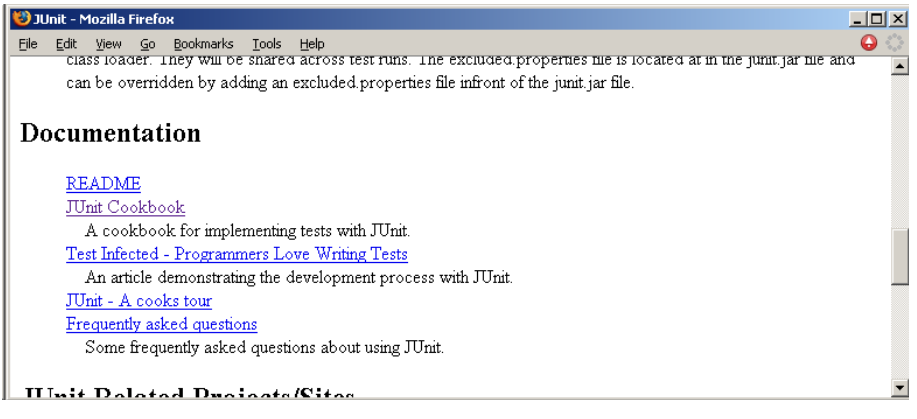


Fig. 6. Example of a very simple documentation roadmap delivered with JUnit

The audiences addressed can be implicitly identified as the following:

- *first-time users*, which will probably be attracted by the "JUnit Cookbook" entry;
- *selectors* and *common users*, by the "Test Infected - Programmers Love Writing Tests" entry;
- and *framework developers*, by the "JUnit - A cooks tour" entry.

The kinds of reuse clearly expressed in this roadmap are:

- *instantiating*, by the "JUnit Cookbook" entry;
- *selecting* and *instantiating*, by the "Test Infected - Programmers Love Writing Tests" entry;
- and *mining*, by the "JUnit - A cooks tour" entry.

Finally, the tasks mentioned are simply *how to implement tests*.

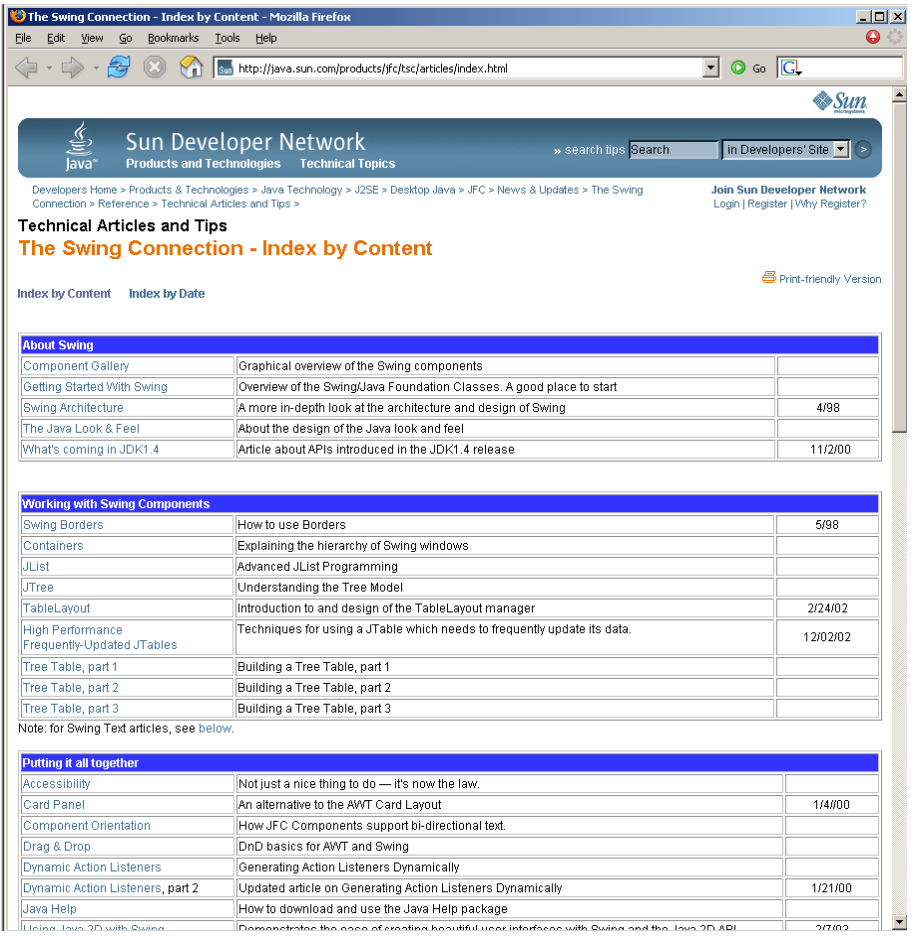


Fig. 7. Example of the complex documentation roadmap for JFC/Swing framework

This roadmap reflects the simplicity of the JUnit's framework itself, the intent of the writers on making the documentation simple and short, by documenting the main usage task of JUnit: to write tests.

Swing. The Sun's JFC/Swing framework [20] is a popular example of a large framework for which exists a lot of documentation.

In part due to its large dimension and vast diversity of possible tasks when reusing Swing, be it black-box or white-box reuse, the roadmap is split along several documents, according to the kind of audience and reuse tasks.

Figure 7 presents "The Swing Connection - Index by Content", which is the document of Swing that mostly resembles a framework documentation roadmap.

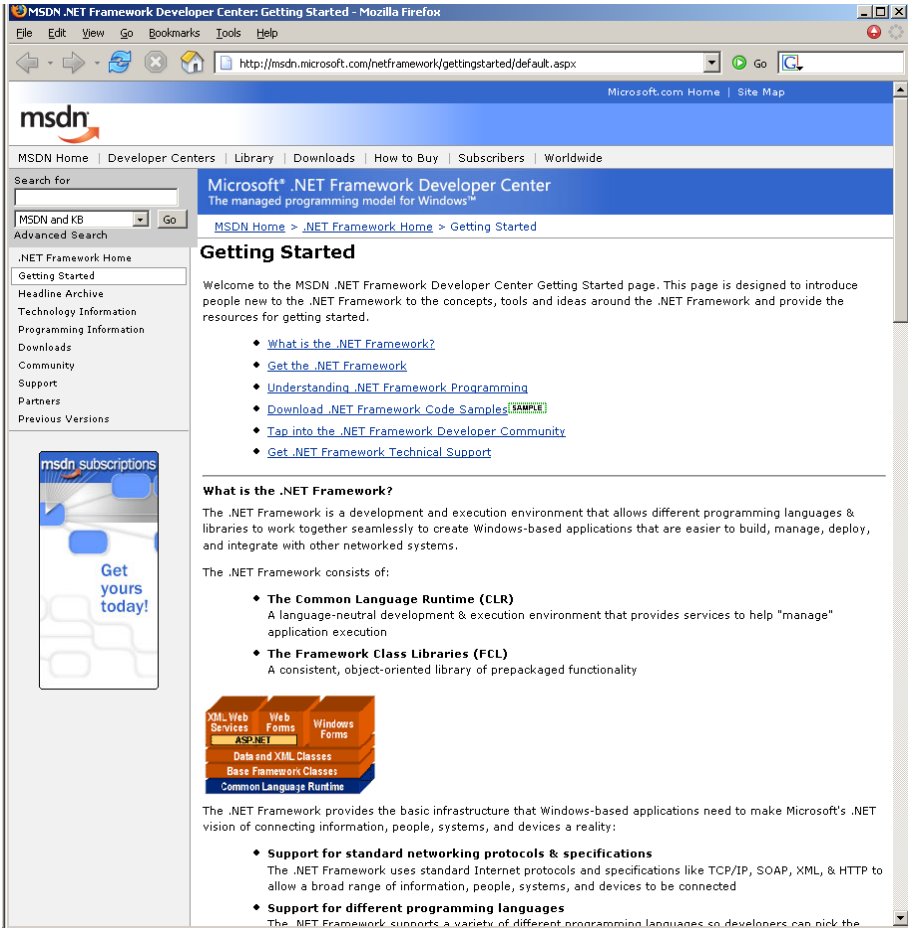


Fig. 8. Example of the documentation roadmap for the .NET framework

.NET. The Microsoft's .NET framework is another popular example of a very large framework for which also exists a lot of documentation.

For this framework there exists a documentation roadmap that clearly addresses different audiences and different kinds of reuse, from where the reader is driven to other documents more specific to the kind of reuse selected with more detailed about the tasks possible with the framework.

Figure 8 presents the "Getting Started" document, which is intended to first-time (re)users, a secondary document accessible from the right-hand side menu that contains the main entry points to the documentation, organized by kind of audience.

4 Pattern FRAMEWORK OVERVIEW

To be effective, the documentation of a framework must include information that explains the purpose of the framework, how to use it, and how it is designed and implemented [39][16].

Problem

In addition to different purposes, the documentation of a framework must also meet the needs of different categories of software engineers involved in framework-based application development, playing different roles (framework selectors, application developers, framework developers, framework maintainers, and developers of other frameworks [15]), having varying levels of experience, and therefore requiring different kinds of information.

How to help readers on getting a quick, but precise, first impression of a new framework?

