Ademar Manuel Teixeira de Aguiar

Framework Documentation
A Minimalist Approach

Porto, Setembro de 2003
Framework Documentation
A Minimalist Approach

Dissertação apresentada para a obtenção do grau de Doutor em Engenharia Electrotécnica e de Computadores

Dissertação realizada sob a orientação científica de Doutor Gabriel David
Professor Associado do Departamento de Engenharia Electrotécnica e de Computadores

Setembro de 2003
Contact information:
Ademar Aguiar
Faculdade de Engenharia da Universidade do Porto
Departamento de Engenharia Electrotécnica e de Computadores
Rua Dr. Roberto Frias, s/n
4200-465 Porto
PORTUGAL

Tel.: +351 22 508 1400
Fax.: +351 22 508 1440
Email: ademar.aguiar@fc.up.pt
URL: http://www.fc.up.pt/~aaguiar

Ademar Aguiar
“A minimalist approach to framework documentation”
Copyright © 2002 and 2003 by Ademar Aguiar. All rights reserved.
ISBN 972-99181-0-4
To my father Ademar
and the memory of my mother Clementina

To my wife and daughters
Quisitria, Mariana and Lomar
Abstract

The minimalist approach to framework documentation proposed in this dissertation leads to a more effective and economical production and usage of framework documentation than the traditional and general-purpose techniques to document software.

Object-oriented frameworks are a powerful technique for large-scale reuse that helps developers to achieve higher productivity and shorter time-to-market through high levels of design reuse and code reuse. However, before to start using a framework successfully, users usually need to invest time on understanding its underlying architecture and design principles. Good documentation can significantly reduce the typical steep learning curve of frameworks and is, therefore, an important prerequisite for their effective reuse. The problem is that defining and writing good quality framework documentation is not easy, quick, cheap, or pleasant to do.

The approach defined in this dissertation is economical and easy-to-use to produce, organize and present high-quality framework documentation. It reuses existing documentation styles, techniques and tools and combines them in a way that follows the design principles of the minimalist instruction theory. The key idea of minimalist instruction is to minimize the obtrusiveness to the learner of training material, by offering the minimal amount of information that the learner needs to get the task done. Typically, minimalist manuals are considered easy to read and understand, contribute for shorter learning curves on how-to-use a system, and lead to a better understanding of the systems being trained.
The proposed approach covers the overall documentation process, from the creation and integration of contents till the publishing and presentation. It encompasses a documentation model, a process and a set of supporting tools. The model organizes the contents required to produce a minimalist framework manual along a virtual layered documentation space. The process proposed defines the roles, techniques and activities involved in the production of minimalist framework manuals, being generic enough to adapt to most concrete development processes. To make the approach convenient to be adopted in a wide range of software development environments, a specific set of tools is provided, which are collectively named XSDoc.

XSDoc is an extensible documentation infrastructure based on the WikiWikiWeb concept and XML technology. It provides several document templates and a simple cooperative web-based environment to produce and use minimalist framework documentation. By combining the strengths of Wiki engines, XML technology, modern open integrated development environments, and the support for major industry standards (Java, C++, UML, XML), XSDoc is a very attractive infrastructure that helps in reducing the documentation effort and improving its quality.

The minimalist approach to framework documentation as a whole (model, process and tools) represents a significant improvement over current practice considering the typical problems of framework documentation. The validation of this thesis was done with the help of two case studies and one small experiment that illustrates how the approach works in practice.
Resumo

A abordagem minimalista para documentação de frameworks proposta nesta dissertação, permite uma produção e utilização de documentação de frameworks mais económica e eficaz do que as abordagens tradicionais e genéricas para documentação de software.

As frameworks aplicacionais orientadas por objectos constituem uma poderosa técnica de reutilização de software em larga-escala. Através de elevados níveis de reutilização de código e desenho, as frameworks permitem aumentar a produtividade, reduzir tempos de desenvolvimento e aumentar a qualidade das aplicações. Contudo, antes de se conseguir utilizar uma framework de forma eficaz, é normalmente necessário dedicar tempo e esforço considerável na aprendizagem e compreensão dos detalhes essenciais da sua arquitectura e desenho. Uma boa documentação ajuda significativamente a reduzir a longa curva de aprendizagem das frameworks e constitui por isso um importante pré-requisito para a sua reutilização eficaz. Mas definir e produzir boa documentação para uma framework é quase sempre bastante difícil, desagradável e dispendioso.

Nesta dissertação é proposta uma abordagem económica e fácil de usar para apoiar as actividades de produção, organização e apresentação de documentação de qualidade para frameworks. A abordagem reutiliza estilos de documentos, técnicas e ferramentas de documentação já existentes, e combina-os num todo coerente seguindo os princípios da teoria de instrução minimalista. A ideia principal da instrução minimalista é a de minimizar os obstáculos que perturbam a estratégia de aprendizagem espontaneamente.
adoptada pelo leitor, tentando fornecer a quantidade mínima de informação que o leitor necessita para conseguir aprender a realizar a tarefa que tem em mãos. De uma forma geral, os manuais de instrução minimalista são considerados fáceis de ler e entender, pelo que contribuem tanto para uma redução dos tempos de aprendizagem na utilização dum sistema, como para uma melhor compreensão do sistema em aprendizagem.

A abordagem proposta cobre todo o processo de documentação, desde as fases iniciais de criação e integração de conteúdos até às fases de publicação e apresentação. É composta por um modelo de documentação, um processo e um conjunto de ferramentas de suporte. O modelo define os diversos conteúdos necessários à produção de manuais minimalistas para frameworks e organiza-os ao longo de um espaço virtual de documentação definido em camadas. O processo define os papéis, técnicas e atividades envolvidas na produção de manuais de instrução minimalista, sendo bastante genérico e simples por forma a se adaptar a diversos processos de desenvolvimento concretos. Por forma a tornar a abordagem fácil de adoptar numa vasta gama de ambientes de desenvolvimento, a abordagem fornece também um conjunto específico de ferramentas, coletivamente denominadas de XSDoc.

XSDoc é uma infraestrutura extensível baseada no conceito WikiWikiWeb e em tecnologia XML. XSDoc disponibiliza diversos documentos-tipo e um ambiente cooperativo baseado na web especificamente concebido para a produção e utilização de documentação minimalista para frameworks. A combinação das vantagens dos servidores Wiki, tecnologia XML, ambientes de desenvolvimento integrado abertos e suporte para os standards mais importantes em termos de desenvolvimento de frameworks (Java, C++, UML, XML), tornam-na uma infraestrutura bastante atractiva que permite reduzir de forma considerável o esforço de documentação e melhorar a sua qualidade.

A abordagem minimalista para documentação de frameworks, vista como um todo (modelo, processo e ferramentas), representa um avanço significativo relativamente à prática corrente, tendo em consideração os problemas associados à documentação de frameworks. A validação desta tese foi realizada com a ajuda de dois casos de estudo e uma pequena experiência que permitem ilustrar a aplicabilidade da abordagem e as suas qualidades.
Resumé

L'approche minimaliste à la documentation de frameworks proposée dans cette dissertation, rend la production et l'utilisation de documentation de frameworks plus efficace et plus économique que les approches traditionnelles et généralistes utilisées dans la documentation de logiciels.

Les frameworks orientés-objets sont une technique puissante facilitant la réutilisation à grande échelle. En effet, ils réduisent le "time-to-market" en améliorant la productivité des développeurs grâce à la réutilisation de conceptions et de code. Cependant avant de tirer pleinement profit d'un framework, les utilisateurs doivent passer du temps à comprendre l'architecture et la conception sous-jacente au framework. Une documentation de qualité peut de manière significative réduire le long et typique temps d'apprentissage d'un framework et devient donc un prérequis important à son utilisation efficace. Cependant la définition et l'écriture d'une telle documentation n'est ni facile, ni rapide, ni bon marché, et encore moins plaisante à faire.

L'approche définie dans cette dissertation est une approche économique et facile à utiliser pour produire, organiser et présenter une documentation de frameworks de très bonne qualité. Elle réutilise des modèles, des techniques et des outils existants de documentation et les combine en suivant les principes de conceptions de la théorie d'instruction minimaliste. L'idée principale de cette théorie est de minimiser l'obstacle que représente le support de cours à l'apprenti en lui proposant la quantité minimale d'informations dont il aura besoin pour mener à bien sa tâche. Typiquement,
les manuels minimalistes sont considérés comme faciles à lire et à comprendre, et contribuent à réduire le temps d'apprentissage du "comment utiliser un système" et à mieux comprendre le système étudié.

L'approche proposée couvre le processus global de documentation: de la création et de l'intégration de contenus jusqu'à leur publication et leur présentation. Elle inclut un modèle de documentation, un processus ainsi qu'un ensemble d'outils associés. Le modèle organise les contenus nécessaires à la production d'un manuel minimaliste de framework dans un espace virtuel et organisé de documentation. Le processus est très générique et définit seulement les rôles, les techniques, et les activités impliquées dans la production de manuels minimalistes de frameworks. Pour faciliter l'adoption de cette approche par un grand nombre d'environnements de développement, un ensemble d'outils nommé XSDoc a été créé.

XSDoc est une infrastructure extensible de documentation basée sur le concept de WikiWikiWeb et sur la technologie XML. Il fournit plusieurs modèles de documents et un simple environnement coopératif web pour la production de documentation minimaliste de frameworks. En combinant la force des Wikis, des technologies XML, et des environnements de développement avec d'importants standards industriels (Java, C++, UML, XML), XSDoc fournit une infrastructure très attrayante réduisant l'effort de production de documentation tout en améliorant sa qualité.

Dans sa globalité, l'approche minimaliste pour la documentation de frameworks (modèle, processus et outils) représente une amélioration significative par rapports aux pratiques actuelles de documentation de frameworks. La validation de cette thèse a été faite grâce à deux études de cas et une petite expérience qui montrent dans la pratique le fonctionnement de l'approche.
# Table of Contents

1. Introduction
   1.1. The Power of Object-Oriented Frameworks ........................................... 1
   1.2. Problems with Framework Reuse ............................................................ 2
   1.3. The Problem of Documenting Frameworks ................................................ 3
   1.4. Research Goals
      1.4.1. Primary Goal .................................................................................... 4
      1.4.2. Secondary Goal ................................................................................ 5
   1.5. Research Strategy .................................................................................... 6
      1.5.1. Software Engineering Research Methods ........................................... 7
      1.5.2. Research Model and Phases .............................................................. 8
   1.6. Thesis Statement – Preliminary Version ................................................. 9
   1.7. Main Results
      1.7.1. Documentation Model ....................................................................... 10
      1.7.2. Documentation Process .................................................................... 11
      1.7.3. Documentation Tools ....................................................................... 12
   1.8. Future Work ......................................................................................... 13
   1.9. How to Read this Dissertation ............................................................... 14

2. Framework Fundamentals
   2.1. What Is a Framework? ............................................................................ 15
   2.2. Frameworks and Reuse
      2.2.1. Reuse Techniques ........................................................................... 16
      2.2.2. How Object-Oriented Leverages Software Reuse? .......................... 17
      2.2.3. The Power of Frameworks ............................................................... 18
   2.3. Object-Oriented Software Architecture
      2.3.1. What is Software Architecture? ....................................................... 19
      2.3.2. Architectural Levels ....................................................................... 20
   2.4. Fundamental Concepts
      2.4.1. Object-Oriented Concepts ............................................................... 21
      2.4.2. Template and Hook Methods ............................................................ 22
      2.4.3. The Flow of Control in Framework-based Applications ................. 23
      2.4.4. Classifying Frameworks ................................................................. 24
   2.5. History of Frameworks
      2.5.1. Early frameworks ........................................................................... 25
      2.5.2. GUI frameworks ............................................................................. 26
      2.5.3. Taligent frameworks ....................................................................... 27
      2.5.4. Frameworks today (2000s) .............................................................. 28
3. Building, Using and Documenting Frameworks ........................................... 43
   3.1. Framework-Based Application Development ........................................ 44
       3.1.1. Artifacts .................................................................................. 45
       3.1.2. Activities ............................................................................... 46
       3.1.3. Roles ...................................................................................... 46
   3.2. Building Frameworks ........................................................................... 46
       3.2.1. Typical Activities of Building Frameworks .................................. 47
       3.2.2. Approaches for Building Frameworks ........................................ 50
       3.2.3. Key Issues of Building Frameworks .......................................... 54
   3.3. Using Frameworks ............................................................................... 56
       3.3.1. Typical Activities of Using Frameworks ..................................... 56
       3.3.2. Techniques for Using Frameworks .............................................. 58
       3.3.3. Key Issues of Using Frameworks ............................................... 59
       3.3.4. Main Benefits of Using Frameworks .......................................... 60
   3.4. Documenting Frameworks ................................................................... 63
       3.4.1. Typical Activities of Framework Documentation ............................ 64
       3.4.2. Classifying Framework Documentation ....................................... 66
       3.4.3. Framework Overview .................................................................. 74
       3.4.4. Cookbooks and recipes .............................................................. 75
       3.4.5. Patterns .................................................................................... 77
       3.4.6. Architectural models ................................................................... 82
       3.4.7. Contracts .................................................................................. 85
       3.4.8. Reference Manual ...................................................................... 86
       3.4.9. Example Applications .................................................................. 86
       3.4.10. Software Documentation Techniques ....................................... 87
       3.4.11. Key Issues of Documenting Frameworks .................................... 92
   3.5. Maximizing Benefits of Frameworks ................................................... 101
       3.5.1. Aspects to Consider When Building Frameworks ........................... 101
       3.5.2. Characteristics of Successful Frameworks .................................... 102
       3.5.3. Examples of Successful and Unsuccessful Frameworks ................. 103

4. Minimalist Documentation ........................................................................... 105
   4.1. Minimalist Documentation .................................................................... 106
       4.1.1. More to Do ................................................................................. 106
       4.1.2. Less to Read .............................................................................. 106
       4.1.3. Help with Errors ....................................................................... 106
   4.2. Learning systems .................................................................................. 106
       4.2.1. Being Overwhelmed ................................................................. 107
       4.2.2. Following Directions ............................................................... 107
       4.2.3. Interacting with the system ....................................................... 107
       4.2.4. Recovering from Errors ........................................................... 107
   4.3. Minimalist Instruction .......................................................................... 108
       4.3.1. The Weakness of Systems Approach .......................................... 108
       4.3.2. Design Principles .................................................................... 109
       4.3.3. The Limits of Learning ............................................................. 112
   4.4. The Minimal Manual ............................................................................ 113
       4.4.1. Design Analysis ....................................................................... 113
       4.4.2. Subskill Evaluation .................................................................. 114
       4.4.3. Using the Minimal Manual ....................................................... 115
       4.4.4. Evaluating the Minimal Manual .............................................. 115
4.4.5. Inference and Rehearsal.......................................................... 115
4.4.6. Developing Minimal Manuals............................................. 115
4.5. Task Intelligence .................................................................... 116
4.6. Developing Minimalist Technology......................................... 117
   4.6.1. Science and Design....................................................... 117
   4.6.2. Technology Transaction............................................... 118
   4.6.3. Minimalist Know-How................................................. 118
4.7. Principles for Designing Minimalist Documentation.............. 118

5. Research Problem .................................................................... 121
   5.1. Research Focus................................................................. 121
   5.2. Related Work .................................................................... 123
       5.2.1. Software Documentation........................................... 124
       5.2.2. Framework Documentation....................................... 130
       5.2.3. Minimalist Documentation....................................... 133
       5.2.4. Summary .................................................................. 135
   5.3. Research Issues............................................................... 136
   5.4. Research Goals.................................................................. 137
       5.4.1. Primary Goal............................................................. 137
       5.4.2. Secondary Goal......................................................... 138
   5.5. Thesis Statement................................................................ 138
       5.5.1. Preliminary Version................................................... 138
       5.5.2. What is the scope of “production” and “usage”?........ 139
       5.5.3. What is meant by “easier”?...................................... 139
       5.5.4. Which are the “traditional approaches”?.................. 140
       5.5.5. “Who” does benefit from these improvements?.... 141
       5.5.6. Final version............................................................. 141
   5.6. Validation Methodology...................................................... 142