Forces

- **Different audiences.** In a first contact, the most important kinds of readers to consider are framework selectors, someone who is responsible for deciding which frameworks to use in a project, and new framework users, developers without previous knowledge or experience with the framework [1]. The first kind of information that a new framework user looks for is contextual information. Framework selectors will look for a short description of the framework's purpose, the domain covered, and an explanation of its most important features, ideally illustrated with examples.
- **Completeness.** The readers appreciate complete information, i.e. that all of the required information is available. Completeness implies that all of the relevant information is covered in enough detail, but only the necessary, and that all the promised information is included. But completeness depends on the reader's point of view, and therefore requires knowing the audience and the tasks the documentation should support [30].
- **Easy-to-understand.** Information that is clear, concrete, and written using an appropriate style is usually easy to understand the first time. Clarity (especially

conciseness) often conflicts with completeness (especially relevance and too much information), requiring a good knowledge about *what* readers need to know and *when* they need to know it [30]. Concrete examples help readers on understanding what they are learning because they map abstract concepts to concrete things, which readers can see or manipulate.

Solution

Provide an introductory document, in the form of a framework overview, that describes the domain covered by the framework in a clear way, i.e. the application domain and the range of solutions for which the framework was designed and is applicable.

In addition, a framework overview usually defines the common vocabulary of the problem domain of the framework. It clearly delineates what is covered by the framework and what is not, as well as, what is fixed and what is flexible in the framework. An effective way of communicating this information consists on presenting the basic vocabulary of the problem domain illustrated with a rich set of concrete application examples.

This information is of great value for potential users especially during the selection phase because it helps them to evaluate the appropriateness of the framework for the application at hands, and thus fundaments the selection, or rejection.

In a framework overview, it is common practice to refer or review examples, from simple to complex ones (pattern **GRADED EXAMPLES**), and to refer or include an overview of all the documentation (pattern **DOCUMENTATION ROADMAP**). A framework overview is often the first recipe in a cookbook (pattern **COOKBOOK & RECIPES**).

Examples

JUnit. The framework overview that accompanies JUnit is represented in Figure 9. It was extracted from the document "Frequently Asked Questions (FAQ)" [16], which, from all the documents delivered with the JUnit's framework, is the one that most clearly presents the information typical of a framework overview, despite its placement in a FAQ not being evident in a first contact with the documentation.

Although not being a good exemplar of a framework overview, it contains its most basic ingredients (the domain covered, the features, and an overview of the documentation), and thus it reasonably fulfils the requirements of a framework overview.

The biggest problem with the JUnit's framework overview is that it is not easy to find in a first look at the documentation and is not complete. It is however, easy to understand, what is also very important.

Swing. The framework overview of JFC/Swing is provided in the Sun's tutorial "Creating a GUI with JFC/Swing", lesson "Getting Started with Swing", topic "About the JFC and Swing" (Figure 10).

This framework overview clearly describes the domain covered by the JFC/Swing and its main features. Although, the customizations possible with the framework are described textually in the overview, they are not linked to the concrete examples of applications included in the documentation in other lessons of the tutorial, both visual and source code examples.

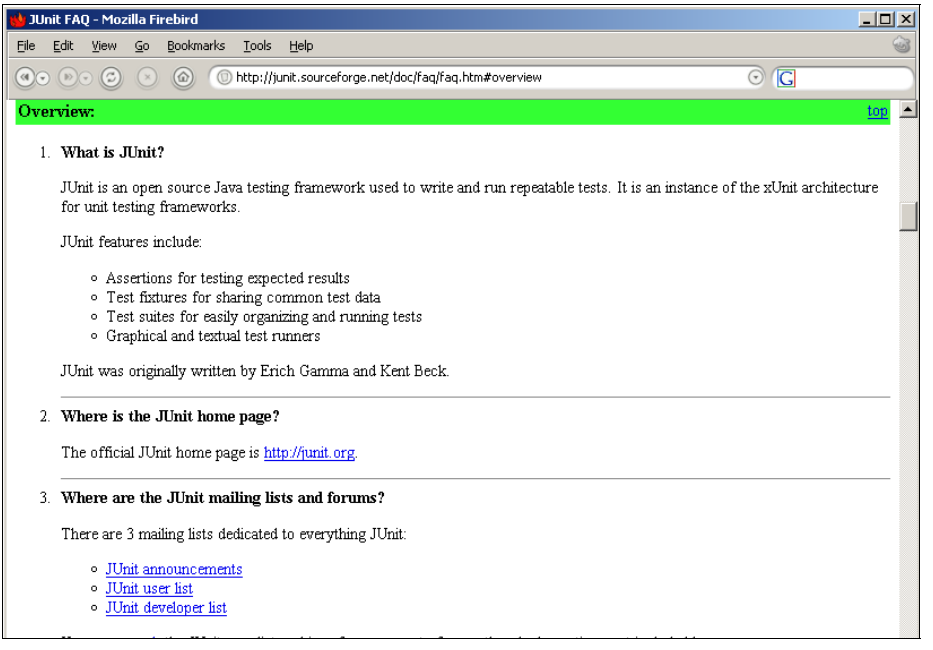


Fig. 9. Example of the framework overview delivered with JUnit

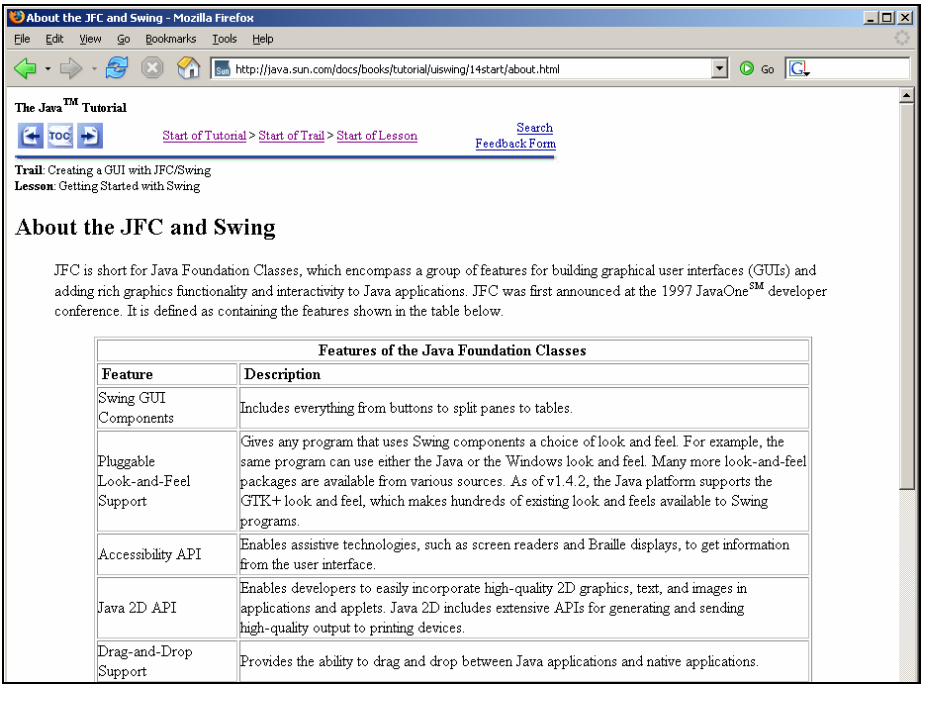


Fig. 10. Example of the framework overview provided for JFC/Swing

Being the JFC/Swing framework so well-known, this placement of the overview so deep in the documentation hierarchy, instead at the top, to be easy to find in a first contact with the documentation, is acceptable and possibly even more convenient, considering that only a small minority of readers are expected to need to learn what JFC/Swing is about. What most readers would need to learn is *what* and *how* they can customize in JFC/Swing to create the GUI they have in mind.

The contents and organization of this framework overview reflects the importance of knowing well the audience and the tasks more likely to need support from the documentation.

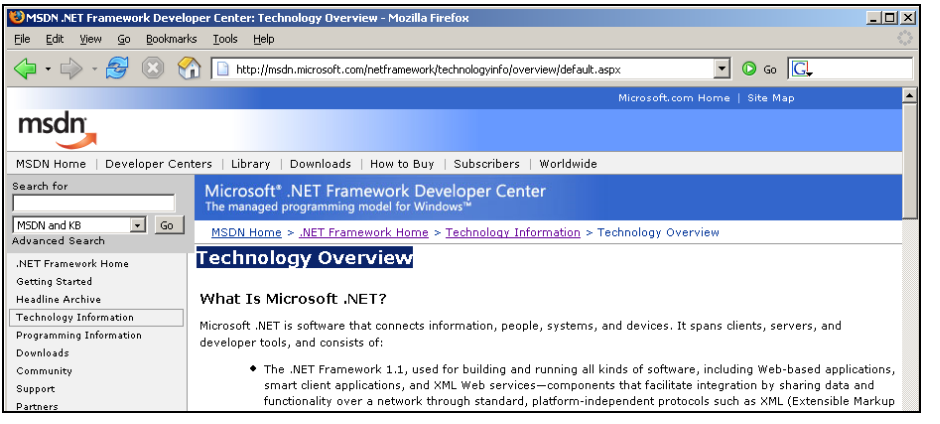


Fig. 11. Example of the framework overview provided for the .NET framework

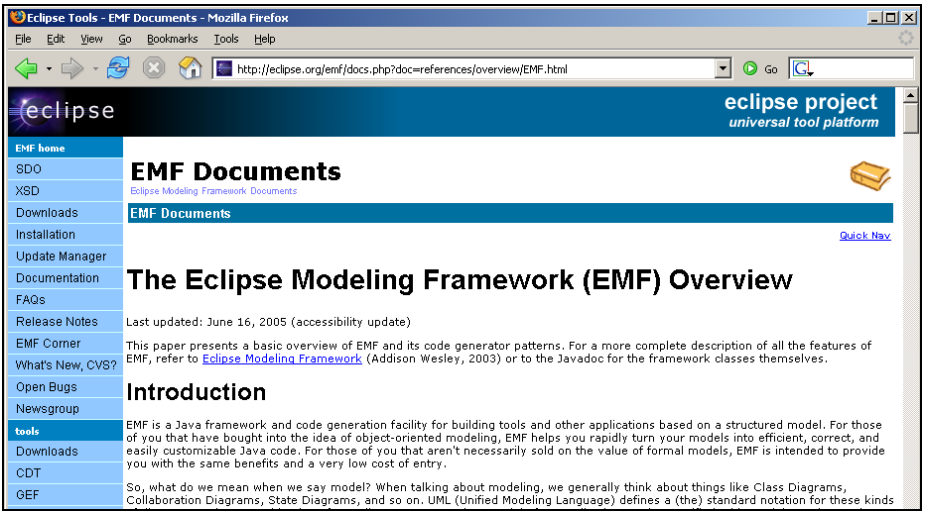


Fig. 12. Example of the overview provided with the Eclipse Modeling Framework (EMF)

.NET. The overview of the .NET framework is provided in the "Getting Started" document (Figure 8), topic "What is the .NET Framework?". A more detailed technical overview is provided in the "Technology Overview" document (Figure 10).

This framework overview briefly describes the purpose of the framework, its application domain and its main components, and refers other documents containing more specific information about the framework.

EMF. The overview document of the Eclipse Modeling Framework (EMF), "Eclipse Modeling Framework Overview", is another good example (Figure 12). It is very complete and provides a brief presentation of the framework and its key features illustrated with the help of concrete examples.

5 Pattern COOKBOOK AND RECIPES

The quality of information explaining how-to-use the framework is very important to achieve its effective reuse. In most of the cases, the informal communication channels between framework developers and framework users are not available anymore, and, as a result, all information has to be communicated in alternative ways, with different kinds of manuals delivered with the framework, possibly using different media (documents, animations, videos).

Problem

To help application developers being effective on the reuse of a framework, ideally, the documentation should be organized in a way that can help readers quickly locate the explanations they strictly need to learn how to use the framework parts required to implement the specific features of the application at hands.

> **How to quickly provide users with information that helps them learning how to use the framework?**

Forces

- **Task-orientation.** Readers want to learn how to use the framework, so the documentation must focus on real tasks that users really want to perform, instead of artificial tasks, imposed by the framework.
- **Balancing Prescriptive and Descriptive information.** 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. This perfect balance may vary with the reader of the documentation, and thus the "one size fits all" solution doesn't work here.
- **Different Audiences.** Readers of different audiences have different needs and interests that must be addressed by the documentation. New framework users want to identify, understand and manipulate the flexible features of the framework they need, as quick as possible, without being forced to understand

the detail of the whole design, but only its basic architecture (static and dynamic views). More advanced users may want to learn how to do less common customizations that possibly require a more detailed understanding of the framework's internal design.

- **Completeness.** Readers appreciate complete information, i.e., that all possible customizations are mentioned, with all the possible detail, but this is not always feasible, as it largely depends on the reader's point of view and the tasks to support.
- **Easy-to-use.** The resulting documentation must be easy to use, otherwise readers would need to spend more time than needed to browse it, and can get lost on it.
- **Cost-effectiveness.** Unfortunately, many frameworks are very complex to use and understand resulting hard, tedious and expensive to document in detail.

Solution

Provide a collection of recipes, one for each framework customization, organized in a cookbook, which acts as a guide to the contents of all its recipes.

Cookbook. A good cookbook is specific to a particular framework. Users search the cookbook for the recipe that is most appropriate for their needs, and when found, they follow the steps it describes. Recipes in a cookbook are usually organized in a spiral approach, where the most common forms of reuse are presented early, and concepts and details are delayed as long as possible. A *framework overview* is often the first recipe in a cookbook [35], being responsible for presenting the framework.

Recipe. A recipe is a document that describes how to use the framework to solve a specific problem of application development [42]. Recipes typically present information in natural language, perhaps illustrated with some pictures and code fragments. Although recipes are rather informal documents, they are usually structured in:

- a purpose section, which describes the problem the recipe is meant to solve plus some of the limitations of the solution;
- a procedure section, or how to do section, containing a sequence of the basic steps necessary to carry out the solution, and
- an examples section containing explicit, or implicit, references to models, examples and fragments of source code, and also to other related recipes.

The most important kind of information provided by cookbooks and recipes is prescriptive information that instructs users on how to use and customize the flexibility features provided by the framework. In addition, they also informally describe architectural constructs and design details, but only in a very small amount, the strictly needed to help users minimally understand what they are doing.

The best kind of documentation for beginners seems to be one that provides detailed instructions for using each individual feature of the framework without describing in detail all the theory behind them [17][39]. Considering that the main purpose of a framework is to reuse design, if it is well designed, then there must be large parts of its design that are easy to reuse even without knowing them.

On the other hand, to start using a framework without having a clear understanding of it seems to be wrong at first, but the fact is that people can't understand well a framework until they have really used it. The effective understanding of a framework thus requires that theory follows practice [39]. After the first use, the framework user has a much better understanding of what the framework does and is more capable to understand how it works, i.e. to understand its internal design details.

Related Approaches

Active cookbook. The active cookbook is a specialization of the cookbook concept that provides active guidance to framework users [39]. Active cookbooks extend the cookbook idea with a hypertext representation, a visual development environment and supporting tools. The tool support provided by active cookbooks helps the user navigate the steps of the recipes and provide the tools needed in each step, thus increasing the productivity.

Johnson's patterns. Johnson documented the HotDraw framework [39] using a pattern language comprising a set of patterns, one pattern for each recurrent problem of using the framework. The pattern language organizes the documentation, as a cookbook does with the recipes, and each pattern provides a format for each recipe.

To avoid confusion with design patterns, the term motif was later introduced in [38] to name Johnson's patterns [39]. The description of a motif has sections similar to a recipe, except that use additional references to design patterns, to provide information about the internal architecture, and references to contracts for a more rigorous description of the collaborations relevant to the motif.

Hook description. Hook descriptions were first introduced in [26] and present knowledge about framework usage, providing an alternative view to design documentation. Hook descriptions 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: a hot spot tends to have several hooks within it. The usage of hooks can be semi-automated.

Examples

Cookbooks and recipes were historically the first technique used to document frameworks, namely the MVC framework [35] and the MacApp framework [8].

JUnit. Figure 13 presents a recipe for writing a simple test with JUnit, the first of the five recipes contained in the cookbook provided with JUnit, the "JUnit Cookbook" document [9].

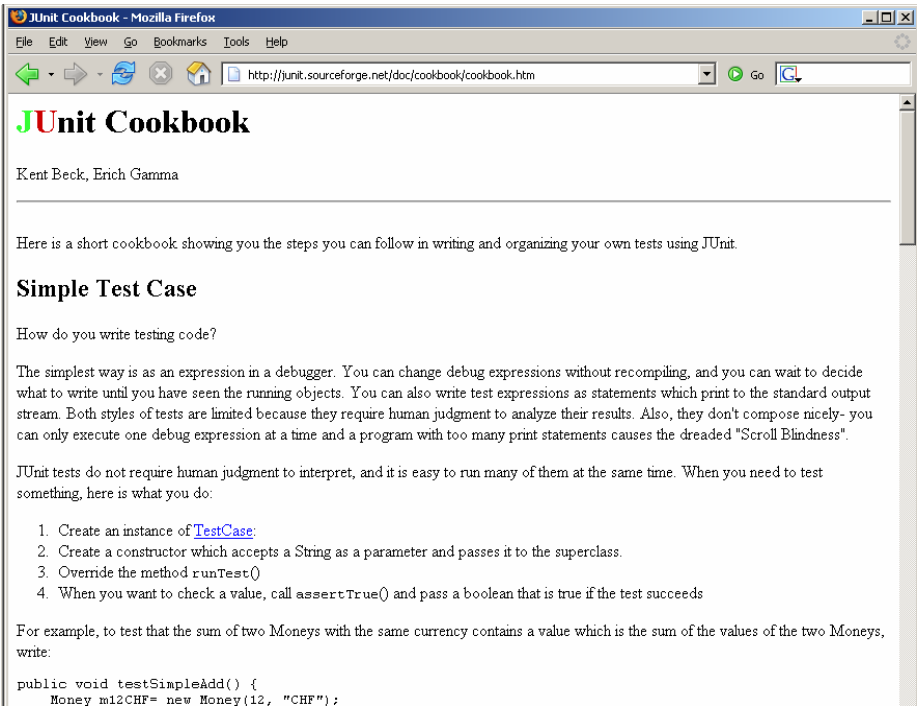


Fig. 13. Example of a recipe for writing a simple test with JUnit

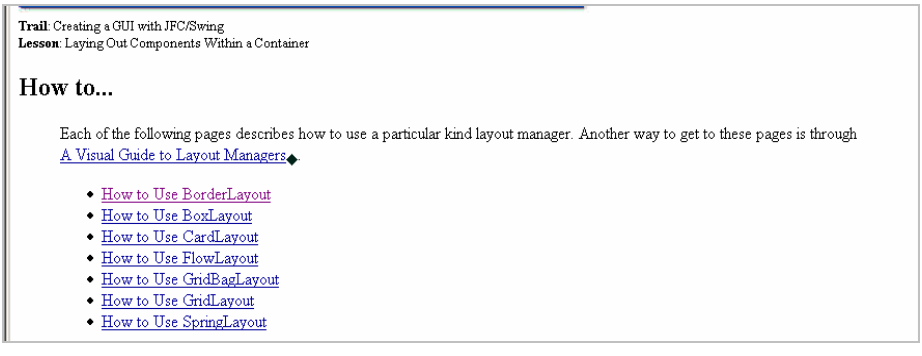


Fig. 14. Example of a cookbook from Swing framework to teach how to use layout managers

Swing. The Sun's JFC/Swing framework [20] contains several recipes organized in several cookbooks. Figure 14 shows the cookbook related with layout managers, which contains a set of recipes explaining how to use layout managers. Figure 15 shows a specific recipe explaining how to use a BorderLayout manager.