   6.1. Requirements for the approach......................................... 146
       6.1.1. Helping on “documenting-while-developing”......... 146
       6.1.2. When producing documentation............................. 147
       6.1.3. When using documentation..................................... 148
       6.1.4. Functional requirements.......................................... 149
       6.1.5. Non-functional requirements................................. 152
       6.1.6. List of requirements ............................................... 154
   6.2. Documentation Model...................................................... 155
       6.2.1. Resources................................................................. 156
       6.2.2. Contents ................................................................ 157
       6.2.3. Concrete contents................................................... 158
       6.2.4. Templates................................................................. 159
       6.2.5. Multidimensional structure..................................... 160
       6.2.6. The (meta)model....................................................... 163
       6.2.7. Minimalist framework manual.............................. 165
   6.3. Documentation Process...................................................... 170
       6.3.1. Rationale: “agile documentation”............................ 170
       6.3.2. Goals....................................................................... 171
       6.3.3. Roles ...................................................................... 172
       6.3.4. Activities............................................................... 173
6.3.5. “Good practices” ................................................................. 173
6.3.6. Guidelines ................................................................. 175

6.4. Tools and Utilities — XSDoc ......................................................... 178
6.4.1. The XSDoc documentation infrastructure ......................... 179
6.4.2. XSDocWiki ......................................................................... 180
6.4.3. XML converters and presentation processors .................... 183
6.4.4. Integration of multiple sources of contents ....................... 183
6.4.5. Wiki-centric functional integration .................................... 185
6.4.6. Integration in development environments ......................... 186
6.4.7. Side-by-side edition of all contents .................................... 186
6.4.8. Extensibility .................................................................. 187
6.4.9. Using XSDoc .................................................................. 188
6.4.10. XSDoc: plan for the future .............................................. 189

7. Case Studies and Experiment ......................................................... 193
7.1. Case Study — JUnit .................................................................. 194
7.1.1. Model definition ................................................................. 194
7.1.2. Semantic consistency problem .......................................... 194
7.1.3. Wiki-based software documentation .................................... 195
7.1.4. Main results .................................................................. 195
7.2. Case Study — COOLFluiD .......................................................... 195
7.3. Controlled Experiment with JUnit ............................................. 196
7.3.1. Designing the experiment .................................................... 196
7.3.2. Selecting the subjects ......................................................... 203
7.3.3. Testing the experiment ....................................................... 204
7.3.4. Running the experiment ...................................................... 205
7.3.5. Processing the results ......................................................... 205
7.3.6. Discussion ................................................................ 208

8. Thesis Validation ........................................................................... 209
8.1. Thesis review .......................................................................... 210
8.1.1. Baseline for comparing approaches ..................................... 210
8.1.2. Baseline for comparing documentation ................................ 210
8.1.3. Thesis decomposition .......................................................... 211
8.1.4. Organizing the sub-theses for validation ............................. 212
8.2. Assessing the documentation model ........................................ 213
8.3. Assessing the documentation process ....................................... 214
8.4. Assessing the documentation tools ........................................... 215
8.4.1. Flexible integration of contents ............................................. 215
8.4.2. Wiki: effective, attractive, and easy-to-use ......................... 216
8.4.3. Easy integration in development environments .................. 216
8.4.4. Interoperability ................................................................. 217
8.4.5. Effective guidance through templates and guidelines .......... 217
8.4.6. Widely available ............................................................... 218
8.4.7. Main benefits of using XSDoc ............................................. 218
8.5. Consolidation ........................................................................... 220
8.5.1. Summary of arguments ....................................................... 220
8.5.2. Assessing the maturity of the minimalist framework documentation .... 221
8.6. Conclusions ............................................................................ 222
9. Conclusions ........................................................................................................... 225

9.1. Summary of Contributions ............................................................................. 226

9.2. Future Research ............................................................................................... 226
List of Figures

1.1. Research model: phases and respective research methods. ...................................... 9
1.2. Design elements of object-oriented architectures .................................................. 26
2.2. The class TestCase. ............................................................................................... 31
2.3. The Unification meta-pattern attached to the testing framework .......................... 33
2.4. Unification, connection, and recursive meta-patterns. ...................................... 34
2.5. Evolution of control flow in programs. ................................................................. 35
2.6. Illustration of control flow in the framework example ......................................... 37
2.7. Classification of frameworks based on the extension technique ......................... 38
3.1. Traditional versus framework-based application development. ....................... 44
3.2. Activities, artifacts and roles of framework-based application development. .... 45
3.3. Typical activities of framework development. .................................................... 47
3.4. Inheritance-based and composition-based hot-spot subsystems. ....................... 48
3.5. Evolving Frameworks pattern language. .............................................................. 52
3.6. Key activities of application development using frameworks. ......................... 56
3.7. Typical activities of framework development .................................................... 65
3.8. Relating documentation audiences with categories of reuse. ........................... 72
3.9. Example of the framework overview delivered with JUnit. ............................... 75
3.10. Example of a recipe for writing a simple test with JUnit. ................................. 76
3.11. The Template Method. ....................................................................................... 78
3.12. Example of using design patterns to document the design of JUnit ................ 79
3.13. Template Method is instantiated by the TestCase class. ................................ 80
3.14. Example of attaching a meta-pattern to a hot spot from JUnit ....................... 82
3.15. UML-F representation for the Composite pattern instance in JUnit. ............. 84
3.16. Example of a framework specification using Helm contracts. ......................... 85
3.17. MoneyTest: an example provided with JUnit. ............................................... 87
3.18. Literate programming processes: tangling and weaving. ................................ 90
3.19. Quality dimensions for technical information. ................................................ 94
5.1. Enumeration of framework documentation issues. .......................................... 122
5.2. Documentation scheme for object-oriented systems. ........................................ 129
5.3. Virtual subsystem stratifications. ...................................................................... 130
5.4. Prioritization of framework documentation issues for research ....................... 136
5.5. Separation of "production" and "usage" documentation activities .................... 139
5.6. Classification of issues based on the activities they are related. ....................... 140
5.7. Sets of sub-these to validate (non-blank central cells). ................................ 143
6.1. Use cases for a minimalist framework documentation environment ................ 149
6.2. Use cases for the browsing facilities. ................................................................. 150
6.3. Use cases for the edition facilities. .................................................................. 151
6.4. Use cases for the configuration facilities. ......................................................... 152
6.5. Summary of requirements for the approach. .................................................... 154
6.6. Parts of the model: resources, contents, templates, structure ....................... 155
6.7. Model concepts for physical resources ....................................... 156
6.8. Model concepts for contents .................................................. 157
6.9. Modeling concrete contents ................................................... 158
6.10. Model concepts for document templates ................................ 160
6.11. Model concepts for structuring the contents ............................ 161
6.12. Configuration of layers and filters for an uni-dimensional structure. 162
6.13. Example of a configuration of layers ...................................... 163
6.14. The complete documentation (meta)model ................................ 164
6.15. Contents and structure of a minimalist framework manual ............. 166
6.16. Contents and structure of patterns and pattern instances ............... 167
6.17. Configuration of layers for framework documentation .................... 168
6.18. Quality aspects of technical documentation ............................... 176
6.19. Typical authors for the different types of documents ..................... 177
6.20. XSDoc components and their interconnections ............................ 180
6.21. Inlining Java source code in a Wiki document ........................... 184
6.23. Linking Java source code in a Wiki document ............................ 185
6.24. Command pattern instance ................................................... 185
6.25. A snapshot of the XSDoc plugin for the Eclipse IDE ...................... 187
7.1. Form of the experiment describing the tasks ................................ 199
7.2. Task completion times registered during the experiment ................. 206
7.3. Times aggregated by task category and subjects group ................... 207
7.4. Completion times for setup tasks ............................................. 207
7.5. Completion times for first-usage tasks ...................................... 207
7.6. Percentile reductions of the task completion times ....................... 208
8.1. Impact of the minimalist approach components on the issues ............ 212
Preface

My first memories of designing, building, and evolving "things" date back to my early childhood, playing with LEGO's, geometric drawings, musical themes, toys, etc. Many years after, I discovered in software a new and huge space for playing again with design, at first with algorithms and data structures, and later, with objects, the smallest "LEGO's bricks" of contemporary software.

I like design, specially good design. There are several ways of defining it. In Webster's dictionary, design is defined as the act of: "working out the form of something", "sketching a plan for something", "creating something in the mind", or "doing something for a specific role or purpose or effect". But what is a "good design"? Christopher Alexander explains this in a very detailed way using his theory of centers described in his latest work entitled "Nature of Order". In a few words, I define a good design as "a beautiful arrangement of design elements serving the purpose in mind", whatever these elements are. But, again, how to define "beautiful" in this context? Yes, this is a long story...

So, after designing small object systems, using C++, Eiffel and OMT, in late 1980s, the experience gained with the design of applications for NeXTstep, using its Objective-C and component libraries, created new design challenges, in scale and complexity, and this compelled me to "dive in" object-oriented software architecture, particularly into object-oriented frameworks and design patterns.
Object-oriented frameworks conquered me very fast: they are a powerful reuse technique and are the kind of object-oriented product by far most complex and difficult to design. The big challenge of framework design is to find the perfect balance between generality and usefulness, which is very difficult to fine-tune, being best achieved by iteration.

Unfortunately, many projects fail to fully exploit the reuse power of frameworks, due to many reasons, being poor quality of design and implementation just two of them. Although good design and implementation are necessary pre-requisites for the successful reuse of frameworks, they are not sufficient. Good documentation is also a crucial pre-requisite, although its importance is not always recognized.

This dissertation addresses pertinent technical problems of framework documentation. My goal is not only to make a small, yet significant, contribution to the field, but also to provide software developers with support for producing good quality framework documentation. The support provided takes the form of a new documenting approach: a model, a process and a set of supporting tools.

This research work started in autumn 1998. It was carried out both in half-time (Sep.1998–Mar.2000, Sep.2001–Sep.2003) and in full-time (Apr.2000–Ago.2001). The research started during my collaboration in SIMAT project, at INESC Porto, a project that provided me a good practical experience with the issues of software documentation. After the SIMAT project, I had small collaborations in three other research projects: SIMATWARE, a project of an industrial application framework for the domain of urban planning and management, at ParadigmaXis; and two “METAMEDIA” projects that investigated the use of metadata to preserve multimedia and software components. All these projects have positively influenced this work.

Along the years, the work received great help, support and encouragement from many people, to whom I am deeply indebted. The most important person for the success of this work was my supervisor, colleague and friend Gabriel David, who cleverly guided me along the way, and supported me all the time. I am highly grateful to him, not only for the feedback, suggestions, and time we spent working together, but also for the way he conducted this supervision. Thank you Gabriel. A special thanks also to my friend José Bonnet for his feedback on parts of this work, and for the interesting conversations about this and many other professional things while running along Douro riverside.

I also want to thank my university colleagues Raul Vidal, João Correia Lopes, João Pascoal Faria, Cristina Ribeiro, and António Pina Martins for their encouragement, and for having generously coordinated with me some faculty service during the more intensive periods.
Special thanks to Aurélio Pires, Alexandre Sousa, Lígia Silva, Artur Rocha, and the rest of the SIMAT project team at INESC Porto, for the pleasant atmosphere of cooperation and learning we have established. In the same vein, I want to thank João Falcão Cunha, Mário Santos, and Jorge Simões, who, with me and José Bonnet, formed the TEIA group at INESC, where I have started studying object-orientation, the most remote reason for my present interest on frameworks.

I am also grateful to the institutions and software companies that have cooperated with me for this work: Mário Jorge Leitão, from INESC Porto, and José Silva Matos, from DEEC, for all the support provided; ParadigmaXis, for the invitation to collaborate on the simatWare framework project; Thomas Vestdam and Kurt Nørmark, from the Department of Computer Science of Aalborg University, for the scientific collaboration we had first in Aalborg and then in Porto related with software documentation; José Carlos Ramalho and Pedro Rangel Henriques, from the Department of Informatics of the Universidade do Minho, for their initial help with some tricky details of XML technology; PT Inovação Porto, for the experiment to validate my work with their developers; Tiago Quintino, from Von Karman Institute for Fluid Dynamics, for the COOLFluid case study; LEIC students from FEUP, for having participated in an experiment, without knowing about it.

During the years of this research, I had the opportunity to know and exchange ideas with many researchers, and some of them, for one reason or another, have contributed or influenced my work. Thanks to all them: Neil Harrison, Jim Coplien, Richard Gabriel, John Goodsen, Frank Buschmann, Peter Sommerlad, Jutta Eckstein, Jorge Ortega-Arjona, Joe Bergin, Alistair Cockburn, Jens Coldewey, Andreas Rüping, Paul Dyson, Kent Beck, Martin Fowler, Ralph Johnson, Kevlin Henney, Joshua Kerievsky, Dennis Kafura, Thomas Vestdam, and Pascal Rapicault.

Thanks also to Pat Metheny, J. S. Bach, and Madredeus for their wonderful music.

Thanks to my parents, Ademar and Clementina, my aunt Fernanda, and my sister Alice, for all the tender and love you gave me during all your life.

To my lovely wife Quitéria and sweet daughters Mariana and Leonor, thank you for pushing me forward and giving me so much love and joy.
This research work was financially supported by the Department of Electrical Engineering and Computers (DEEC) from the Faculty of Engineering of University of Porto (FEUP), and the Fundação da Ciência e Tecnologia (FCT) projects “Meta-informação para Arquivos Multimedia” (PRAXIS/EEI/13208/1998) and “Metadata for the Preservation and Retrieval of Multimedia Components” (POSI 34903/SRI/2000).
Object-oriented frameworks are a powerful technique for large-scale reuse capable to deliver high levels of design and code reuse. Frameworks promise higher development productivity, shorter time-to-market, and higher quality, but these benefits are only gained over time and require up-front investments. Before starting to use a framework successfully, users usually need to spend a lot of effort in understanding its underlying architecture and design principles, and learning how to use it. The steep learning curve typical of frameworks can be significantly reduced with good framework documentation.

This dissertation focuses on the problem of documenting frameworks, one of the several technical, organizational, and managerial issues that must be well managed in order to employ frameworks effectively.

This chapter gives a general introduction to the overall dissertation. It starts with a brief review of frameworks and some of the most relevant issues related to framework reuse. The chapter presents several key problems related with producing and using framework documentation and then states the research questions that have driven the work presented here. The chapter concludes with an overview of how and where these questions are answered.
1.1 The Power of Object-Oriented Frameworks

Software productivity has increased significantly in the last three decades [Yourdon, 1992] but the gap between the demands placed on the software industry and what it can deliver is still large. The key factor to leverage developer’s productivity is the number of software source instructions that have to be written to deliver a given functionality: instead of searching for ways of writing code faster, we have to look for ways of writing less of it [Boehm, 1987].

The intensive research done in software engineering left software reuse as the (only) realistic approach capable of bringing out the gains of productivity and quality that the software industry needs [Mili et al., 1995]. Automatic programming [Rich and Waters, 1988], the other alternative whereby a computer system is capable of producing executable code based on informal, incomplete, and incoherent user requirements is still very far from feasible, if ever possible.

Simply stated, 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 [Krueger, 1992]. Firmly in the middle of this range, we find object-oriented frameworks, a powerful technique for large-scale reuse.