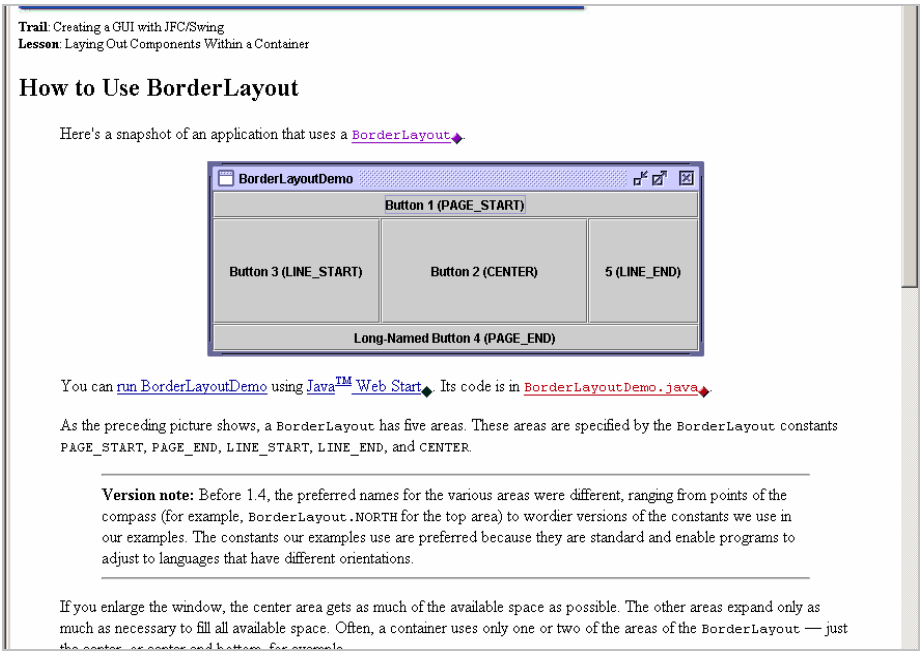


Fig. 15. Example of a recipe from Swing framework to teach how to use the BorderLayout manager

Consequences

Providing a cookbook and recipes for a framework helps on improving its task-orientation, usability, and on combining prescriptive with descriptive information.

The main audience targeted by cookbook and recipes are framework users, i.e., developers of applications intending to reuse the framework as part of it. Due to its user-orientation, this kind of documentation is usually easy to use and understand, but not complete, i.e. it doesn't intend to cover all the possible customizations with the framework but only the most relevant for the audience on target.

Because cookbook and recipes focus on teaching how-to-use a framework, it is useful to combine them with concrete or running examples, and references to the related **CUSTOMIZABLE POINTS** and source code, thus adding some redundancy, which although useful must be managed to preserve its consistency.

6 Pattern GRADED EXAMPLES

Information explaining what for a framework can be used is of great value for potential users, as it helps them to evaluate the appropriateness of the framework for the application at hands, and thus fundaments its selection, or rejection.

Problem

The first usage of a framework is always the hardest one because it may simultaneously involve the learning of the problem domain, the domain covered by the framework, and the range of solutions for which the framework was designed and is applicable. When illustrated with examples, first and subsequent usages of a framework can be easier, and more complex usages may become more interesting.

How to help readers on evaluating the appropriateness of a framework to his particular application at hands?

How to help readers on getting started fast using a framework both for simple and complex kinds of usage?

Forces

- **Task-orientation.** Framework users want practical information illustrating what can be done with the framework, and how-to-do it.
- **Different audiences.** A framework selector is someone (manager, project leader, developer) who is responsible for deciding which frameworks to use in an application development project. Among other information, framework selectors will look for the domain covered and an explanation of the most important features of the framework. New framework users want to identify, understand and manipulate the flexible features of the framework they need, as quick as possible, without being forced to understand the detail of the whole design, but only its basic architecture (static and dynamic).
- **Cost-effectiveness.** For many frameworks, the source code of example applications is the first and only documentation provided to framework users, as examples can be produced during framework development without extra effort.

Solution

Provide a small but representative graded set of training examples to illustrate the framework's applicability and features, each one illustrating a single new way of customization, smoothly growing in complexity, and eventually altogether providing complete coverage. The usage of hypertext links in the source code and the availability of executable code are valuable aids for better understanding of the examples.

Set of examples. A good set of examples, ranging from simple to complex ones, can serve as a live catalogue for the basic vocabulary of the problem domain and the key features of the framework, constituting a perfect complement to all other purposes of documentation. A proper selection of examples can be very effective for illustrating the domain covered, how-to-use the framework, and revealing some design internals. When selecting the examples to include, it is important to consider their growing level of complexity, in order to cover different levels of complexity, from low to high.

Examples. Examples play a key role in framework documentation as they make a framework more concrete; they help on understanding the flow of control, and are easier to understand than design abstractions, although less general.

The source code of example applications constructed using the framework is often the first documentation provided to application developers. Linking documents to source code and providing executable code may significantly help on understanding the examples.

The cost of producing examples to deliver with the framework is usually not high, if well planned, as the examples can also be used to drive the development, to verify the real reusability of the framework, and to help on documenting the framework.

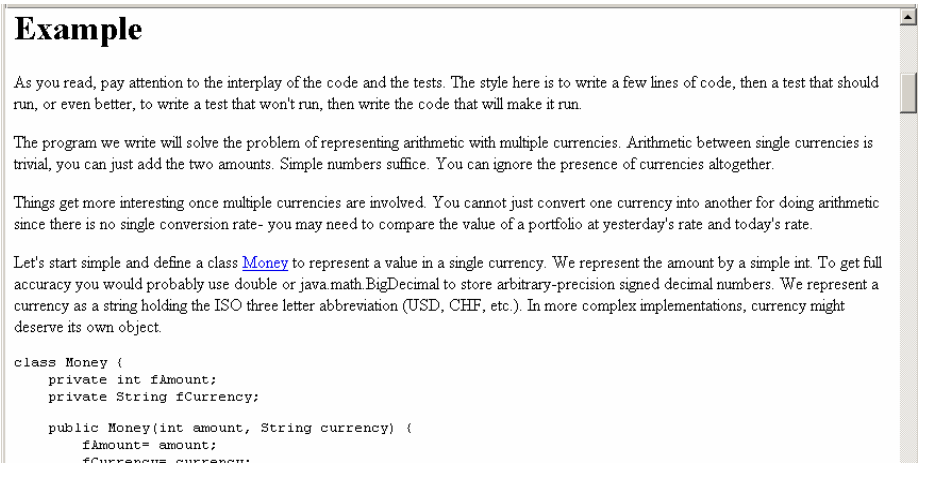
The examples must be used to show what the framework is good for, and not for showing how to use the framework, nor for explaining how the framework is designed.

Because good examples help new users on getting started fast, the study of working examples is a nice and motivating way of learning a framework, and help drive the learning of the framework to the points of most interest to users, thus making the learning more effective.

Examples

The documentation of many successful frameworks provides a lot of examples, which make them easier to understand, to use and extend [29]. It was observed that the most typical documentation of successful frameworks includes examples that work right "out-of-the-box". Some examples are: MVC [39], ET++ [45], and Java Swing [28].

JUnit. Figure 16 presents an extract from the article "Test Infected: Programmers Love Writing Tests" [11] that uses and describes an example named `MoneyTest` provided with JUnit.



Example

As you read, pay attention to the interplay of the code and the tests. The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won't run, then write the code that will make it run.

The program we write will solve the problem of representing arithmetic with multiple currencies. Arithmetic between single currencies is trivial, you can just add the two amounts. Simple numbers suffice. You can ignore the presence of currencies altogether.

Things get more interesting once multiple currencies are involved. You cannot just convert one currency into another for doing arithmetic since there is no single conversion rate- you may need to compare the value of a portfolio at yesterday's rate and today's rate.

Let's start simple and define a class Money to represent a value in a single currency. We represent the amount by a simple int. To get full accuracy you would probably use double or java.math.BigDecimal to store arbitrary-precision signed decimal numbers. We represent a currency as a string holding the ISO three letter abbreviation (USD, CHF, etc.). In more complex implementations, currency might deserve its own object.

```
class Money {
    private int fAmount;
    private String fCurrency;

    public Money(int amount, String currency) {
        fAmount= amount;
        fCurrency= currency;
```

Fig. 16. MoneyTest: an example provided with JUnit

Swing. Swing provides a rich set of graded examples, very helpful for both new and experienced users on better evaluating the applicability and feasibility of the framework regarding the specific needs of the application at hands. Figure 17 shows part of a Swing tutorial fully based on examples.