A framework can be defined as a reusable, semi-defined application that can be specialized to produce custom applications [Johnson and Foote, 1988]. Through design and code reuse, frameworks help developers to achieve higher productivity, shorter time-to-market and improved compatibility and consistency [Taligent Press, 1994]. When combined with components and patterns, frameworks are considered the most promising current technology supporting large-scale reuse [Fayad and Schmidt, 1997; Fayad et al., 1999].

However, as other benefits from reuse in general, the benefits from frameworks are only gained over time and require up-front investments. Fichman and Kemerer noted that the steep learning curve it takes until developers can work productively is a key challenge for the adoption of object technology [Fichman and Kemerer, 1997]. When compared to the adoption of basic object-oriented software development, frameworks require an even higher up-front learning investment. Learning a framework is harder than learning a regular class library because the classes in a framework are designed to work together, are more abstract and their behavior is usually not completely implemented.
1.2 Problems with Framework Reuse

Whilst practical evidence does suggest that framework usage can significantly increase reusability and decrease development effort [Moser and Nierstrasz, 1996], it can be a non-trivial undertaking due to the several technical, organizational, and managerial issues that must be addressed and resolved in the development process.

The issues spread across many activities of framework development and framework usage, which, if not well managed, can compromise framework reuse benefits – the primary reason for building and using frameworks [Fayad and Schmidt, 1997; Bosch et al., 1999; Fichman and Kemeler, 1997; Mattsson and Bosch, 1999]. Some of these issues are briefly summarized below:

- **Development Issues** include scoping the framework domain, lack of appropriate development methods, testing frameworks, producing high-quality framework documentation, and selecting adequate strategies for framework maintenance and evolution.

- **Usage Issues** include evaluating framework applicability, learning and understanding a framework, lack of mature processes for developing applications based on frameworks, and validating and debugging applications.

From the list above, the issues of understanding and learning are particularly important for the effectiveness of reuse. Before reusing a piece of software, one first has to invest time on understanding and learning how to use it. The more complex a reusable piece of software is, the more difficult and time consuming will probably be these tasks. As one of the most complex kinds of object-oriented products, frameworks are thus inherently difficult to understand, specially by first-time users. Grady Booch stated that “the most profoundly elegant framework will never be reused unless the cost of understanding it and then using its abstractions is lower than the programmer’s perceived cost of writing them from scratch” [Booch, 1994].

Understandability is thus a very important prerequisite for the effective reusability of frameworks. The easier to understand a framework, the easier to reuse it. In the literature [Fayad and Schmidt, 1997], we can find different alternatives claimed to be able to improve framework understandability:

- by refining framework’s *internal design*;

- by using *methods* that can ensure a successful development and usage of frameworks;

- by adhering to *standards* for framework development, adaptation, and
integration (non-existent yet);

- and by producing comprehensible framework documentation.

The focus of this work is precisely on this last topic of framework documentation. High-quality framework documentation, including not only the low-level method-oriented documentation usually available in the vast majority of frameworks, but also documentation that captures the strategic roles and collaborations among framework components, is used in this research as a means to improve the understandability and usability of frameworks, and therefore, as a means to increase the effectiveness of framework reuse, the fundamental goal of this work.

1.3 The Problem of Documenting Frameworks

As stated before, good documentation is crucial for the success of frameworks. Without a clear, complete and precise documentation describing how to use the framework, how it is designed, and how it works, the framework will be particularly hard to understand and nearly impossible to use by software engineers not initially involved in its design.

Good documentation and good training material provide a significant help in the process of learning a new framework, by guiding users on the customization process and making explicit the framework design principles and details, thus contributing to a framework easier to reuse.

But simple, complete and easy-to-use documentation can be seen as the acid test for the quality of a framework. It is not easy to write “good documentation” for a poorly designed framework. So, good design is a necessary, but not sufficient, prerequisite for good documentation.

Many approaches to document frameworks have been suggested in the literature, and among these, several proved to be effective in reducing the learning curve of frameworks, namely the cookbook approach [Krasner and Pope, 1988], the patterns approach [Johnson, 1992], [Beck and Johnson, 1994], and the meta-patterns approach [Pree, 1995].

However, it is still hard, costly and tiresome to define and write good quality documentation for a framework. Documenting frameworks is, at least, an order of magnitude more difficult than documenting object-oriented applications or class libraries, because it must cover not only a single concrete product (application or class), but, instead, a tool (a framework) able to produce a family of similar concrete products. Additionally to all the aspects that are relevant to object-oriented applications and class libraries, the documentation of a framework must meet the needs of different
audiences and to encompass several purposes.

To be complete, the overall documentation of a framework usually combines a lot of information that must be produced, organized and maintained consistent. The documentation must describe the application domain covered by the framework, its purpose, how-to-use it, how it works, and to provide details about its internal design, what globally may involve a large diversity of contents, and many different ways of presenting them [Butler et al., 2000]. The inherent complexity of framework documentation is mainly due to the following requirements:

- **Different audiences.** As frameworks can be (re)used in several ways and by different kinds of people, this variety must be appropriately reflected in different views of the accompanying documentation. There are four main kinds of framework users: framework selectors, application developers, framework developers and developers of other frameworks. To be useful, the documentation must be targeted to the requirements of each kind of users, according to their specific needs in terms of: kind of information, level of abstraction or detail, and level of focus, either global or local.

- **Different types of documents.** To fulfill all the audiences and requirements, framework documentation usually provides multiple views (static, dynamic, external, internal) at different levels of abstraction (architecture, design, implementation). Typical examples of framework document types are: framework overviews, example applications, cookbooks and recipes, design patterns, use cases, contracts, design notebooks and reference manuals.

- **Different notations.** Depending on the concrete aspect to document, it may be convenient to use and mix contents represented in different notations: free text, structured text, source code, object models, images, formal specifications, etc. In addition, to ensure a good navigability, the contents must be fully cross-referenced, a task that requires tool support in order to be effective and economical.

- **Easy-to-use.** All this complex web of documents and contents must be properly presented to the different audiences in a simple manner, so that readers don’t become overwhelmed or lost when using the documentation and thus can quickly acquire the strict degree of understanding they need to accomplish their particular engineering tasks (reuse, maintenance or evolution).

As a result of this complexity of requirements, the cost of producing good framework documentation, especially without well-defined methods and tools, can be very high, and even worst, it can be risky.
The production of high-quality documentation is a crosscutting issue of framework-based development, whose contributions may help also on reducing the steep learning curve of frameworks, on improving framework understanding, and on evaluating framework applicability.

1.4 Research Goals

Much of the work on framework documentation has focused more on finding ways of documenting the design and architecture of frameworks [Butler and Denommée, 1999] and less on exploring effective ways of describing the purpose and intended use of frameworks.

Although several approaches to framework documentation have proven to be effective, the best mix of document types, writing techniques, and presentation styles depends strongly on the specific objectives, context and economics of the project at hands.

Despite the research done, there are still open issues related with framework documentation [Butler et al., 2000; Fayad, 1999], namely the definition of suitable methods and tools for an economic production of high-quality framework documentation, and the exploration of effective ways of describing and presenting the purpose and intended use of a framework [Butler and Denommée, 1999].

This research focuses on the following questions:

- What are the attributes of "high-quality framework documentation"?
- How to streamline the production of "high-quality framework documentation"?
- How to organize and present the documentation of a framework in order to improve its understandability and usability, especially by first-time users?

1.4.1 Primary Goal

The primary goal of this research is to define an approach to produce high-quality framework documentation capable of reducing the typically high costs associated with its production. The most important requirements of such approach are the following:

- easy-to-use by all kinds of authors (developers, technical writers, testers, etc.), so that the activity of documentation can act as a means to improve development productivity and quality, instead of being considered an obstacle, as happens in many development
environments;

- *flexible* enough to be easily adaptable to the needs of different projects and development environments;

- *economical*, to reduce the typical high-costs associated with the production of good quality documentation.

### 1.4.2 Secondary Goal

A secondary goal is *to define a way to organize and present framework documentation* so that it can improve the understandability and usability of the respective framework. To achieve such goal, the documentation should be organized and presented in a way that fulfills the following requirements:

- *easy-to-use* by all kinds of audience, so that the readers can locate, read and understand, with little effort, the parts of the documentation containing the knowledge they need to acquire in order to perform their particular engineering tasks;

- *task-oriented* organization of contents, to favor searches on *how-to-do* information over *how-it-works* information.

To help achieve this goal, the application of the minimalist instruction theory will be explored because it seems to be particularly helpful. The key idea of minimalist instruction is to minimize the obtrusiveness to the learner of training material, by offering the minimal amount of information that she needs to get the task done. Typically, minimalist manuals are considered easy to read and understand, contributing for shorter learning curves on how-to-use a system, and for a better understanding of the systems being trained [Carroll, 1990; Carroll, 1998; Rosson et al., 1990].

Although the application of the ideas of minimalist instruction to document frameworks have been reported by other authors as a promising approach, namely [Rosson et al., 1990], [Chai, 1999] and [Østerbye, 1995], the results achieved are superficial, suggesting that the topic still needs more research.

### 1.5 Research Strategy

Before detailing the strategy followed to address the problem of documenting frameworks and reach the goals above defined, a brief review of the methods and terminology commonly used in software engineering research is presented.
1.5.1 Software Engineering Research Methods

Software engineering has a relatively short history and is still maturing as a research area. Which research methods to use in software engineering research is a subject still under debate and is itself an open issue within the software engineering research community, e.g., [Tichy et al., 1993; Zelkowitz and Wallace, 1998].

Software development has specific characteristics that suggest its own research paradigm combining aspects from other disciplines: it is a human creative phenomenon; software projects are costly and usually have long cycle times; it is difficult to control all relevant parameters; technology changes very frequently, so old knowledge becomes obsolete fast; it is difficult to replicate studies; and there are few common ground theories.

A categorization proposed at Dagstuhl workshop [Tichy et al., 1993], groups research methods in four general categories, quoted from [Zelkowitz and Wallace, 1998]:

- **Scientific method.** “Scientists develop a theory to explain a phenomenon; they propose a hypothesis and then test alternative variations of the hypothesis. As they do so, they collect data to verify or refute the claims of the hypothesis.”

- **Engineering method.** “Engineers develop and test a solution to a hypothesis. Based upon the results of the test, they improve the solution until it requires no further improvement.”

- **Empirical method.** “A statistical method is proposed as a means to validate a given hypothesis. Unlike the scientific method, there may not be a formal model or theory describing the hypothesis. Data is collected to verify the hypothesis.”

- **Analytical method.** “A formal theory is developed, and results derived from that theory can be compared with empirical observations.”

These categories apply to science in general. Effective experimentation in software engineering requires more specific approaches. As software engineering research comprises computer science issues, human issues and organizational issues, it is often convenient to use combinations of research approaches both from computer science and social sciences.

The taxonomy described in [Zelkowitz and Wallace, 1998] identifies twelve different types of experimental approaches for software engineering, grouped into three broad categories, quoted from [Zelkowitz and Wallace, 1998]:

- **Observational methods.** “An observational method collects relevant data as
a project develops. There is relatively little control over the development process other than through using the new technology that is being studied”. There are four types: project monitoring, case study, assertion, and field study.

- **Historical methods.** “A historical method collects data from projects that have already been completed. The data already exist; it is only necessary to analyze what has already been collected”. There are four methods: literature search, legacy data, lessons learned, and static analysis.

- **Controlled methods.** “A controlled method provides multiple instances of an observation for statistical validity of the results. This method is the classical method of experimental design in other scientific disciplines”. There are four types of controlled methods: replicated experiment, synthetic environment experiment, dynamic analysis, and simulation.

The best combination of methods to use in a concrete research approach is strongly dependent on the specific characteristics of the research study to perform, namely its purpose, environment and resources. Hereafter, the research methods referred will use this terminology. A detailed description of each method can be found in [Zelkowitz and Wallace, 1998].

### 1.5.2 Research Model and Phases

The strategy devised for this research encompasses four main phases, as depicted in Figure 1.1. Different categories and types of research methods were used in different phases of the work, taking in consideration the specificities of each phase: the object of research, the results to produce (an hypothesis, a model, a process, or a product), the purpose (characterization, evaluation, prediction, control, improvement), the focus (the aspect of interest of the object of study), and naturally the resources available (time, projects).

![Figure 1.1](image)

**Figure 1.1** Research model: phases and respective research methods.

Succinctly, scientific methods were applied to derive theoretical models from real-world observations; engineering methods were applied to develop concrete software solutions; empirical methods were used only at the end of the research to validate part of the results achieved.
The initial phase of the research was dedicated to gather and synthesize information on the topics most relevant for the problem defined. The goals of this phase were: (1) to overview the main topics involved and their open issues, and (2) to narrow the directions of the research. The most important topics reviewed directly related with frameworks were the following: software reuse, software architecture, object-oriented frameworks, software patterns, software components, and software development methods. In terms of documentation, the topics reviewed were: software documentation, literate programming, minimalist instruction theory, and XML technology. The methods used were historical methods, namely literature search, study of lessons learned documents, study of legacy data, static analysis of existing tools, and observational methods, concretely one case study and assertions made during SIMAT project [Pires et al., 1999; INESC Porto, 2000] and SIMATWARE project [ParadigmaXis, 2001]. The main tangible results of this phase are included in the background and state-of-the-art review parts of this dissertation (Chapters 2, 3 and 4).

Based on the information collected, my own experience on framework-based software development, and the resources available to this work, the research questions were derived (see Section 1.3, p. 4) and an hypothesis was formulated. The envisioned solution consists on a new approach to document frameworks inspired on the existing theory of minimalist instruction, presented in Chapter 4, “Minimalist Documentation” (p. 105). The thesis statement is outlined in Section 1.6, is detailed in Chapter 5.5, “Thesis Statement” (p. 138), and validated in Chapter 8, “Thesis Validation” (p. 209).

Once formulated the hypothesis, the approach was developed using an agile method [Agile Alliance, 2001] with which the documentation approach was iteratively designed, built, explored, analyzed, validated and refined. The research methods used in this phase were mostly observational methods, namely the monitoring of the development, researcher assertions, and a case study for documenting JUnit – a very popular open source Java testing framework [Beck and Gamma, 1997]. This case study was the most important experimentation method for validating the solution during its development phase. The main results of this phase are included in the central chapters of this dissertation (Chapters 6 and 7).

After successive improvements of the solution and its continuous evaluation with the case study, the solution was considered satisfactory. In order to do an external validation of the solution from a framework developer's perspective, another case study was initiated by the authors of COOLFluID framework [Quintino, 2003], but it was not concluded at the moment of this writing. In order to evaluate the approach from a framework learner's perspective a controlled method was chosen. The method consisted on a replicated experiment in a synthetic environment where two groups of
programmers (subjects) were staffed to perform a similar set of tasks using two different sets of documents (factors) of a same framework. The goal was to evaluate the effect of the factors on attributes of interest. Due to the costs of such experiment, it was performed only once, but the experiment was designed to enable its easy replication by other interested researchers. The evaluation made with the case study and the experiment is presented in Chapter 8, “Thesis Validation” (p. 209).

1.6 Thesis Statement – Preliminary Version

The initial thesis statement of this dissertation is the following:

- The minimalist approach to framework documentation makes the production and usage of framework documentation easier than is possible with traditional approaches.

The analysis of this initial thesis statement rises some questions that need discussion and clarification:

- what is the scope of “production” and “usage”?
- what is meant by “easier”?
- which are the “traditional approaches”?
- “who” does benefit from these improvements?

In Chapter 5, “Research Problem” (p. 121), this and other questions will be examined and the final thesis statement detailed to explain the validation methodology. The thesis statement is validated in Chapter 8, “Thesis Validation” (p. 209).

1.7 Main Results

The main result of the research is the proposal of a new framework documentation approach that aims to be simple and economic to adopt. The approach reuses existing styles, techniques and tools for documenting frameworks, and combines them in a way that follows the design principles of minimalist instruction theory, hence it was coined a minimalist approach to framework documentation.

The resulting documentation assumes the form of a minimalist framework manual, a kind of instruction manual that emphasizes the understandability and usability of a framework, including information about the application
domain, its purpose, how-to-use it, how it works, and internal design details.

The approach informally provides guidance about what, when and how to
document, and supports a kind of integrated content management covering
the overall process of framework documentation from the initial phases of
creation and integration of contents till the last phases of publishing and
presentation to target audiences.

In concrete, the minimalist approach proposes a documentation model, a
documentation process, and a set of support tools that make it convenient to use in
mainstream development environments. The components of the approach
are summarized below and described in detail in Chapter 6.

1.7.1 Documentation Model

The documentation model is the core of the approach. It enumerates and
organizes all the contents required to produce a minimalist framework
manual. The overall contents can be divided in two main categories:

- **typical contents of framework documentation**: code examples, recipes and
cookbooks, design patterns, framework overviews, reference manuals,
design notebooks, use cases, scenarios and contracts;

- **typical contents of minimalist manuals**: user tasks, usage patterns, task
information contexts, error inventories, error recovery guidelines, and
classification of contents according to usage modes.

To be useful, the overall repository of contents can be configured, using
meta-information, to be distributed over a virtual layered documentation
space. In addition, to cope with a vast diversity of documentation
requirements, the model is extensible so that new custom styles of
documents, notations, and layers can be added with a small effort.

1.7.2 Documentation Process

The documentation process defines the roles, techniques, and activities
involved in the production of the minimalist framework manuals. The roles
identified are:

- **developers**, such as framework developers and framework maintainers,
  which are responsible for creating contents mostly during the
development phase;

- **technical writers**, which are responsible to structure, guide, review, and
  conclude the documentation;
• **documentation managers**, which are responsible for configuring and maintaining the documentation base, namely the template documents, template instances, and content transformations.

Because the production of framework documentation is closely related with framework design and usage, ideally, these activities should be done side-by-side, if we want to obtain documentation that is understandable, consistent, and easy-to-maintain. The key activities of producing framework documentation are:

• **configuration** of the documentation base;
• **creation and cross-referencing** of the various kinds of contents;
• **normalization, integration, and storage** of contents;
• **publishing and presentation** of contents to target audiences.

The documentation process was abstracted from concrete documentation processes and designed with a special concern on lightweight processes, which typically allocate very little effort for documentation, thus being very restrictive on adopting a documentation approach. Therefore, the resulting process is simple, flexible, and easy to adapt to different development processes and environments, ranging from literate programming environments [Knuth, 1984; Williams, 1992] to industrial integrated development environments.

### 1.7.3 Documentation Tools

To support the approach and making it convenient to be adopted in a wide range of software development environments, a specific set of tools has been developed, collectively named XSDoc.

XSDoc is an infrastructure based on XML [Bray et al., 1998] and WikiWikiWeb [Cunningham, 1999] that provides a simple cooperative web-based environment for the creation, integration, publishing, and presentation of minimalist framework documentation.

The XSDoc infrastructure is composed by one Wiki engine, plug-ins for integration in open IDEs, and a set of pluggable document templates (for cookbooks, design patterns, framework overviews, etc.), markup languages (JavaML, XSDocML) and converters of contents to and from XML. Currently, XSDoc supports source code written in Java and C++ programming languages, models described in UML [OMG, 2001], and a simple integration with Eclipse IDE [Eclipse, 2003].

XSDoc combines the simplicity, easiness and versatility of the collaborative
environment for document edition provided by Wiki engines, with the powerful development features of modern open IDEs, taking advantage of the well-known qualities of XML technology for information processing.

This combination of technologies results in a very attractive infrastructure that helps to reduce the effort typically required to document frameworks. Some of the qualities of XSDoc are: easy-to-use by all team elements (technical or not); easy to integrate in software development environments; promotes the participation of all team elements in the documentation process, improving their intercommunication; provides an easy access to and high-availability of the documentation; and finally, facilitates the edition, revision and evolution of framework documentation contents in a controlled and structured way.

1.8 Future Work

This research addressed primarily issues related with framework documentation, but by inference it has uncovered other issues on framework development processes, software documentation, and on empirical research on software engineering. Chapter 9 presents areas for future research in the sequence of this work.

1.9 How to Read this Dissertation

This dissertation is logically organized in three parts. The first part reviews the most important concepts and issues of framework technology, framework documentation and minimalist instruction theory (Chapters 2, 3 and 4). The second part develops the research questions based on the analysis of related work, and presents the proposed approach for documenting frameworks, both in theory and applied to the case study used along the development of the approach (Chapters 5, 6 and 7). The third and final part reiterates the claims posed by this dissertation and validates them using assertions, experiences gained from the case studies, and results obtained from the controlled experiment (Chapters 8 and 9).

Part 1: Background and Reviews. The first part provides background and reviews on frameworks, framework documentation and minimalist documentation.

- Chapter 2, “Framework Fundamentals” (p. 17), puts object-oriented frameworks into the context of software reuse and software architecture and reviews the most fundamental concepts of frameworks.
• Chapter 3, "Building, Using and Documenting Frameworks" (p. 43), reviews the major activities and techniques used in framework-based software development, putting a special emphasis on their requirements in terms of framework documentation.

• Chapter 4, "Minimalist Documentation" (p. 105), reviews the design theory of minimalist instruction, namely its values, design principles, and applicability to software documentation.

**Part 2: Problem and Solution.** The second part states the problem researched and the proposed approach.

• Chapter 5, "Research Problem" (p. 121), analyses related work previously reviewed, formulates the final thesis statement of this dissertation, and outlines the validation strategy.

• Chapter 6, "The Minimalist Approach to Framework Documentation" (p. 145), presents the solution proposed to improve framework documentation: a model, a process and a set of tools.

**Part 3: Validation and Conclusions.** The third part presents case studies and one experiment, validates the thesis, and presents the conclusions of the dissertation.

• Chapter 7, "Case Studies and Experiment" (p. 193), presents and discusses the results of case studies used to evaluate and validate the approach proposed.

• Chapter 8, "Thesis Validation" (p. 209), reviews the results obtained from the case studies, and supports them with qualitative arguments, and results from a controlled experiment.

• Chapter 9, "Conclusions" (p. 225), reviews the dissertation as a whole, explains the dissertation results, both from the thesis perspective and from a wider perspective, and points questions for future research.

For a comprehensive understanding of all the dissertation, the review and solution chapters must be read first. Then case studies may be read followed by the thesis validation. Those already familiar with framework technology who only want to get a fast but detailed impression of the work may restrict their reading to this introductory chapter, Chapter 5, and Chapter 6.
Framework Fundamentals

An object-oriented framework is a cohesive design and implementation artifact. Frameworks serve to implement larger-scale components, and are implemented using smaller-scale classes [Riehle, 2000].

Since its creation at the end of the 1980s, the concept of object-oriented frameworks has attracted a lot of attention from researchers and software engineers, resulting in many frameworks being developed in industry and academia, covering different application domains. The benefits from frameworks include reduced time to market and improved compatibility and consistency [Taligent Press, 1994; Fayad and Schmidt, 1997a; Fayad et al., 1999].

This chapter presents the fundamental characteristics of the framework concept. It reviews common terminology, associated object-orientation concepts, the key benefits of reusing frameworks, and the role of frameworks in the context of object-oriented software architecture. The chapter concludes with a brief history of frameworks, from the early Simula frameworks (1960s) to present frameworks.
2.1 What Is a Framework?

A framework is a reusable design together with an implementation. It consists of a collection of cooperating classes, both abstract and concrete, which embody an abstract design for solutions to problems in an application domain [Johnson and Foote, 1988; Deutsch, 1989; Campbell et al., 1991; Cotter and Potel, 1995; 1; LAC+95; Fayad and Schmidt, 1997b; Fayad et al., 1999].

But frameworks are more than just collections of classes. Frameworks are also architectural. A framework defines the overall application structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. So, frameworks dictate the architecture of the applications we build with them, but still leave enough design space to accommodate particular solutions. By predefining design parameters that are invariant in an application domain, frameworks help application developers get the key architectural aspects right from the beginning, letting them concentrate on the specifics of their applications.

When using a framework, we reuse not only analysis and design but also implementations. With a framework, developers can build applications by extending or customizing only some parts, while reusing framework implementations of the rest and retaining the original design.

2.2 Frameworks and Reuse

The simple yet powerful vision of software reuse was introduced in 1968 [Naur and Randell, 1968] as a means to reduce the time and effort required to build and maintain high-quality software systems. In a broad sense, software reuse is the process of creating new software systems by starting from existing artifacts rather than building them from scratch.

Reuse does not happen by accident. We need to plan to reuse software, and look for software to reuse. Reuse requires the right attitude, tools and techniques [Johnson and Foote, 1988]. Object-oriented frameworks are one reuse technique, actually a powerful one that enables large-scale reuse.

2.2.1 Reuse Techniques

There are several techniques for software reuse, each possibly using different artifacts. Reusable software artifacts include source code fragments, design structures, abstract specifications, and documentation, to mention a few. Tools and techniques to support software reuse are usually categorized in
compositional and generative approaches. While the compositional approach is based on reusing software artifacts, the generative approach is based on reusing software development processes, often embodied in tools that help automate them. Source code components and application generators are just two examples of such approaches, respectively [Krueger, 1992].

A reuse technique must support one or all of the four important activities of software reuse, namely: abstracting artifacts; selecting artifacts, which includes classifying, finding and understanding them; specializing artifacts; and integrating artifacts [Biggerstaff and Richter, 1989].

The determination of the best reuse technique is often difficult to do, as it depends a lot on the specificities of the project at hands. As an intuitive gauge to compare the effectiveness of different reuse techniques, Krueger used the notion of cognitive distance [Krueger, 1992]. He informally defines it as the amount of intellectual effort that must be expended in developing a software system to go from the initial conceptualization to a specification expressed in abstractions of the reuse technique, and from these to an executable system.

An ideal reuse technique should let us quickly find components that exactly fit our needs, are ready to use without being customized, and don't force us to learn how to use them. The developer's ability to reuse software is limited primarily by his ability to reason in terms of the abstractions used by the reuse technique. In other words, the cognitive distance between informal reasoning and the abstract concepts of the technique must be small. Natural, succinct and high-level abstractions describing artifacts in terms of “what” they do, rather than “how” they do it, are thus very important for effective software reuse. From all the existing reuse techniques, the reuse of software architectures is probably the technique that comes closest to this ideal.

2.2.2 How Object-Orientation Leverages Software Reuse?

Historically, in the 1960s, reusable software components have been procedural libraries. In 1967, with their language Simula 67 [Dahl et al., 1970], Dahl and Nygaard have introduced most of the key concepts of object-oriented programming, namely objects, classes and inheritance, and with these concepts one of the main paradigms of programming have started, the object-oriented programming.

Simula concepts have been important in the discussion of abstract data types and models for concurrent program execution, starting in the early 1970s. Alan Kay's group at Xerox PARC used Simula as a platform for their development of the first language versions of Smalltalk, in the 1970s, extending object-oriented programming importantly with the integration of
graphical user interfaces and interactive program execution. In the 1980s, Bjarne Stroustrup started his development of C++, by bringing the key concepts of Simula into the C programming language. Simula has also inspired much work in the area of component reuse and construction of component libraries.

Object-oriented programming is today becoming the dominant style for implementing complex applications involving a large number of interacting components. Among the multitude of object-oriented languages are Smalltalk, Object Pascal, C++, Common Lisp Object System (CLOS), Eiffel, BETA, and SELF. In particular, the Internet-related Java (developed by Sun) has rapidly become widely used in the last 1990s.

With the integration of data and operations into objects and classes, reusability has increased. The classes were packaged together into class libraries, often consisting of classes for different data structures, such as lists and queues. Class libraries were further structured using inheritance to facilitate the specialization of their classes. As a result, class libraries became capable of delivering software reuse beyond traditional procedural libraries.

Object-oriented programming languages combine features, such as data abstraction, polymorphism and inheritance, that encourage the reuse of existing code instead of writing new code from scratch. These features are detailed later in Section 2.4.1 (p. 29).

Taking advantage of these features, object-oriented languages promote the development of class libraries, which, like the procedural libraries, are mainly focused on reuse of code. But code reuse has limitations, working best when the domain is narrow, well understood, and the underlying technology is very static. In the long run, reusing the design of an application is probably more beneficial in economical terms than reusing the implementation of any of its components [Biggerstaff and Richter, 1987], because design is the main intellectual content of software and it is far more difficult to create and re-create than code [Deutsch, 1989].

Although object-orientation has started with object-oriented programming languages, it is more than object-oriented programming. Object-orientation covers also earlier phases of programming, such as analysis and design. Using a small set of concepts (objects, classes, and their relationships) developers can model an application domain (analysis), define an architecture to represent that model on a computer (design), and implement that architecture to let a computer execute the model (programming) [Booch, 1994].

As a whole, object-orientation introduced in software development more qualities that favor software reuse, namely, problem-orientation, resilience to
evolution, and domain analysis.

- **Problem-orientation.** The object-oriented models produced during analysis are all described in terms of the problem domain, which can be mapped directly to object-oriented concepts, such as classes, objects and relationships. This seamlessness from analysis to programming models makes them simpler to communicate between users and developers, and enables the delivery of better software products [Hoydalsvik and Sindre, 1993].

- **Resilience to evolution.** In an application domain, processes change more often than the entities. As object-oriented models are structured around the entities, they are more stable to changes and therefore less resilient to evolution [Meyer, 1988].

- **Domain analysis.** Object-oriented analysis is naturally extensible to domain analysis, a broader and more extensive kind of analysis that tries to capture the requirements of the complete problem domain, including future requirements [Schafer et al., 1994].

Taking advantage of all these qualities, reusability appeared as one of the great promises of object-orientation, based on the reuse of code through inheritance. But the efforts only provided reuse at the level of small-scale components, usable as primitive building blocks of new applications. Neither object-orientation nor class libraries made possible the reuse of large-scale components. This understanding led to the conception of object-oriented frameworks, a kind of large and abstract application specially designed to be tailored for the development of concrete applications in a particular domain. In the beginning of the 2000s, object-oriented frameworks represent the state-of-the-art in terms of object-oriented reusable products.

### 2.2.3 The Power of Frameworks

Since its conception at the end of 1980s, the appealing concept of object-oriented framework has attracted a lot of attention from many researchers and software engineers. During the 1990s, frameworks have been built for a large variety of domains, such as user interfaces, operating systems, and distributed systems.

A framework can be shortly defined as a reusable design of an application together with an implementation [Johnson and Foote, 1988; Campbell et al., 1991; LAC+95; Fayad and Schmidt, 1997b; Fayad et al., 1999]. The definitions for a framework are not consensual and vary from author to author. In few words, a framework can be defined as a semi-complete design and implementation for an application in a given problem domain.
A powerful reuse technique

As mentioned before, frameworks are firmly in the middle of the reuse techniques. They are more abstract and flexible (and harder to learn) than components, but more concrete and easier to reuse than a raw design (but less flexible and less likely to be applicable). Frameworks are considered a powerful reuse technique because they lead to one of the most important kinds of reuse, the reuse of design. When compared to other techniques for reusing high-level design, such as templates [Spencer, 1988] or schemes [Katz et al., 1989], frameworks have the advantage of being expressed in a programming language, thereby resulting easier to learn and apply by programmers.