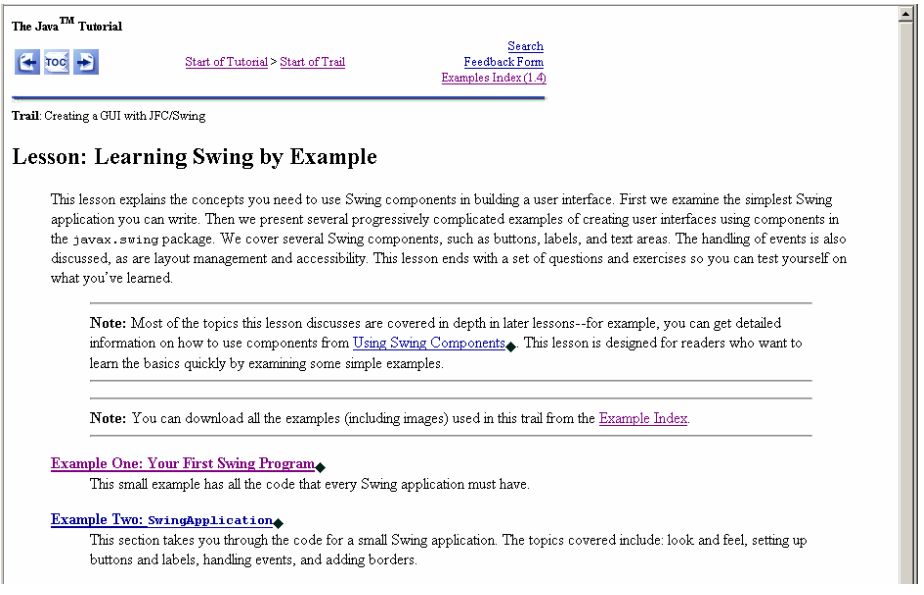


Fig. 17. An example-based tutorial provided with Swing

Consequences

Providing a good set of graded examples for a framework, ranging from simple to complex ones, is usually the less expensive kind of documentation. Examples are useful during framework development and testing, and help both new and experienced users on quickly evaluating what can be done with the framework and also reveal a little about how to use it. Examples are very task-oriented and can thus reach several audiences at the same time but they don't completely satisfy all of them (e.g. framework maintainers). Despite their value, they are however not sufficient to completely document a framework.

It is useful to include in the examples references to the **COOKBOOK & RECIPES** explaining the **CUSTOMIZABLE POINTS** used in the example, optionally referencing information about the **DESIGN INTERNALS** and source code, but this adds an extra need for maintaining the consistency of these references.

7 Pattern CUSTOMIZATION POINTS

You are documenting a framework to provide application developers with prescriptive and descriptive information capable of helping them customize the framework.

Problem

To help application developers customize a framework effectively, the documentation should be organized in a way that can help readers obtain detailed information quickly. Developers typically look for prescriptive and descriptive information to learn *which* framework parts are strictly required to customize, and *how* to customize them, to be able 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 know which framework parts are customizable?

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

Forces

- **Task-orientation.** As readers want to learn in detail how to use a certain customizable part of the framework, 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 describing 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. 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, but this 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 (clarity, easy-to understand and navigate).

Solution

Provide a list of the framework's *customization 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 customization 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 [26]. 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 18 below shows an example of both kinds of 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.

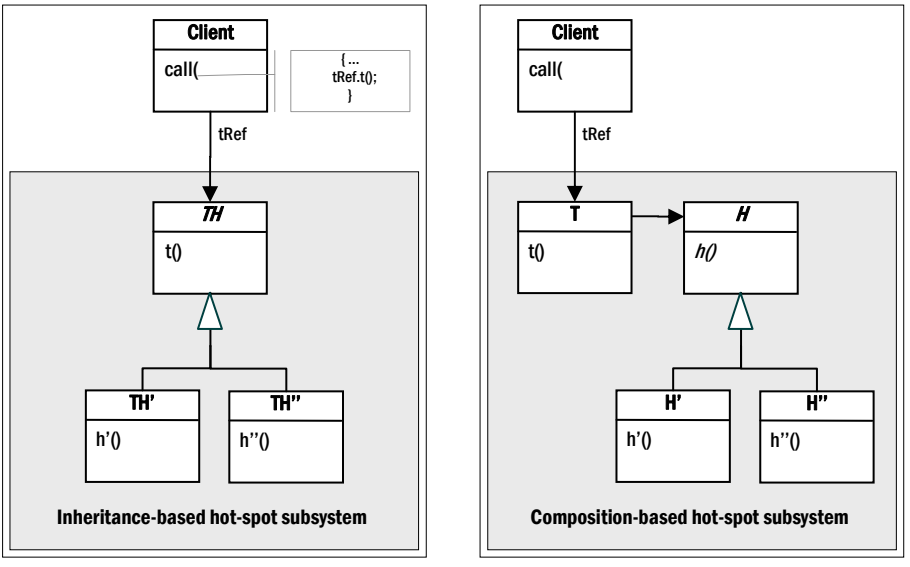


Fig. 18. Inheritance-based and composition-based hot-spot subsystems

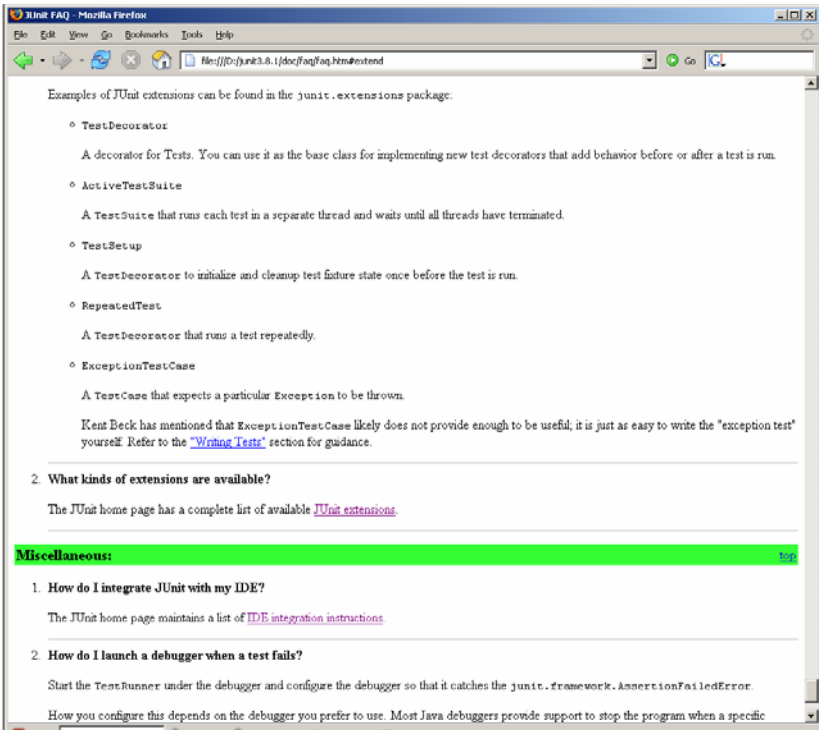


Fig. 19. JUnit: hot-spots are only implicitly mentioned in the FAQ

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 [9].

However, some more customizations can be done with JUnit, such as test runners, and test decorators, but information about these and other less used customization points is only briefly mentioned in the "JUnit FAQ" document [18] and in the low-level Javadoc documentation. Figure 19 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 20).

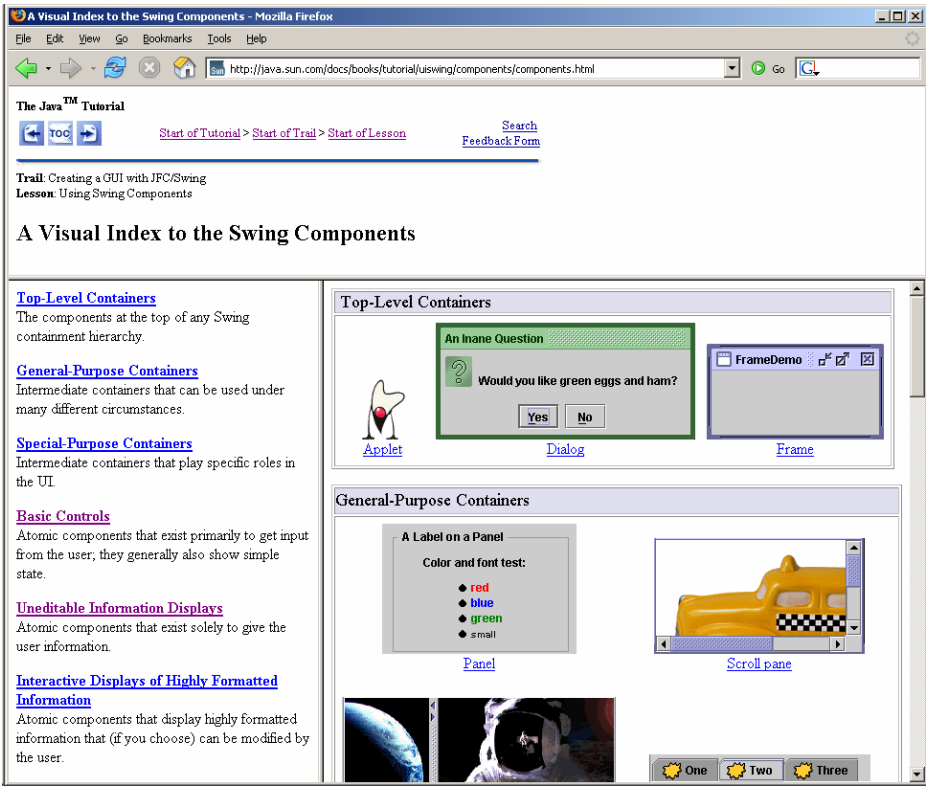


Fig. 20. "A Visual Index to the Swing Components"

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 21-left), which gives access to more detailed lists of possible customizations of each component (Figure 21-right). Even more detailed information about how the flexibility is supported in each customization point although not explicit in the documentation, is left to the reader to explore by herself, probably using the Javadoc comments and source code inspection.

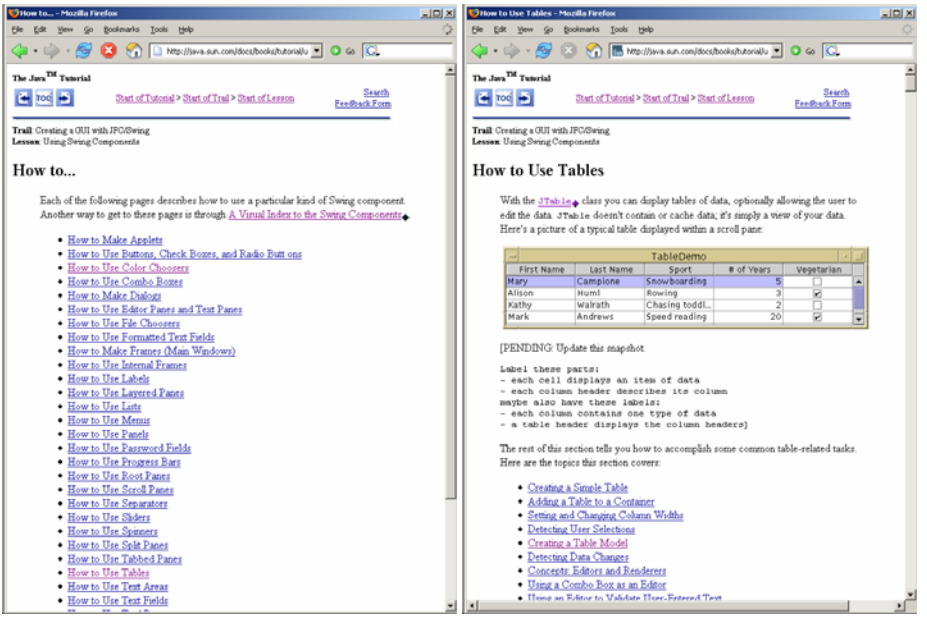


Fig. 21. List of the most frequently used customizations possible with Swing and Tables

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 if the framework is applicable to the problems at hands, and therefore to decide with more confidence whether or not to reuse it.

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 & 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 & RECIPES** related with those customization 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 provide a good balance between prescriptive and descriptive information thus being a good complement to the prescriptive information of **COOKBOOKS & RECIPES** and the descriptive information of **DESIGN INTERNALS**.

8 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 on how to reuse it in advanced ways.

Problem

Framework instantiation for a particular application often requires 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, and also other 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.

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

Forces

- **Different purposes.** In addition to the framework purpose and usage instructions, the documentation must also help framework users on understanding the underlying principles and the basic architecture of the framework. With such detailed information, users are able to develop not only trivial and planned applications, but also advanced ones, 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 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 may include not only the different classes of the framework, but also the strategic roles and collaborations of their instances, rules and constraints, such as cardinality of framework objects, creation and destruction of static and dynamic framework objects, instantiation order, and synchronization and performance issues.

Solution

Provide concise detailed information about the design internals of the framework, especially the areas designed to support configuration, known as hot-spots. You should start by describing how the framework hot-spots support configuration. This can be done, for example, by describing the roles of framework participants using *design patterns* and *design pattern instantiations*, or at a meta-level using *meta-patterns*, or, even at source-code level.

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 [40].

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.

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* [6]. 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 [6]. 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 [39]: 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 for documenting frameworks because they capture design experience at the micro-architecture level and capture meta-knowledge about how to incorporate flexibility [27][13]. In fact, design patterns are capable of illuminating and motivating 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 [27] and POSA patterns [14]. 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 [39], itself documented under the form of a pattern language.

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 [42].

Examples

Design patterns are commonly used to document the global architecture of the framework. 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 [39], 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 & RECIPES**), where each pattern provides a format for each recipe.

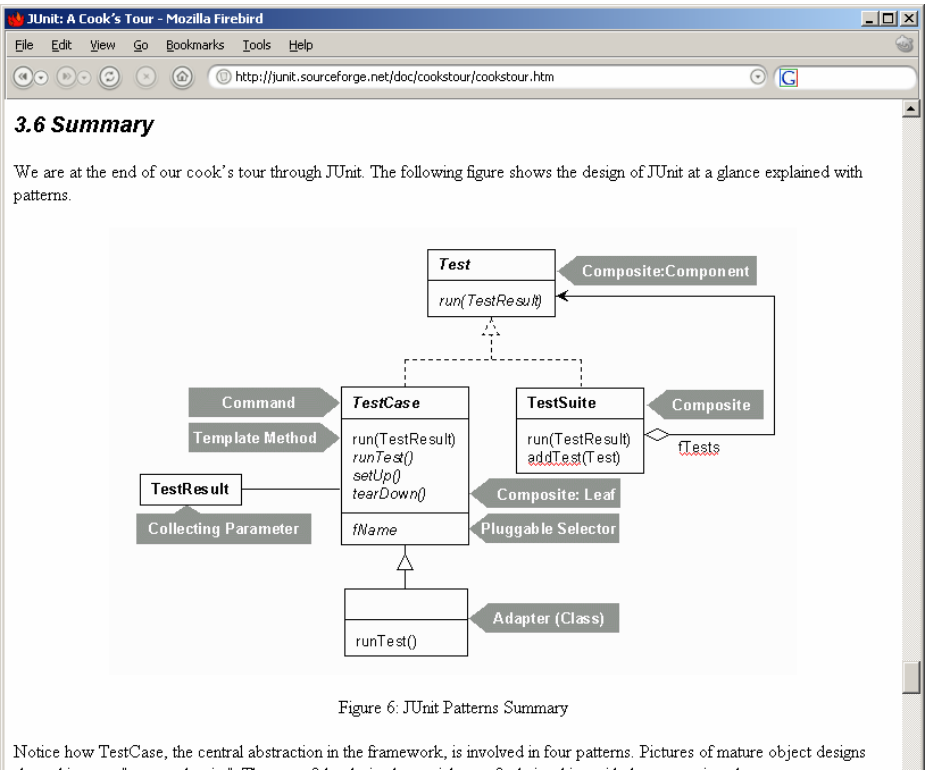


Fig. 22. Example of using design patterns to document the design of JUnit

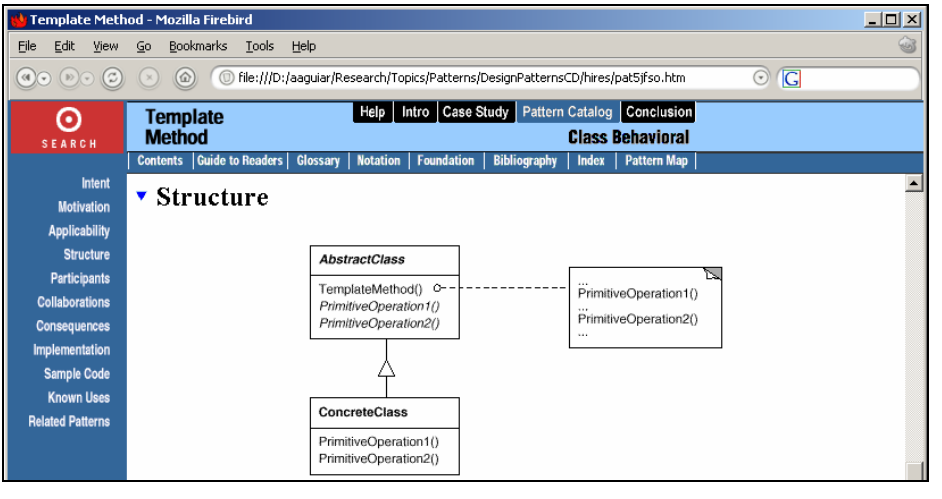


Fig. 23. Template Method Pattern

JUnit. The document "A Cook's Tour" [12], devoted to explain how JUnit was designed, includes a pattern-by-pattern tour to the design internals of JUnit. Figure 22 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 23 presents 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 23 presents an extract from the documentation relative to the Template Method pattern [27] that shows the