Frameworks and components

Frameworks and components are cooperating technologies. Software components are “binary units of independent production, acquisition, and development that interact to form a functioning system, with explicit interfaces and context dependencies only. A software component can only be deployed independently and is subject to composition by third parties.” [Szyperski, 1998]. Frameworks provide a reusable context for components, in the form of component specifications and templates for their implementation, thereby making it easier to develop new components.

Frameworks and patterns

Frameworks and design patterns are concepts closely related as well, representing two different categories of high-level design abstractions [Johnson, 1992]. A single framework typically encompasses several design patterns. Patterns provide an intermediate level of abstraction between the application level and the level of classes and objects.

A design pattern is commonly defined as a generic solution to a recurring design problem that might arise in a given context [Alexander et al., 1977; 1; Buschmann et al., 1996]. The relationships between individual patterns unfold in the application domain naturally and form a high level language, called a pattern language [Alexander 1977]. A pattern language represents the essential design knowledge of a specific application domain, i.e. the experience gained by many designers in solving a class of similar problems. Design patterns and pattern languages are particularly good for documenting frameworks because they capture design experience and enclose meta-knowledge about how flexibility was incorporated. Pattern languages help document the application domain of the framework, the design of the framework in terms of classes, objects and their relationships, and also the specifications of important framework classes.

The combined use of frameworks with patterns and components is very effective, significantly helping to increase software quality and reduce development effort [Fayad et al., 1999].

Framework benefits

The benefits of frameworks stem primarily from the levels of code and design reuse being much higher than what is possible with other reuse
technologies, such as code generators and class libraries. In addition to the reusability benefits, other advantages are due to the inversion of control, the modularity and the extensibility that frameworks provide to developers.

In general terms, the benefits from frameworks include higher development productivity, shorter time-to-market and higher quality. However, framework benefits are not necessarily immediate, but only gained over time. As significant productivity gains usually start appearing after multiple uses of the technology, frameworks must be considered a medium-to-long term investment.

The benefits from frameworks impact in many phases of application development, from analysis to maintenance and evolution. During the analysis phase, frameworks help developers reduce the effort usually required to understand the overall application domain, and enable them to focus on the details of the application at hands.

It is during the design, coding, testing and debugging of applications that frameworks have more advantages over traditional application development. Most of the benefits result from the high levels of design and code reuse provided, and also the inversion of control (Section 2.4.3) possible with frameworks:

- provide guidance on application architecture;
- improve programming productivity and quality by reducing the amount of code to design and write;
- improve modularity and understandability by encapsulating volatile implementation details behind stable interfaces;
- promote the development of generic solutions reusable across an application domain;
- and improve application integrability and interoperability due to shared architecture and design.

The benefits of frameworks are not less important at maintenance and evolution phases. The reusability and extensibility possible with frameworks help to decrease the effort of maintenance due to its amortization over many application specific parts. Framework-based applications are also easier to evolve without sacrificing compatibility and interoperability because frameworks provide explicit hook methods that allow applications to modify or extend framework's behavior.

In summary, through design and code reuse, frameworks help us reduce the amount of design we must create and the lines of code we must write, therefore significantly improving productivity. As a result, not only we can
build applications faster, but also build applications that are easier to maintain and more consistent, because they share a similar structure at all levels of software design.

As software systems evolve in complexity, object-oriented application frameworks are being successfully applied in more application domains and therefore becoming more important for industry and academia. Application frameworks offer software developers an important vehicle for reuse and a means of capturing the essence of successful architectures, patterns, components and programming mechanisms.

Perhaps the best evidence of the power of object-oriented frameworks is reflected on the well-known success of many examples of popular frameworks, such as: Model-View-Controller (MVC) [Goldberg, 1984], MacApp [Schmucker, 1986], ET++ [Weinand et al., 1989], Interviews [Linton et al., 1989], OpenDoc [Feiler and Meadow, 1996], Microsoft Foundation Classes (MFCs) [Prosise, 1999], IBM’s SanFrancisco [Monday et al., 2000], several parts of Sun’s Java Foundation Classes (RMI, AWT, Swing) [Drye and Wake, 1999], many implementations of the Object Management Group’s (OMG) Common Object Request Broker Architecture (CORBA), and Apache’s frameworks (CoCoon, Struts) [Apache, 1999]. 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.

2.3 Object-Oriented Software Architecture

Software design, and system design in general, take place at different levels. Each level has components, both primitive and composite, rules of composition guiding the construction of non-primitive components, and rules of behavior providing semantics for the system. For software, at least three design levels are usually identified [Shaw and Garlan, 1996]:

- an architecture level, where the design issues involve the overall organization of a system as a composition of components, the definition of global control structures, and the assignment of functionality to design elements;
- a code level, where the design issues involve algorithms and data structures;
- and an executable level, where the design issues involve memory maps, call stacks, register allocations, and machine code operations.
As the size and complexity of software systems increase, the most important design problems are no longer the design of the algorithms and data structures, but instead the design and specification of the overall system structure.

2.3.1 What is Software Architecture?

Software architecture is an emergent field of study in software engineering specifically concerned with software design at the architecture level. Its importance to software engineering practitioners and researchers has significantly increased during the 1990s, in response to the growing need for exploiting commonalities in system architectures, on making good choices among design alternatives, and describing high-level properties of complex systems.

According to Webster's Dictionary, architecture is “the art or practice of designing and building structures...”. The main concern of software architecture is the design and building of structure, and not the individual building blocks that bring the structures into existence.

Abstractly, software architecture describes the components from which systems are built, and the interactions among those components—the connectors. Software architecture may also describe the rules and mechanisms that guide the composition of components and eventual constraints on those rules.

Components at the architecture level can be things such as clients, servers, databases, filters, and layers of a hierarchical system. Examples of connectors range from simple procedure calls to complex protocols, such as client-server protocols, database-accessing protocols, event multicast, and pipes.

2.3.2 Architectural Levels

Object-oriented software architecture is particularly interested in the architecture of object-oriented systems, that is, on architectures having objects and classes as their primitive building blocks. Current practice suggests four levels of granularity to describe an object-oriented system: the class level, the pattern level (micro-architecture), the framework level (macro-architecture), and the component level.

Class level

At the smallest level of granularity, a system is designed as a set of classes, whose instances cooperate to achieve some sophisticated behavior otherwise impossible with a single object.

A class represents a well defined concept or entity of the domain. An object is an instance of a class, has a state, exhibits some well-defined behavior, and
has a unique identity. The structure and behavior of similar objects are defined in their common class. Whereas an object is a concrete software entity that exists in time and space, a class represents only an abstraction, the essence of an object [Booch, 1994].

For small systems, objects and classes are sufficient means for describing their architecture. However, as a system becomes bigger, more and more classes get involved in its architecture, and higher-level abstractions are needed to help developers cope with the complexity of designing and implementing such systems.

![Diagram of Components, Frameworks, Patterns, Classes](image)

**Figure 2.1** Design elements of object-oriented architectures

Pattern level: Immediately above the level of classes, we can use patterns to describe the micro-architectures of a system. 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 [Alexander et al., 1977]. 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 [Alexander et al., 1977; Alexander, 1979; Lea, 1994].

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.

There are different kinds of patterns, of varying scale and level of abstraction, being usually classified in architectural patterns, design patterns, and idioms [Buschmann et al., 1996].

- **Architectural patterns** express fundamental structural organization schemes for software systems.

- **Design patterns** are medium-scale tactical patterns that reveal structural and behavioral details of a set of entities and their relationships. They do not influence overall system structure, but instead define micro-architectures of subsystems and components.

- **Idioms** (sometimes also called coding patterns) are low-level patterns that describe how to implement particular aspects of components or relationships using the features of a specific programming language.

Patterns represent useful mental building blocks for dealing with specific design problems of software system development.

### Framework level

Object-oriented systems of medium size typically involve a large number of classes, some patterns, and few layers of cooperating frameworks. Frameworks are used to describe a system at an higher level than classes and patterns.

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.

A framework can also be seen as a representation of a specific domain under the form of a reusable design together with a set of implementations often
reusable and ready to instantiate.

A good framework has well-defined boundaries, along which they interact with clients, and an implementation that is usually hidden from the outside. Frameworks are a key part of medium to large-scale development, but even they have an upper limit to cope with high levels of complexity [Bäumer et al., 1997].

Component level

On the highest level of granularity, a system can be described as a set of large-scale components that work together to support a cohesive set of responsibilities. A component is defined in [Szyperski, 1998] as a "unit of composition with contractually specified interfaces and explicit context dependencies only; (...) it can be deployed independently and is subject to composition by third parties". Examples of large-scale components are domain components, which are collections of related domain classes covering a well-defined application domain or a part of. Large components may or may not have been built from one or more object frameworks [Wegner et al., 1992], but in the case of an object-oriented system they typically are.

2.4 Fundamental Concepts

An object-oriented framework is a reusable software architecture comprising both design and code. Although this statement is generally accepted by most authors, there are a number of different definitions for object-oriented frameworks that emphasize other aspects of the framework concept.

The most referenced definition is perhaps the one found in [Johnson and Foote, 1988], which says that: "a framework is a set of classes that embodies an abstract design for solutions to a family of related problems". This definition captures the essential aspects of the object-oriented framework concept, namely: (1) a framework comprises a set of classes; (2) a framework embodies a reusable design; and (3) a framework addresses a family of problems in a domain.

Other definitions present other aspects of frameworks, which altogether help us get a better understanding of the concept. For example, Deutsch states that "a framework binds certain choices about state partitioning and control flow; the (re)user (of the framework) completes or extends the framework to produce an actual application" [Deutsch, 1989]. The first part of this definition emphasizes (4) the structural aspect of a framework, by stating that architectural design decisions have been taken. The second part explicitly describes the main purpose of a framework, which is (5) to be adapted to the problem at hands, namely by extending or completing some of its parts.
In [1] a framework is defined as “a set of cooperating classes that make up a reusable design for a specific class of software”, which is based on the two definitions above mentioned.

The definition given in [Cotter and Potel, 1995] concisely presents almost all the aspects previously presented (all but the (4)): “A framework embodies a generic design, comprised of a set of cooperating classes, which can be adapted to a variety of specific problems within a given domain”.

In the following definition, in [Johnson, 1997], a framework is defined as “(...) the skeleton of an application that can be customized by an application developer”. This definition reinforces the structural aspect of a framework, and that future applications will conform to them by customizing parts of the framework. The activity of framework “adaptation” is referred in this definition as framework “customization”, but the essential meaning of both terms are similar. In the same reference, a framework is also defined as “(...) a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact”. This definition indicates that a framework doesn’t necessarily need to cover a complete problem domain, but possibly only smaller parts of it, thereby suggesting the possibility of composing several frameworks together to build concrete applications. The wording “set of abstract classes” may suggest that the extension of a framework has to be done through inheritance, but this is not completely true as there are other ways of extending a framework, namely by composition.

Therefore, using a more complete and longer definition, we can define a framework as a software artifact:

- encompassing a set of cooperating classes, both abstract and concrete;
- expressed in a programming language, providing reuse of code and design;
- and specially designed to be customized, by inheritance or composition, for the construction of concrete solutions (systems, or applications) for a family of related problems within a specific problem domain.

Shortly, a framework emphasizes the more stable parts of an application domain, as well as their relationships and interactions, and provide customization mechanisms that let application developers solve their particular problems in the that domain.

### 2.4.1 Object-Orientation Concepts

Much of the reuse power of object-oriented frameworks comes from the
most distinguishing characteristics of object-oriented programming languages: data abstraction, inheritance, and polymorphism.

Data abstraction
Class definitions in an object-oriented language are primarily a data abstraction mechanism that enable the unification of data together with the procedures that manipulate them. Through abstraction and encapsulation, classes enable the separation of interfaces from implementations, and thus the change of implementation details without affecting its clients. As a result, classes can often serve as fine-grained reusable components.

Inheritance
In object-oriented languages, classes can be organized along hierarchies supporting different kinds of inheritance. Class inheritance allows the properties and behavior of a class to be inherited and reused by its subclasses. Inheritance in programming languages can be seen as a built-in code sharing mechanism that, without polymorphism and dynamic binding, won't be much different from several module import mechanisms of traditional languages.

Polymorphism
This is a feature of object-oriented languages that enables a variable to hold objects belonging to different classes. When combined with overloading and dynamic binding, polymorphism becomes a powerful feature of object-oriented languages that enables to mix and match components, to change collaborators at runtime, and to build generic objects that can work with a wide range of components. Overloading makes it possible for several classes to offer and implement many operations with the same name, being up to the compiler or runtime environment to disambiguate references to a particular operation.

The combination of these features allow for a greater flexibility in programming. Due to polymorphism, a single variable in a program can have many different types at run-time. Inheritance provides a way of controlling the range of types a variable can have, by allowing only type mutations within an inheritance tree. Finally, dynamic binding enables delaying until run-time the determination of the specific operation implementation (method) to be called in response to an operation request, when the actual types of the variable and operation parameters are known [Meyer, 1988; Booch, 1994].

Two of the most distinguishing features of the framework concept rely heavily on the use of dynamic binding: the extensive use of template and hook methods, and the inversion of control flow.

2.4.2 Template and Hook Methods
Frameworks are designed and implemented to fully exploit the use of dynamically bound methods. To illustrate this, we will present a simple
example of a hypothetical single class framework for unit testing.

The framework consists of a single abstract class named `TestCase`. This class has three operations named `setUp`, `runTest` and `tearDown`. In order to implement tests for database connection operations, or mathematical operations, for example, the framework is supposed to be extended with concrete subclasses, such as `DBConnectionTest` or `MoneyTest`.

As different tests usually have different ways of being setup, executed, and terminated, the framework, i.e. the `TestCase` class, doesn’t provide implementations for these operations, being up to framework users to provide them. Although the details of concrete test implementations may differ, the overall running of a test is always the same, consisting of: the setup of the test, the running of the concrete test, and its finalization.

To capture this commonality between different test implementations, the framework implements a generic operation to run tests. This generic operation is named `run`, and its implementation is responsible for handling the invocation of the `setUp`, `runTest` and `tearDown` operations. In other words, to run any test case, it is only needed to invoke the `run` operation of `TestCase` being up to `run` to do the rest. An illustrative implementation of the `TestCase` class using the Java programming language is shown in Figure 2.2.

Concrete subclasses of `TestCase`, such as `MoneyTest`, should provide implementations for the `setUp`, `runTest` and `tearDown` operations. Due to the mechanism of dynamic binding, when the `run` operation is called on an instance of `MoneyTest`, it is the `run` operation of `TestCase` that will be used (if not overridden in `MoneyTest`). The method `run` of `TestCase` will then invoke the `setUp`, `runTest` and `tearDown` operations implemented in the `MoneyTest` class.

```java
abstract public class TestCase {
    public void run(){
        setUp();
        try {
            runTest();
        } finally {
            tearDown();
        }
    }
    abstract protected void setUp();
    abstract protected void runTest();
    abstract protected void tearDown();
}
```

Figure 2.2 The class `TestCase`.
The point here deserving attention is the fact of an operation in a superclass, the run operation of `TestCase`, being able to call operations in subclasses, and therefore (the superclass) having control over the execution flow of the overall test sequence. The run operation is often called a template method, and the setUp, runTest and tearDown operations are called hook methods.

Template and hook methods are two kinds of methods extensively used in the implementation of frameworks. These terms are commonly used by several authors in [Wirfs-Brock et al., 1990; Pree, 1991; Gamma et al., 1993; Pree, 1995].

Template methods are implemented based on hook methods, and call at least one other method. A hook method is an elementary method in the context which the particular hook is used, and can be either an abstract method, a regular method, or another template method. An abstract method is a method for which the particular hook is provided, and thus lacks an implementation. A regular method is a method that doesn't call hook or template methods, but only provides a meaningful implementation.

Generally, template methods are used to implement the frozen spots of a framework, and hook methods are used to implement the hot spots. The frozen spots are aspects that are invariant along several applications in a domain, possibly representing abstract behavior, generic flow of control, or common object relationships. The hot spots of a framework are aspects of a domain that vary among applications and thus must be kept flexible and customizable.

The difficulty of good framework design resides exactly on the identification of the appropriate hot spots that provide the best level of flexibility required by framework users. More hot spots offers more flexibility, but results in a framework more difficult to design and use, so somewhere in between resides a balanced design.

In our simple testing framework example, the run template method implements the overall execution of a test case (a frozen spot), which consists on preceding the execution of the test case with a setup, followed by a tear down operation responsible to release any resources eventually used during the test. All these operations are supposed to be provided by the hook methods setUp, runTest and tearDown, which are abstract methods. The class `TestCase` is considered an abstract class because it has at least one abstract method (actually it has three).

Template and hook methods can be organized in several ways. Although they can be unified in a single class, as in our example, in most of the situations it is better to put frozen spots and hot spots into separate classes. When using separate classes, the class that contains the hook method(s) is
considered the *hook class* of the class containing the corresponding template method(s)—the *template class*. We can consider that hook classes parameterize the corresponding template class. The hook methods on which a template method is based can also be organized in different ways. They can be defined all in the same class, or in separate classes, in a superclass or subclass of the template class, or in any other class.

In [Pree, 1995] are identified several ways of composing template and hook classes, and presented under the form of a set of patterns, globally called *meta-patterns*. Meta-patterns categorize and describe the essential constructs of a framework, on a meta-level. Design patterns provide proven solutions to recurrent design problems and are extremely useful to design object-oriented frameworks.

In our framework example, template and hook methods are unified in a single class, because the object providing the template method is not separated from the objects providing the hook methods, actually being an instance of *MoneyTest* (an instance of *MoneyTest* is also an instance of *TestCase*). This organization of template and hook methods is classified as the *Unification meta-pattern*, which corresponds to the simplest way of organizing template and hook methods. In Figure 2.3, this meta-pattern is represented attached to the classes of our example.

![Unification meta-pattern](image)

**Figure 2.3** The Unification meta-pattern attached to the testing framework.

With this unification meta-pattern, the developer must provide a subclass to
adapt the behavior of running a test, and this can’t be done at run time. Organizations that separate template classes and hook classes are called *Connection meta-patterns*, which allow the modification of the behavior of a T object by composition, that is, by plugging in specific H objects. The more sophisticated way of separating template and hook classes, called *Recursive meta-patterns*, occurs when the template class is a descendant of the hook class, which enables the composition of whole graphs of objects. In Figure 2.4 we show the basic differences of unification, connection and recursive meta-patterns.

![Unification, connection, and recursive meta-patterns](image)

With the single class framework example we have illustrated the usage of inheritance and dynamic binding for operations in one single class. By scaling up the example to a larger framework, with more abstract classes, more template and hook methods organized according to more powerful meta-patterns, we can have a better idea of the potential reuse power that well-designed frameworks can deliver to their users.

### 2.4.3 The Flow of Control in Framework-based Applications

The development of applications reusing frameworks leads to an inversion of control between the application and the software on which it’s based. When we use a class library, we write the main body of the application and call the code we want to reuse. When we use a framework, we reuse the main body and write the code it calls [1]. By consequence, the code to be written must satisfy particular names and calling conventions defined by the framework, what reduces the design decisions we need to do. This inversion of control is characteristic to frameworks and is referred as the *Hollywood Principle*, meaning “Don’t call us, we’ll call you” [Cotter and Potel, 1995; Bosch et al., 1999].
This inversion of control flow in programs is an idea that has evolved over years of application development, passing by different ways of structuring programs: from procedural programs, to event-loop programs, and then to framework programs (Figure 2.5).

![Diagram showing the evolution of control flow in programs: Procedural programs, Event-loop programs, and Framework programs.]

**Figure 2.5** Evolution of control flow in programs.

**Procedural programs**

In *procedural programs*, all the code for control flow is provided by the programmer. The program is executed sequentially, always under the programmer's control, and when necessary calls procedures from libraries provided by the operating system. The system takes action only when it is called by the program.

**Event-loop programs**

When using graphical user interfaces, sequential control flow is no longer appropriate, as end users may select when and which actions to perform. A solution to this problem led to the concept of *event-loop programs*, which let the user choose the order in which events happen, through the interaction with input devices (mouse, keyboard, etc.). These programs have an event loop that is responsible to sense user events and call the corresponding parts of the program configured to handle them, remaining programmer's responsibility the flow of control within these parts.

**Framework programs**

Framework-based applications turn over control to the user, as happens with event-loop programs, and then to the original framework developers. The framework code assumes almost all flow of control, calling application code only when necessary. Calls are however not made exclusively in one direction: application code often calls framework code too. As a result of this two-way flow of control, it is not needed to design and write the control code required by event-loop programs or other code common to many applications that can be written once and reused many times afterwards. Ideally, with frameworks we design and write only a small part of the total
flow-of-control code required to implement the application.

The shifting of control flow is a question of degree, and not absolute. We can say that a program exists on a scale somewhere between 0% and 100% framework-owned control flow. When developing applications using frameworks the goal is to shift the control flow as much as possible to the framework.

Back to our example, we will now analyze the oscillation of the flow of control between the framework and the application. As described before, the framework of this very simple example consists of a single class (TestCase) which is only customizable by inheritance. The application (MoneyTest) customizes the framework by providing implementations for the abstract hook methods setUp, runTest and tearDown.

The flow starts in the main method of the application's code. A MoneyTest object is created, and its run method is called. Due to the mechanism of dynamic binding, the run method selected to be executed is the one implemented in TestCase, the superclass of MoneyTest, and thereby the control flow is transferred to the framework. The run method then starts and calls the setUp method, declared as abstract in TestCase and implemented in MoneyTest. Now, the dynamic binding mechanism selects to be executed the setUp method implemented in MoneyTest, and thereby the control flow is returned back to the application. When the setUp method terminates, the control flow turns back to the framework, and then to the application again in order to execute the runTest method implemented in MoneyTest, and so on until the end of the main method.

The Figure 2.6 graphically describes how the control flow have oscillated back-and-forth from the application to the framework, until the moment of calling the runTest method.

The mechanism used by this framework to call application-specific code relies on deriving application-specific classes (MoneyTest) from the base classes provided by the framework (TestCase), and on overriding their methods (setUp, runTest, tearDown).

While this customization mechanism focus on inheritance, there are other mechanisms that rely on composition. Both kinds of mechanisms have advantages and drawbacks. The most significant difference is on how they trade-off flexibility of customization with run time adaptability. Inheritance based mechanisms offer a good extension flexibility but don't support adaptation at run time. Composition based mechanisms requires explicit definition of points of flexibility but support adaptation at run time.

Inheritance and composition based mechanisms lead to two broad categories of frameworks: black-box and white-box frameworks.
2.4.4 Classifying Frameworks

Frameworks are typically classified according to the extension techniques provided and their scope of work.

White-box and black-box frameworks

Based on the extension techniques provided, frameworks can be classified in a range along a continuum from white-box frameworks to black-box frameworks [Johnson and Foote, 1988], as illustrated in Figure 2.7.

White-box frameworks rely heavily on inheritance and dynamic binding in order to achieve extensibility. Although white-box reuse is the hardest way to use a framework, it is by far the most powerful.

Black-box frameworks are the easiest to use, because they are structured using object composition and delegation rather than inheritance. On the other hand, black-box frameworks are the most difficult to develop, because they require the definition of the right interfaces and hooks able to anticipate a wide range of application requirements.

Most real-world frameworks combine black-box and white-box characteristics, being thus called gray-box frameworks. They allow extensibility both by using inheritance and dynamic binding, as well as by defining interfaces. Gray-box frameworks are designed to avoid the disadvantages of black-box frameworks and white-box frameworks.
In addition to the classification above, frameworks can also be classified according to their scope of work. In [Fayad and Schmidt, 1997b] is proposed a classification for frameworks based on their scope which consists of three categories: system infrastructure frameworks, middleware integration frameworks, and enterprise application frameworks [Fayad and Schmidt, 1997b].

System infrastructure frameworks aim to simplify the development and support of system infrastructure areas such as operating systems, user interfaces, communications, and language processing. Graphical user interface (GUI) frameworks, Java Foundation Classes (JFC), Microsoft Foundation Classes (MFC), or MacApp, are examples of frameworks used as underlying frameworks for other applications.

Middleware integration frameworks are usually used to integrate distributed applications and components. Examples of middleware integration frameworks include ORB frameworks, message-oriented middleware, and transactional databases.

Enterprise application frameworks address large application domains, such as telecommunications, banking, or manufacturing, and can provide a substantial return on investment as they support directly the development of end-user applications. A famous example of an enterprise framework is the IBM SanFrancisco Project.

These kinds of frameworks are related, as they layer up on top of each other. Middleware integration frameworks usually includes a system infrastructure in its underlying layer. Similarly, an enterprise framework includes both a middleware integration framework and a system infrastructure in the underlying layers.

### 2.5 History of Frameworks

Although the framework concept reached popularity recently (1990s), the history of frameworks dates back to the 1960s. The first examples of the framework concept found in the literature were designed to solve mathematical problems in Simula (1960s) and Smalltalk (1970s).
2.5.1 Early frameworks

The Simula programming language, created more than 30 years ago (1967), has precipitated the invention of the concepts of object-oriented programming. Simula is particularly important for framework technology because it was specifically designed to support frameworks, or application-oriented extensions, as they were then called. Simula was designed as a minimal addition to Algol, extending it with the basic concepts of object-oriented programming: objects, classes, inheritance, virtual methods, references, and a type system. The object concepts first introduced by Simula have percolated into most current object-oriented languages, such as C++ or Java.

It is generally accepted that the most significant distinction between a framework and a mere class library of classes depends on the presence of inverted control. In other words, the possibility that code in the framework may call code in the user part. In primitive languages this is implemented with callbacks, that is, procedure parameters. In most object-oriented languages, invocation of control is achieved through virtual procedures, which are declared and invoked by the framework code, but whose implementations can be redefined by the user code. Simula has virtual procedures, but also has an inner mechanism, which has the same characteristics of a framework calling user code. Beta [Madsen et al., 1993] is the only other language also having this mechanism [Hedin and Knudsen, 1999].

Simula provides a standard library containing two frameworks, Simset for list handling and the simulation for discrete-event simulation. Each framework consists of a single packaging class that contains all the component classes, procedures, and variables. An application program obtains these capabilities by using the framework name as a prefix to the program. Simset framework implements two-way circular lists. Simulation is a framework that allow the language to handle the discrete event Simulation [Birtwistle, 1979]. By means of these two object-oriented frameworks, Simula provides superb facilities for simulation, namely pseudo-parallelism, real time abilities, and simulation of complex systems. In addition, with Simula it is particularly easy to combine quite diverse frameworks.

2.5.2 GUI frameworks

In the late 1970s, the emerging interactive paradigm of Graphical User Interfaces (GUI) based systems made windows and events to stand up as a new challenging domain for programmers, for which they need help to write software.

The difficulty of coding GUI applications directly on top of the complex
procedural application programming interfaces (APIs) provided by the most popular GUI systems (Macintosh, X-Window System, and Microsoft Windows) started a growing demand for finding better ways of developing software solutions.

The Smalltalk-80 user interface framework, named Model-View-Controller (MVC) and developed in the late 1970s, was perhaps the first widely used framework [Goldberg, 1984; Krasner and Pope, 1988]. MVC showed at that time (and continues showing today) that object-oriented programming is well suited for implementing GUIs. MVC divides an user interface into three kinds of components working in trios: a view and a controller interacting with a model.

One of the first user interface frameworks influenced by MVC was MacApp, which was developed by Apple Inc. to support the implementation of Macintosh applications [Schmucker, 1986]. MacApp was followed by user interface frameworks from universities, such as Interviews from Stanford University [Linton et al., 1989], and ET++ from the University of Zurich [Weinand et al., 1989].

MacApp, InterViews, and ET++ became very popular during the 1980s. These frameworks provided useful, generic abstractions for drawing views and windows, and offered an event-handling mechanism based on the MVC concept. Most importantly, with any of these frameworks, the writing of an application became much easier, and resulted in a more stable code base, than directly using the APIs provided by the respective GUI systems.

But frameworks are not limited to user interfaces, being applicable to basically any area of software design. They have been applied to the domains of operating systems [Russo, 1990], very large scale integration (VLSI) routing algorithms [Gossain, 1990], hypermedia systems [Meyrowitz, 1986], structured drawing editors [Vlissides and Linton, 1990; Beck and Johnson, 1994], network protocol software [Hueni et al., 1995], and manufacturing control [Schmidt, 1995], to mention a few.

2.5.3 Taligent frameworks

In 1992, Apple and IBM have founded Taligent as a joint venture, which was joined by Hewlett-Packard in 1994. Taligent goal was to develop a fully object-oriented operating system and portable application environment, which shipped in July 1995 as the CommonPoint Application System. CommonPoint was a set of tools for rapid application development consisting of more than a hundred small object-oriented frameworks [Andert, 1994; Cotter and Potel, 1995] running on top of OS/2, Windows NT, AIX, HP-UX, and a new Apple OS kernel.
CommonPoint was most similar in scope and portability to Sun's subsequent Java environment, but based on C++ and without a virtual machine and a new object programming language (Java). The CommonPoint development environment was a visual component-based incremental development environment akin to the now-familiar IBM's VisualAge or Borland's JBuilder IDE's. The CommonPoint user interface paradigm known as "People, Places, and Things", extended the personal computer desktop metaphor to collaborative, distributed, task-centered workspaces that anticipated today's web-based environments. In terms of framework technology, Taligent's approach for CommonPoint made a shift in focus away from large monolithic frameworks to many fine-grained integrated frameworks.

In 1996, IBM took over sole ownership of Taligent, and in 1998 formally merged Taligent into IBM. During these two years, Taligent was an important center for object technology, providing key software components to IBM development tools, and licensed other key Java and C++ technologies to other industry partners, such as Sun, Netscape, Oracle and others. After 1998, Taligent engineering teams continued their development of object technologies and products at IBM.

### 2.5.4 Frameworks today (2000s)

The influence of the new GUI frameworks and the Taligent's innovative technological approach have attained a lot of interest to the framework concept, and both widely promoted frameworks in larger communities.

At present, in the 2000s, frameworks are important and are becoming even more important as software systems increase in size and complexity. Component systems such as OLE, OpenDoc, and Beans, are frameworks that solve standard problems of building compound documents and other composite objects. Frameworks like Microsoft Foundation Classes (MFCs), many parts of Sun's Java Development Kits (AWT, Swing, RMI, JavaBeans, etc.), implementations of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA), IBM's WebSphere, SanFrancisco, Apache's frameworks (Struts, Turbine, Avalon, etc.), Eclipse framework for integrated development environments, and JUnit testing framework, are all very important in contemporary software development. Since its appearance in 1995, Sun's Java has been one of the most successful, innovative and evolving language and frameworks (JavaBeans, Java Foundation Classes, Enterprise JavaBeans components, JavaOS, etc.). Java supports many platforms, from very small ones as smartcards, thin and thick clients to large mainframe installations. Java 2 Enterprise Edition (J2EE) has become one of the most successful frameworks for the web and enterprise technology.
In 2001, a new framework called .NET has emerged from Microsoft. It has many similar features to J2EE and will probably be one of the closest competitors of J2EE. Although both frameworks stand on a same foundation of programming languages, object models and virtual machines, they are different when considering the design goals of their runtime environment, namely the portability of code to different platforms: .NET uses a common intermediate language, and J2EE uses bytecode for a virtual machine. These two dominating frameworks promise to compete very closely in the next few years to come.
3

Building, Using and Documenting Frameworks

Frameworks represent partial-to-complete solutions to a particular problem domain. Although frameworks can be used exactly as they were created, in most of the cases they need to be extended or customized to fit the specific problems at hands. The inherent complexity of frameworks usually make them difficult to master by first time users, specially when frameworks are poorly documented. As a result, good quality documentation is a crucial factor for effective framework reuse, but it is also one of the most difficult to formalize.