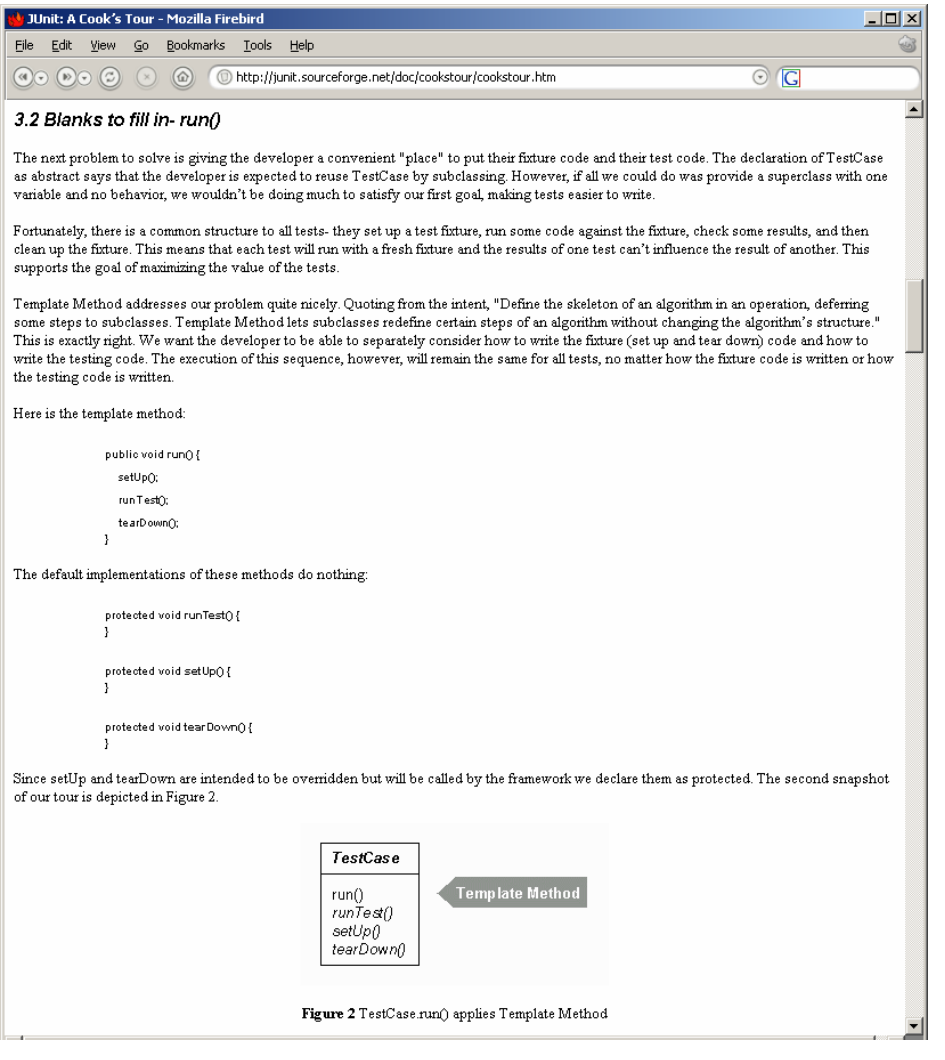


Fig. 24. Template Method being instantiated by the TestCase class

structure of the solution proposed by the pattern, the participants involved and their roles, and the consequences of instantiating the pattern.

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 25 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.

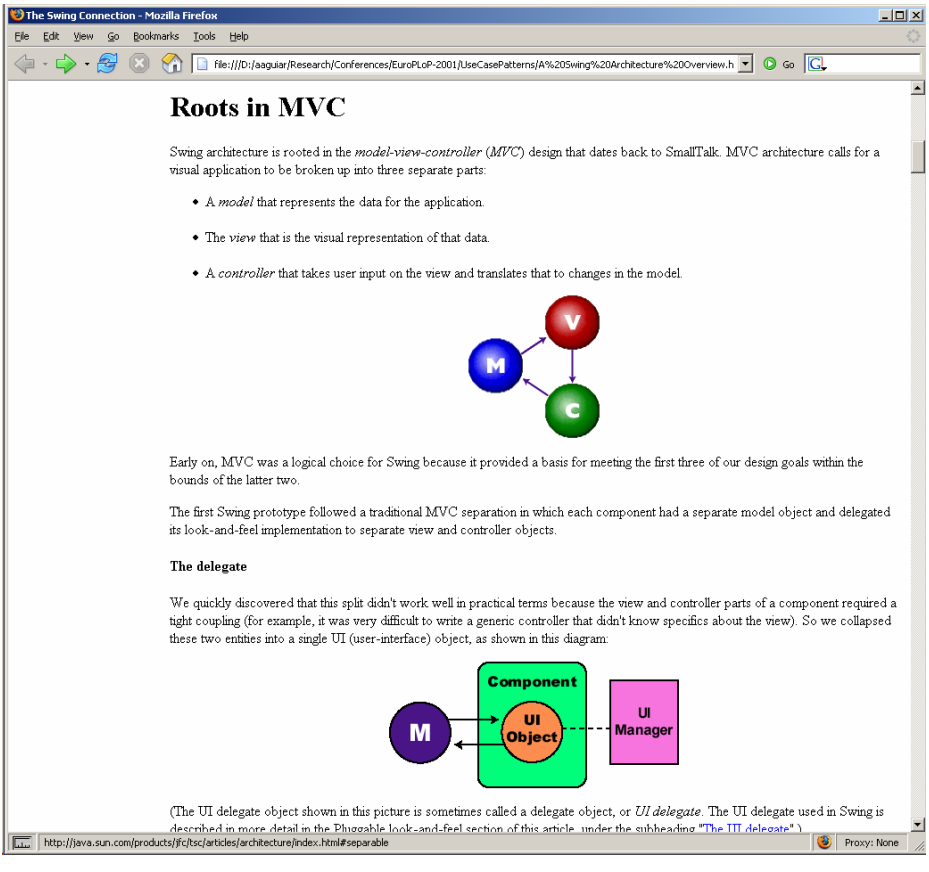


Fig. 25. An extract from "A Swing architecture overview" showing MVC and its instantiation in Swing

J2EE. The patterns underlying the design of the enterprise version of Java is documented in the core J2EE patterns catalog [7], 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 26 shows the index of all the core J2EE patterns.

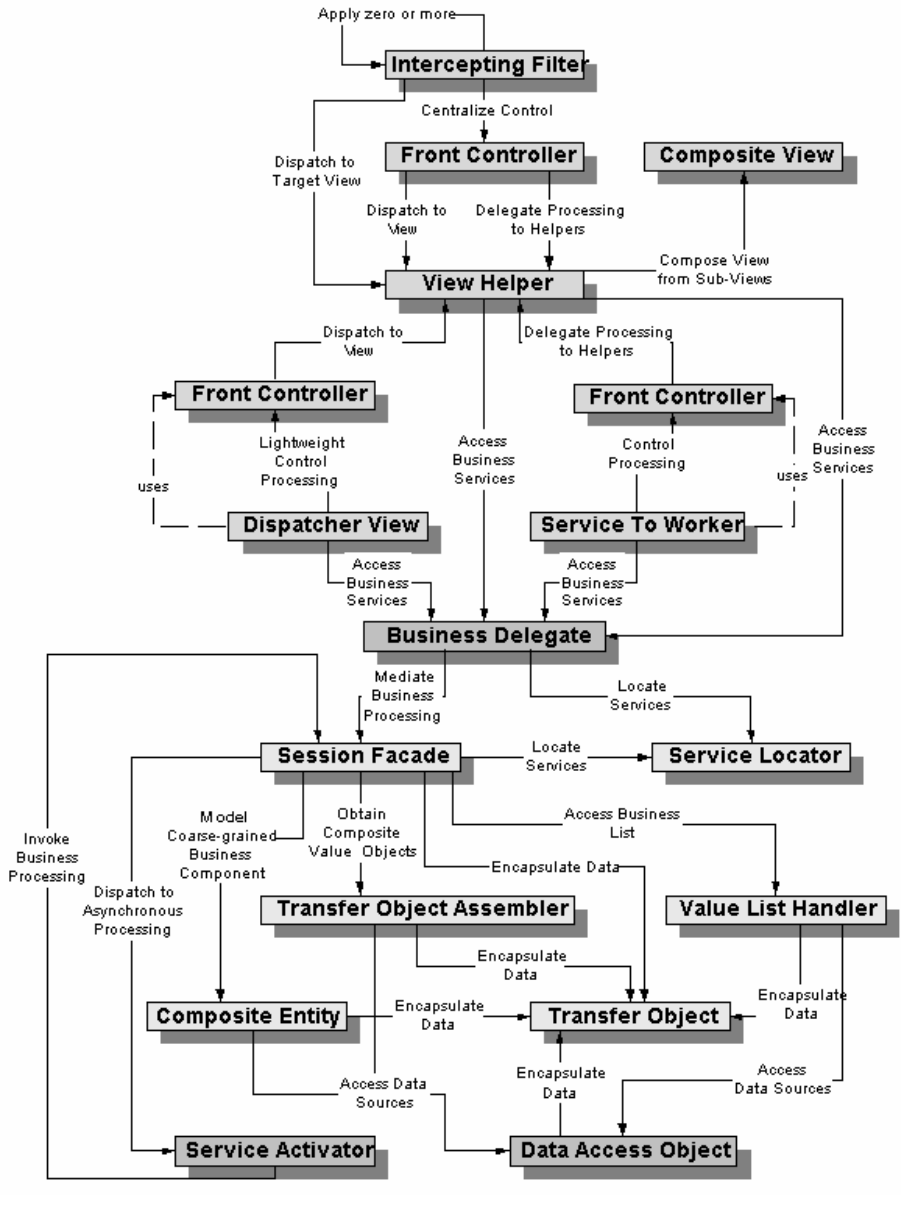


Fig. 26. Core J2EE Patterns: patterns index

.NET. Similarly to J2EE, there is a document that presents the patterns underlying Microsoft's .NET framework for enterprise applications. Figure 27 shows the documentation of the MVC pattern, which includes an example of its instantiation in .NET.