This chapter describes the major activities, techniques and issues related with building, using, and documenting frameworks, in the context of framework-based application development. In addition, the chapter presents the specificities of framework documentation, and reviews the different techniques and types of documents used with frameworks. The chapter concludes with an overview of the main benefits of frameworks, and ways of maximizing them.
3.1 Framework-Based Application Development

The introduction of reuse in a software development process implies splitting traditional software life cycle into two new interrelated life cycles: one focused on developing reusable assets, and another focused on searching and reusing the reusable assets already developed.

In the particular case of framework-based application development, the traditional software life cycle can be organized in:

- a framework development life cycle devoted to build frameworks, corresponding to the abstraction phase of software reuse (Section 2.2.1);

- and an application development life cycle (also known as framework usage) devoted to develop applications based on frameworks, corresponding to the selection, specialization, and integration phases of software reuse.

Although the activities of framework development and application development are separate and often assigned to different teams, they must be closely associated, as the design of a framework for a domain requires past experience in designing applications for that domain. Figure 3.1 depicts both the traditional process and the framework-based process for application development.

![Diagram](image)

**Figure 3.1** Traditional versus framework-based application development.

Framework development is often the most effort-consuming phase of framework-based application development, being slightly different from the development of an ordinary application. While an application is only
concerned with the concepts mentioned in the application requirements, the framework has to cover all the relevant concepts from a domain.

In application development, frameworks act as generative artifacts, in the sense that they are used as a foundation for several applications of the framework's domain. This contrasts with the traditional way of developing applications, where each application is developed from scratch.

The most distinctive difference between the traditional and the framework-based development of applications is the need to map the structure of the problem to be solved onto the structure of the framework, thereby forcing the application to reuse the design of the framework. The positive side of this is that we don't need to design the application from scratch. But, on the other hand, before starting application development we need to understand the framework design, a task that sometimes can be difficult and time-consuming, specially if the framework is not appropriately accompanied with good documentation and training material.

Figure 3.2 Activities, artifacts and roles of framework-based application development.

Figure 3.2 shows a simplified view of framework-based application development that relates the artifacts, activities, and roles most relevant for this dissertation.

3.1.1 Artifacts

The key artifacts of framework-based application development are the following: the (problem) domain, the framework, applications and the framework
documentation. Of these artifacts, documentation is the only fully tangible, while the others can be more or less abstract. The framework is here explicitly distinguished from its documentation: while the framework comprises all the ideas and concepts involved in the framework as a software product, its documentation makes explicit the design and implementation aspects that are most important for developers who try to understand it.

3.1.2 Activities

Artifacts are related by activities. The framework is designed and implemented by abstracting the concepts and functionality of the problem domain. Documenting the main aspects of the framework leads to its documentation. After designed and implemented, the framework is understood by application developers, using its documentation. Finally, the framework is applied by application developers to build concrete applications.

3.1.3 Roles

Different activities are assigned to different roles: framework developers design (and implement), refine and document the framework, based on their knowledge about the problem domain; application developers try to understand the framework and use it as a foundation for building concrete applications.

The following sections will review in more detail the major activities, issues and techniques related with building, using and documenting frameworks, the three most distinctive sets of activities of framework-based application development.

3.2 Building Frameworks

The development of frameworks is similar to the development of most reusable software [Krueger, 1992; Tracz, 1995], but somewhat different from building an ordinary application.

Framework development usually starts with a deep analysis of the problem domain to collect knowledge and examples about the domain. The domain analysis constitutes the basis for the following phases of framework design and implementation.

Framework design consists on abstracting the concepts and functionality captured during domain analysis, on identifying the parts most likely to
change and to implement them in a flexible way, so that they can be easily applied to concrete applications. The abstracting process and the need to anticipate the needs of future applications implies large domain experience, considerable design skill and is best achieved through iteration.

Because framework development requires iteration and a deep understanding of the problem domain, it is often the most effort-consuming phase of framework-based application development.

### 3.2.1 Typical Activities of Building Frameworks

The development of a framework comprises the activities of domain analysis, framework design (and implementation), framework testing, framework documentation and framework evolution (see Figure 3.3) [Taligent Press, 1995; Taligent Press, 1994; Fayad et al., 1999].

![Diagram of framework development activities](image)

**Figure 3.3** Typical activities of framework development.

**Domain analysis**

Domain analysis aims at describing the domain to be covered by the framework. It consists on realizing a broad and extensive analysis in order to capture the present and possible future requirements of the complete problem domain. This activity is usually performed using systematic analysis of the domain, previous experience of developing applications in the domain, knowledge of domain experts, or existing domain standards [Bosch et al., 1999]. The result is a domain analysis model containing the common requirements, domain concepts and their interrelations [Schafer et al., 1994].

**Design and implementation**

Framework design and implementation is usually the most effort-consuming activity. It encompasses the life of the framework until it is released and used in the first real application. The goal with framework design is to end up with a flexible framework that explicitly divides the set of stable interfaces and behavior from the variable parts intended to be customized by final applications.

The first version of the framework is developed using the key abstractions
found in the domain analysis. It starts by identifying the primary abstractions (e.g. classes and design patterns) that will enable application developers to produce valid solutions for their concrete problems.

To help on the customization process, the framework is designed for flexibility, extensibility and ease of use, with the goals of reducing the amount of client code and client errors through the use of simple but well-defined interactions between client code and framework classes.

Customization is supported at points of predefined refinement, called hot spots, using general object-oriented abstraction techniques, such as, abstract classes, polymorphism and dynamic binding. Each hot spot is implemented by a hot-spot subsystem that contains base classes, concrete derived classes and possibly additional classes and relationships.

The hot-spot subsystem introduces variability either by inheritance or by composition (Section 2.4.2). 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. Below, Figure 3.4 shows both kinds of hot-spot subsystems.

![Inheritance-based hot-spot subsystem](image1)

![Composition-based hot-spot subsystem](image2)

**Figure 3.4** Inheritance-based and composition-based hot-spot subsystems.
As the framework takes shape, it becomes more clear the separation of the domain aspects requiring variability and flexibility, introduced with hot spots, from the fixed aspects of the domain. After introducing the variability required, the framework design should be continually refined to try to make it simpler, smaller and easier to use by clients.

Testing

Framework testing is devoted to evaluate not only the correctness and completeness of the intended functionality, but also the (re)usability of the framework.

The best way to evaluate the (re)usability of software is to (re)use it [Johnson and Russo, 1991]. For frameworks, this means to develop applications using the framework. Framework testing consists on the development of test applications based on the framework, where different aspects of the framework can be tested. The goal of framework testing is to evaluate whether it is really reusable, and sufficiently mature to be released, or if it still needs further refinements.

Documenting

Documenting a framework is one of the most important activities in framework development, although its importance is not always recognized. The effectiveness of framework reuse depends significantly on the quality of the accompanying documentation, so this activity deserves more attention than usually has. The documentation of a framework should include not only the documentation of low-level classes and methods, but also documentation of high-level design details, usage guidelines, and application examples that help illustrate the most important aspects of the framework. Such high-level design documentation improves framework understanding because it provides the programmer with hints to structures eventually familiar that enable him to do a top-down understanding in addition to bottom-up understanding of code, thus allowing for a much faster comprehension of the whole framework [von Mayrhauser and Lang, 1999]. Although intrinsically related with low-level code documentation, the high-level documentation doesn’t fit entirely on code, and thus often requires external documents.

The result of this activity usually consists of a user manual describing the purpose of the framework and how to use it, design documents describing how the framework works, and examples of applications developed with the framework.

Evolution

Framework evolution is a mandatory activity of framework development. It is considered to start immediately after releasing its first version. As frameworks must be seen as a medium to long-term investment, they need to be properly maintained. Early from the beginning of the life cycle, frameworks require several design iterations in order to ensure its reusability.
Several low-level design changes, or refactorings [Opdyke, 1992], are very common before the first release, specially in inheritance hierarchies and component hierarchies. These many design refactorings can consume a considerable amount of tedious work if not done with appropriate tool support.

As happens with all software, frameworks are subject to change, due to changes in requirements, changes in the problem domain, or errors reported from released applications. The most important tasks of framework evolution are thus the support of new requirements, design refactorings and correction of errors, while preserving backward compatibility.

3.2.2 Approaches for Building Frameworks

Despite the considerable number of successful frameworks developed during the last decade, designing a high-quality framework is a difficult task.

One of the most common observations about framework development is that it requires iteration to produce successful frameworks: “Good frameworks are usually the result of many design iterations and a lot of hard work” [Wirfs-Brock and Johnson, 1990].

Iteration in framework development is mainly due to three reasons. One reason is that finding the correct abstractions in immature domains is very hard. This difficulty often results in mistakes being discovered only when the framework is used, which then leads to refinements and iteration. Another reason for iterating is that the flexibility provided by a framework is difficult to fine-tune a priori, and its evaluation usually requires to experiment it in concrete situations. A third reason is that frameworks are high-level abstractions, and thus their generality and reusability is highly dependent on the original examples used during the abstraction process.

Despite the importance of iteration in framework refinement, it would be beneficial to have processes able to reduce the number of design iterations and the effort spent on framework refinement.

The lack of mature processes for developing frameworks is an issue. While framework development processes get more and more mature, framework development is considered (by some) as an art, requiring both experience and experimentation [Johnson and Foote, 1988].

Several methods have been proposed in the literature to support framework development in order to reduce the refinement effort and make it more predictable. Some examples of methods are: TaliGen's method [TaliGen Press, 1994]; a pattern language for evolving frameworks [Roberts and Johnson, 1997]; systematic framework design by generalization [Schmid,
1997]; hot-spot-driven development [Preece, 1999]; the catalysis approach [D'Souza and Wills, 1999]; role-modeling approach for framework design [Richle, 2000]; and the approach for deriving frameworks from domain knowledge [Aksit, 1999].

All these approaches define similar activities for developing frameworks but differ on the emphasis put on some activities and artifacts used as starting points.

In the context of this work it is relevant to review at least two of these approaches, to understand the differences between two possible extremes of framework development methods: the top-down and bottom-up approaches. The methods chosen to present here are the following:

- the pattern language for evolving frameworks [Roberts and Johnson, 1997] suggests starting from concrete applications and abstract the framework using iterative refinements, thus emphasizing a bottom-up approach;

- the approach for deriving frameworks from domain knowledge [Aksit, 1999] suggests starting not from concrete applications, but from domain knowledge, thus emphasizing a top-down approach.

[Roberts and Johnson, 1997] proposes a pattern language to evolve a framework. The method assumes that primary abstractions are very hard to find, and therefore considers that several development efforts are required to achieve a successful framework. The method suggests to start from concrete applications and iteratively abstract and refine the framework from those applications.

This method written in the form of a pattern language describes a common path that frameworks usually take, but it mentions that is not necessary to follow the path to the end to have a viable framework (Figure 3.5). Each step of the path corresponds to a pattern of the pattern language. Each pattern describes the problem, trade-offs involved, and a generic solution. A summary of the patterns is presented.

**Three Examples.** Due to the high difficulty of developing a reusable framework by simply thinking about the problem domain, even for very wise experts, the method suggests to start developing three applications aimed to be derivable from the framework to build. Build a first application, a second slightly different from the first, and then a third even more different than the first two. The most important common abstractions will then become apparent and the resulting framework will be useful and able to evolve, although not yet done.
Figure 3.5 Evolving Frameworks pattern language [Roberts and Johnson, 1997].

**White-Box Framework.** After building the first application, it is not possible yet to anticipate which parts of the framework should be fixed or variable. So, start with a framework that rely on inheritance (easy to change) instead of polymorphic composition (requires knowing what is going to change) by generalizing from the classes of the individual applications. Later, encapsulate and parameterize the immutable code and convert the framework into a black-box framework.

**Component Library.** To avoid writing similar objects for each instantiation of the framework, start with a simple library of the obvious objects and add new objects as needed.

**Hot Spots.** To eliminate common code from the applications developed with the framework separate code that changes (hot spots) from code that remains stable. Hot spots can be implemented using design patterns, which are able to encapsulate various type of changes [1; Buschmann et al., 1996].

**Pluggable Objects.** To avoid having to create trivial subclasses each time you want to use the framework design adaptable classes that can be parameterized to cope with what distinguishes one trivial class from another.

**Fine-Grained Objects.** To make the Component Library more reusable the objects should be divided into finer granularities until it doesn’t make sense to continue any further.

**Black-Box Framework.** As the objects become more fine-grained the framework become more Black-Box. Use inheritance to organize the component library and composition to combine the components into applications. As composition enables the reuse of components by simply plugging them
together, composition should be favored when it isn’t clear which technique is better.

**Visual Builder.** With Black-Box frameworks, entire applications can be made only by connecting objects of existing classes. To simplify the creation of applications it can be provided a graphical program that enables a visual specification of the application objects and their interconnections, and automatically generates the application code.

**Language Tools.** To enable an easy inspection and debugging of the specifications produced by the visual builders it is useful to provide specialized language tools, such as inspectors and debuggers.

[Askit, 1999] proposes an approach to develop frameworks that aims at reducing the amount of refinement time by deriving frameworks directly from domain knowledge.

The approach suggests to start by explicitly modelling the domain knowledge related with the framework. The identified domain models should then be mapped into object-oriented frameworks using standard object-oriented models, design patterns, delegation and message reflection techniques that help increase software adaptability and reusability.

**Modeling domain knowledge.** The top-level structure of the framework is modelled using knowledge graphs [Bakker, 1987]. To find this graph requires searching related literature and sources of information. Each node represents an indispensable concept for the framework. The relations represent direct dependencies between the nodes.

Each node of the top-level knowledge graph is then refined into a sub-knowledge graph called a knowledge domain. The nodes within a knowledge domain correspond to a particular specialization in the domain and the relations typically represent generalization and specialization relations. In this approach, a framework is considered a composition of specializations from related knowledge domains.

The nodes in a knowledge domain that can be included together into the top-level knowledge graph are identified, and the scope of the framework is restricted by adding new user-defined constraints. All together, these constraints define the adaptability space of the framework.

**Mapping domain knowledge to frameworks.** The second part of the approach consists on mapping the knowledge graphs to object-oriented frameworks. Although conceptually clean, this mapping has costs and poses several problems that force the representation of some elements directly in the implementations of object operations instead of using explicit representations. Most of these
problems can be attributed to the degree of expressiveness of the object-oriented model, and can be summarized by the following items:

- specific inheritance semantics are necessary for certain knowledge domains;
- conditional delegation is needed;
- enforcing constraints is essential;
- improvement of object-composition techniques;
- software artifacts must be recorded, related, and integrated.

Despite the difficulties, the approach helps reduce the number of refinement steps when developing frameworks directly from domain knowledge.

In summary, both kinds of approaches (top-down versus bottom-up) pose several difficulties to the present state of the art of object-oriented frameworks technology, and future research is needed to find better solutions for the most common issues of framework development.

3.2.3 Key Issues of Building Frameworks

Building a successful framework is a difficult task that requires highly skilled developers, deep knowledge of the problem domain, and investment in time and effort. In addition, there are several issues related with framework development activities that must be solved, which range from domain analysis to framework maintenance and evolution. Some of the most relevant issues in the context of this work will now be described.