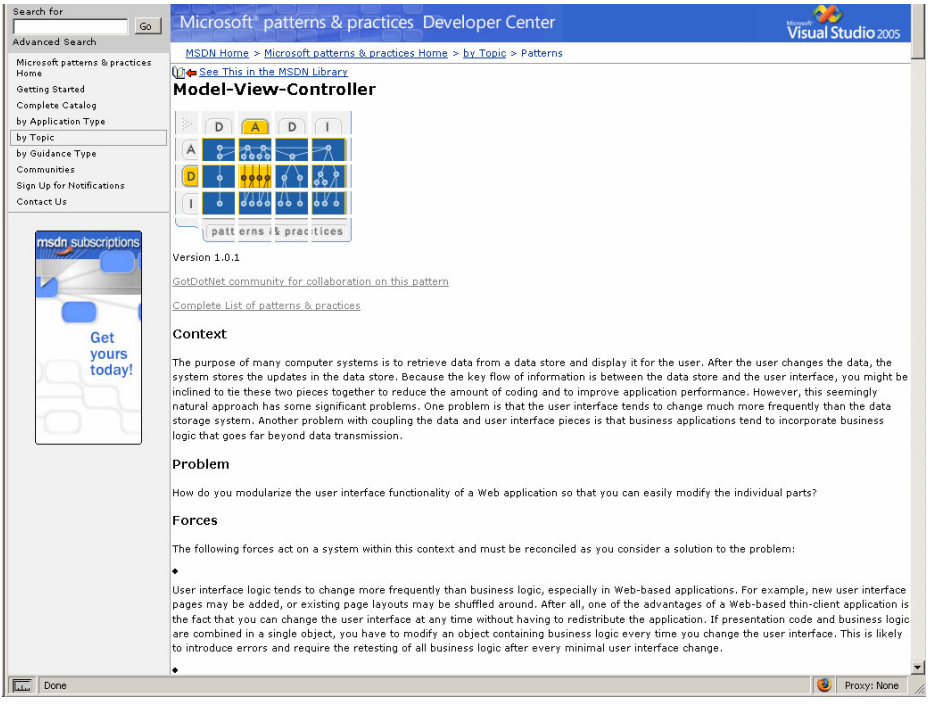


Fig. 27. .NET enterprise solution patterns" showing MVC and its instantiation in .NET

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.

9 Conclusions

Good documentation is crucial for effective framework reuse, but, unfortunately, it is often hard, costly, and tiresome, coming with many issues.

In this paper, we present a set of patterns that capture proven solutions to recurrent problems of documenting object-oriented frameworks. These patterns are part of the results of a large study based on existing literature, case studies and lessons learned, lead by the authors, aiming at helping to improve framework documentation. Other results include a documentation approach and wiki-based documentation tools [1].

The patterns here presented were selected as the most important for the framework documentation itself, here seen as an autonomous and tangible product independent of the process used to create it. Along with these patterns, the essential concepts, forces, and tradeoffs are also described, including information about the different types of documents, audiences, roles, activities, key issues and best practices, thus constituting a solid starting base for those willing to learn about how to document a framework.

The present version of these patterns are a result of several refinements in order to simplify , prescribe simple solutions that can be applied as needed, inexpensively, being easy to follow, incremental, and containing the essential to document object-oriented frameworks effectively. Additional patterns might be used to address other issues of documenting frameworks.

For those already familiar with the problematic of documenting frameworks, or reusable software, these patterns provide a survey on the topic and a consolidation. For those new to the topic, these patterns provide simple but important guidelines that enable them to document a framework in a systematic and effective way.

Acknowledgements

The authors would like to thank all those involved in the shepherding of the patterns in this document, both for having pushed forward the writing of this pattern language and for the valuable comments and feedback provided: Neil Harrison, for VikingPLoP'2005; Uwe Zdun, for EuroPLoP'2006; Rosana Teresinha Vaccare Braga and Ralph Johnson, for PLoP'2006.

We would like to thank also the participants of the writers' workshops at VikingPLoP'2005, EuroPLoP'2006, and PLoP'2006, for their comments and suggestions for improvement: Juha Parssinen, Sami Lehtonen, Eduardo Fernandez, Kevlin Henney, Klaus Marquardt, Sergiy Alpaev, Uwe Zdun, Allan Kelly, Ian Graham, Alexander Füllebornand, Martin Schmettow, Michalis Hadjisimouand, Ward Cunningham, Sachin Bammi, Philipp Bachmann, Andrew Black, Brian Foote, Maurice Rabb, Daniel Vainsencher, Anders, Mirko Raner, and Kanwardeep Ahluwalia. In addition, we are grateful to Richard Gabriel, Joseph Yoder, Mark Perry, and Maria for the feedback provided to other patterns not included in this document, but closely related, addressing process issues. Finally, we thank all those contributed with their precious reviews of this work.

References

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

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

[3] Aguiar, A., David, G.: Patterns for Documenting Frameworks – Part II. In: Proceedings of EuroPLoP 2006, Irsee, Germany (2006)

[4] Aguiar, A., David, G.: Patterns for Documenting Frameworks – Part III. In: Proceedings of PLoP 2006, Portland, Oregon, USA (2006)

[5] Aguiar, A., David, G.: Patterns for Documenting Frameworks – Process. In: Proceedings of SugarLoafPLoP 2007, Porto de Galinhas, Recife, Pernambuco, Brazil (2007)

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

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

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

[9] Beck, K., Gamma, E.: JUnit homepage (1997), `http://www.junit.org`

[10] Beck, K., Gamma, E.: JUnit: Cookbook (2003b), `http://junit.sourceforge.net/doc/cookbook/cookbook.htm`

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

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

[13] Beck, K., Johnson, R.: Patterns generate architectures, vol. 821, pp. 139–149. Springer, Berlin (1994)

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

[15] Butler, G.: A reuse case perspective on documenting frameworks (1997), `http://www.cs.concordia.ca/faculty/gregb`

[16] Butler, G., Keller, R.K., Mili, H.: A framework for framework documentation. ACM Comput. Surv. 32(1es):15 (2000)

[17] Carroll, J.M.: The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill. MIT Press, Cambridge (1990)

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

[19] Cotter, S., Potel, M.: Inside Taligent Technology. Addison-Wesley, Reading (1995)

[20] Demeyer, S., De Hondt, K., Steyaert, P.: Consistent framework documentation with computed links and framework contracts. ACM Comput. Surv., 32(1es), 34 (2000)

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

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

[23] Fayad, M.E., Johnson, R.E.: Domain-Specific Application Frameworks — Frameworks Experience by Industry. John Wiley & Sons, Chichester (2000)

[24] Fayad, M.E., Schmidt, D.C., Johnson, R.E.: Building Application Frameworks — Object-Oriented Foundations of Framework Design. John Wiley & Sons, Chichester (1999a)

[25] Fayad, M.E., Schmidt, D.C., Johnson, R.E.: Implementing Application Frameworks — Object-Oriented Frameworks at Work. John Wiley & Sons, Chichester (1999b)

[26] Froehlich, G., Hoover, H.J., Liu, L., Sorenson, P.G.: Hooking into object-oriented application frameworks. In: International Conference on Software Engineering, pp. 491–501 (1997)
[27] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns — Elements of reusable object-oriented software. Addison-Wesley, Reading (1995b)
[28] Gosling, J., Joy, B., Steele Jr., G.L.: The Java Language Specification. Addison-Wesley, Reading (1996), http://java.sun.com/docs/books/jls/
[29] Hansen, T.: Development of successful object-oriented frameworks. In: Addendum to the 1997 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (Addendum), pp. 115–119. ACM Press, New York (1997)
[30] Hargis, G.: Developing quality technical information, 2nd edn. Prentice-Hall, Englewood Cliffs (2004)
[31] IBM Corporation. Producing quality technical information. IBM Santa Teresa Laboratory (1983)
[32] Johnson, R.: Documenting frameworks using patterns. In: Paepcke, A. (ed.) OOPSLA 1992 Conference Proceedings, pp. 63–76. ACM Press, New York (1992)
[33] Johnson, R.E., Foote, B.: Designing reusable classes. Journal of Object-Oriented Programming 1(2), 22–35 (1988)
[34] Johnson, R.E., Russo, V.F.: Reusing object-oriented design. Technical Report Technical Report UIUCDCS 91-1696, University of Illinois (1991)
[35] Kirk, D.: Framework reuse: Process, problems and documentation. Technical Report EFoCS-43-2001, Department of Computer Science, University of Strathclyde, GLASGOW, UK (2001), http://www.cis.strath.ac.uk/research/efocs/
[36] Kirk, D., Roper, M., Wood, M.: Understanding object oriented frameworks: An exploratory case study. Technical Report EFoCS-42-2001, Department of Computer Science, University of Strathclyde, GLASGOW, UK (2001), http://www.cis.strath.ac.uk/research/efocs/
[37] Krasner, G.E., Pope, S.T.: A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. Journal of Object-Oriented Programming 1(3), 27–49 (1988)
[38] Lajoie, R., Keller, R.K.: Design and reuse in object-oriented frameworks: Patterns, contracts and motifs in concert, pp. 295–312. World Scientific Publishing, Singapore (1995)
[39] Meszaros, G., Doble, J.: Metapatterns: A pattern language for pattern writing. In: The 3rd Pattern Languages of Programming Conference, Monticello, Illinois (September 1996)
[40] Meusel, M., Czarnecki, K., Köpf, W.: A model for structuring user documentation of object-oriented frameworks using patterns and hypertext. In: Liu, Y., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 496–510. Springer, Heidelberg (1997)
[41] Odenthal, G., Quibeldey-Cirkel, K.: Using patterns for design and documentation. In: Akcsit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 511–529. Springer, Heidelberg (1997)
[42] Pree, W.: Design Patterns for Object-Oriented Software Development. Addison-Wesley / ACM Press (1995)
[43] Schappert, A., Sommerlad, P., Pree, W.: Automated support for software development with frameworks. In: ACM SIGSOFT Symposium on Software Reusability, pp. 123–127 (1995)
[44] Press, T.: The Power of Frameworks: for Windows and OS/2 developers. Addison-Wesley, Reading (1995)
[45] Weinand, A., Gamma, E., Marty, R.: Design and implementation of ET++, a seamless object-oriented application framework. Structured Programming 10(2) (1989)