Scoping the framework domain

Finding the right scope for the domain covered by the framework is the first difficulty of framework development. It should be neither too large, to reduce the complexity of the framework, nor too narrow, to be useful for a wide family of applications for that domain. Defining the right scope of the domain is thus an important decision that requires careful balancing. The most suitable frameworks reflect a mature, yet narrow, focus on a particular problem or group of problems.

Mainly due to the increasing popularity of frameworks, there is a growing need for finding appropriate development methods capable of reducing the high effort typically required to develop frameworks. If developing complex software is hard enough, developing high-quality, extensible and reusable frameworks for complex domains is much harder, requiring skills only available in expert developers. Such methods must put more emphasis on the concepts mostly used in framework development: abstract classes, dynamic binding, type parameterization, hot spots, pre and postconditions.
The definition of methods that help developers produce successful frameworks is crucial for a more generalized use of frameworks.

Testing frameworks is much harder than testing traditional object-oriented applications. The main difficulty resides in the fact that it is much harder to test abstract and generic classes without instantiating them first, than to test concrete classes. Since the framework relies on parts implemented by the users it is not possible to completely test the framework before it is released. Therefore, to evaluate the correctness, completeness, and usability of a framework, it is usually required more effort than for testing traditional applications.

The production of quality framework documentation is often hard and tiresome. Documentation must communicate framework design information, mainly during framework development and evolution, to framework developers and maintainers. In addition, documentation must also provide information on how to use the framework, mainly for usage during framework instantiation by framework users. As a result, framework documentation must include a lot of information contained in different software artifacts produced during the development process, from domain analysis to coding. These artifacts are typically stored in different formats, from informal text to executable code.

One of the issues here is to find methods able to produce and manage (record, trace, relate and integrate) rich and effective framework documentation, while being simple and economical, both to use and understand.

A framework is considered ready for release if it is reusable, reasonably stable within the domain, and well documented. All these aspects are difficult to measure: there are no metrics for evaluating reusability, in general; the stability of the framework domain is not possible to ensure, as the problem domain could be unstable itself, or not clearly scoped; finally, it is difficult to evaluate if a framework is well documented or not, because such evaluation should check if the documentation covers all aspects of the framework and is understandable for the intended users.

On the other hand, releasing an immature framework may have a severe negative impact in the maintenance and usage of the framework and instantiated applications.

The selection of adequate strategies for framework maintenance and evolution is very important for coping with the problem of changes in requirements. Application requirements change frequently, and thus framework requirements change too. As frameworks invariably evolve, due to external changes or internal refinements, the applications built with them
must evolve as well.

Therefore, framework changes force the organization to face the problem of selecting adequate maintenance strategies that perfectly balance different factors such as, time pressure, existing and expected future applications, and estimated lifetime for the software involved.

### 3.3 Using Frameworks

In framework-based application development, frameworks must be used as a kind of starting foundation for the intended applications. Although based on similar concepts and mechanisms, developing applications using frameworks is quite different from developing applications using class libraries, at least in two important aspects: frameworks (usually) take the control of the runtime behavior of the applications developed, which is not the case when using class libraries; and the framework imposes the overall structure of the application whereas a class library has minor impact on its structure.

#### 3.3.1 Typical Activities of Using Frameworks

The development of an application using frameworks usually starts with a requirements analysis and a high-level design (Figure 3.6). Based on the application requirements and the perceived appropriateness of a set of frameworks to satisfy those requirements, the developer selects one or more frameworks to be reused (framework selection).

![Figure 3.6](image-url)  
**Figure 3.6** Key activities of application development using frameworks.

After being selected, the frameworks are composed (framework composition) and customized to fit all the application requirements. The customization usually consists on deducing, designing and implementing application-specific extensions to the framework (framework instantiation).
After integrating the frameworks with their application-specific extensions, and any other application code (application implementation), the overall application must be tested and validated to ensure that all original requirements are satisfied.

From all the typical activities of using frameworks, the one with higher impact on the effectiveness of framework reuse is framework instantiation. The difficulty of framework instantiation is strongly dependent on how easy it is to learn, understand and use the framework, a set of properties directly dependent on the quality of the accompanying documentation and the extensibility technique provided for customization (inheritance or composition).

Requirements analysis

The goal of the requirement analysis activity, both in traditional and in framework-based application development, is to collect and analyze the requirements of the application to be built using frameworks.

High-level design

Based on the requirements collected, it must be defined a high-level design for the application. In other words, the components and the organization of collaborations between them must be defined.

Framework selection

Considering the architecture of the application, the functionality to be provided by the application, and other non-functional requirements, one or more frameworks are selected to be reused. The selection of frameworks is done based on the evaluation the application developer does about the appropriateness of a particular framework for the application under consideration. This selection and applicability evaluation usually requires a quick understanding of the frameworks under consideration.

Framework composition

Traditionally, the development of applications using frameworks starts from a single framework, and the application is constructed only in terms of extensions to that framework. However, as applications become more complex, they are increasingly requiring the composition of multiple frameworks. When more than one framework needs to be reused for the development of an application, several composition and integration problems can occur, due to architectural mismatch, overlap of framework entities, or other difficulties related with the control and coordination of multiple event loops [Mattsson and Bosch, 1999].

Framework instantiation

Once selected, a framework must be customized to fit the specific requirements of the application at hands. The customization consists on deducing, designing and implementing application-specific extensions required by the application. Applications developed using frameworks typically have three parts: the framework, the concrete subclasses of the framework classes, and everything else, which usually includes some kind of configuration specifying which concrete classes are used, and how they must
be interconnected. The customization of a framework to a particular application is usually possible through subclassing of framework abstract classes and/or composition of concrete classes.

After being specified, the application-specific extensions must be integrated, by following the design rules and constraints stated in the framework documentation. The result of this activity is an application constituted by three main kinds of code: framework code, code that implements the application-specific framework extensions, and typical application code.

Application testing consists both on individually testing the application-specific extensions (not always possible), before integrating them with the rest of the application, and on testing the global application to verify if the complete application fulfills the original application requirements.

### 3.3.2 Techniques for Using Frameworks

Black-box frameworks and white-box frameworks use different extension techniques for customization, namely composition and inheritance based mechanisms, which require different knowledge from framework users.

**Using black-box frameworks**

Black-box frameworks are the easiest to use, because they are structured using object composition and delegation rather than inheritance. They support extensibility by defining interfaces for components, which can be plugged into the frameworks via object composition. To be customized, they only require the instantiation and connection of components selected from a library provided with the framework, which releases the framework user from having to look inside the details of component implementations. Because the connection of components is achieved through composition, it can be changed dynamically, at runtime.

Black-box frameworks can be made even easier to use, if accompanied with an application builder (visual or not) or domain-specific language tools that can automate the instantiation and connection of components.

Although being the easiest to use, black-box frameworks are the most difficult to develop, because they require the definition of the right interfaces and hooks able to anticipate a wide range of application requirements.

**Using white-box frameworks**

Sometimes, the library of components provided with the framework doesn’t fit all the application requirements, and the framework user needs to create new components.

Inheritance is the most expedient way of changing the behavior of object-oriented code. White-box frameworks rely heavily on inheritance and dynamic binding in order to achieve extensibility. New components can thus
be achieved by subclassing framework abstract classes and overriding predefined operations, which requires intimate knowledge about the framework internals. This is considered the second easiest way to use a framework.

The way of using a framework that requires the most knowledge from the framework user is the extension of a framework by changing the abstract classes that form its core. To use frameworks in such manner can be very difficult and complex, as it usually involves adding new operations or variables directly in the source code, which requires previous understanding of at least some parts of the framework code.

Despite being the hardest to use, white-box frameworks are by far the most powerful.

But pure white-box or black-box frameworks are very rare. Most real-world frameworks combine black-box and white-box characteristics, being thus called *gray-box* frameworks. They allow extensibility both by using inheritance and dynamic binding, as well as by defining interfaces. Gray-box frameworks are designed to avoid the disadvantages of black-box frameworks and white-box frameworks. They have enough extensibility and flexibility, and have also the ability to hide unnecessary information from the application developers. The hiding of selected code leads to the abstraction of the static parts of the framework and as a result protects the core structure from being arbitrarily changed [Yassin and Fayad, 1999].

### 3.3.3 Key Issues of Using Frameworks

Once built and delivered, a framework is supposed to be used as a foundation for the development of concrete applications. The key issues related with the development of applications based on frameworks are described below.

**Applicability evaluation**

To evaluate the applicability of a framework is one initial difficulty of using frameworks. It happens when the application developer is in the phase of selecting which frameworks to use for the development of the application at hands. The issue here is on determining the domain covered by the framework, and on evaluating how appropriate the framework could be for the development of the intended application.

**Framework learning**

Learning and understanding a framework usually constitutes a major obstacle to the effective reuse of frameworks. Before starting to use a framework successfully, users usually need to spend a lot of effort understanding its underlying architecture and design principles. The typical learning curve of frameworks is steep because developers have to
understand not only single isolated classes, but also complex designs of several classes whose instances collaborate for many different purposes, possibly using many different mechanisms. Object-oriented frameworks require considerable effort to be understood, as a result of being full of de-localized plans [Soloway et al., 1988] due to the intensive use of inheritance and delegation [Wilde et al., 1993]. How to obtain the necessary information from the framework itself and its accompanying documentation is the main problem with framework understanding.

Mature processes for developing applications based on frameworks are needed to enable effective project control and management. Whilst development processes for standard applications are well known and have undergone a considerable phase of evolution, processes for framework-based application development are still in the ad hoc stage, and only exploratory styles of development are used.

Validating and debugging applications built using frameworks can be tricky, because it is normally hard to distinguish between framework code bugs and application code bugs. In addition, due to the inversion of control typical of frameworks, applications written with frameworks have a flow of control that oscillates between the framework code and the application code (Section 2.4.3). This inverted flow of control increases the difficulty of single-stepping through the runtime behavior of a framework, as the application developers usually don't understand or have access to the framework code.

3.3.4 Main Benefits of Using Frameworks

The benefits from frameworks widespread many phases of application development, such as analysis, design, coding, testing, maintenance, and evolution. The main advantages of frameworks in each of these phases were summarized in Section 2.2.3, and are now briefly described below [Taligent Press, 1995; Fayad et al., 1999].

During the analysis phase, the use of frameworks for application development helps developers to reduce the effort usually required to understand the overall application domain, thereby enabling to focus on their application details.

Leverage domain expertise. A framework embodies domain expertise in the form of processes, rules, and policies for a particular area. This domain expertise usually encapsulates complex logic that would take a long time to develop from scratch. Frameworks take advantage of object technology in order to tie together this domain expertise in a form that is simultaneously flexible
and usable, avoiding the re-invention of well-known solutions to recurring application requirements.

**More focus on areas of expertise.** When using a framework, developers can focus on their area of expertise, typically the area where they can add the most value to the domain-specific application at hand. Frameworks facilitate development, by providing generic solutions that insulate domain-specific details from system related specificities. Frameworks let developers concentrate on understanding how to support the requirements of their particular problem domain, thereby providing an environment focused on solving domain problems instead of programming problems.

It is during designing, coding, testing, and debugging applications that frameworks have more advantages over traditional application development. Most of the benefits result from the high levels of design and code reuse provided, and also the inversion of control possible with frameworks.

**Provide architectural guidance.** A framework dictates the architecture of the applications built with it, by defining: its overall structure, the partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. Frameworks capture design decisions common to their application domain, thus emphasizing design reuse over code reuse. In addition, frameworks encourage better design of the code written by developers, by guiding developers through examples on how to be more effective with object technology. Applications developed with frameworks tend to be smaller, as well as easier to reuse and maintain.

**Less code to design and write.** Because frameworks already provide much of the application's design and structure, as well as its code, frameworks significantly reduce the amount of software to design, code, test, and debug. Typically, we need to design and write only the code required by the framework, and the code needed to override the default behavior of the framework in order to meet the requirements of the application at hand. As a result, the amount of code to write is typically a small fraction of the code required to create the same application from scratch. This reuse of design and code yields substantial reductions in the effort, time, and cost required to implement the application, thus improving programmer productivity, as well as enhancing the quality, reliability and interoperability of the resulting software.

**Improved modularity.** Frameworks enhance modularity by encapsulating volatile implementation details behind stable interfaces. Framework modularity helps improve software quality by localizing the impact of design and implementation changes. This localization reduces the effort required to understand and maintain existing software.
**Generic solutions reusable across many related problems.** The stable interfaces provided by frameworks enhance reusability by defining generic components that can be reapplied to create new applications. Developers used to work with frameworks tend to think in terms of generic solutions rather than special solutions. This shift in the development culture results in software that is reusable, flexible, and therefore more consistent, compatible and easier to integrate.

**Improved Integration and Interoperability.** Frameworks embody domain expertise, so problems are solved once and the business rules and design are used consistently. Because different applications based on a particular framework inherit not only code but also the same architecture and design, applications are more likely to work together in effective ways, consistently and reliably, resulting in systems better integrated from a user's point of view, while requiring less development effort to be compatible.

If the benefits of frameworks are considerably relevant during application development, they are not less important at maintenance and evolution phases. The reusability and extensibility possible with frameworks help to decrease the effort of maintenance and enable an orderly evolution.

**Reduced maintenance overhead.** For applications based on frameworks, changes made to the framework, such as, adding a new feature or fixing a bug, are automatically updated to the applications built with them. As changes are made only in one place, the chance of introducing additional errors in the rest of the code is minimized. Overall maintenance results easier, because the maintenance of the framework is amortized over many application specific parts. From constant reuses of the framework, more refinements of features are made and more bugs are fixed, resulting from this process a smooth but solid evolution of a very robust framework. New applications built using a framework will contain significant amount of mature code from the framework, plus a smaller amount of new code, thus resulting in a higher overall quality.

**Easier evolution.** While objects and class libraries provide interfaces for extending functionality at a fine-grained level, frameworks provide this flexibility at a higher level. A framework provides explicit hook methods that allow applications to extend its stable interfaces with variations required by concrete applications in a particular context. In this way, applications can be developed by using the framework as a starting point and writing smaller amounts of code to modify or extend the framework's behavior. These extensions can be added without sacrificing compatibility and interoperability because the interfaces are well defined. Framework extensibility is essential to ensure an orderly evolution of applications with new services and features.
Frameworks are a powerful reuse technique with many advantages over other techniques. As a framework (re)user, before being able to benefit from this power, we need to invest time on knowing, learning and understanding them. As a framework developer, we need to spend time on conveniently documenting them, if we want to deliver to (re)users the benefits they enclose.

3.4 Documenting Frameworks

Good quality documentation is crucial for the effective reuse of object-oriented frameworks. Without a clear, complete and precise documentation describing how to use the framework, how it is designed, and how it works, the framework will be particularly hard to understand and nearly impossible to use by software engineers not initially involved in its design.

One of the goals of developing applications based on frameworks is to leverage the expertise of core software designers. Typically, expert designers build frameworks to support less experienced application developers on the implementation of a family of applications. While framework designers have a deep knowledge of the application domain and long experience with software design, application developers are usually less experienced and less knowledgeable about the domain.

However, before being able to reuse a piece of software, a developer has to invest time on understanding and learning how to use it. The implicit complexity of frameworks makes them difficult to understand, specially by first time users, requiring a lengthy learning process on behalf of the developers. The difficulty of understanding frameworks is a serious inhibitor of effective framework reuse, which is mainly due to framework designs being usually very complex and hard to communicate:

- the design is very abstract, to factor out commonality;
- the design is incomplete, requiring additional classes to create a working application;
- the design provides more flexibility than the strictly needed by the application at hands;
- and the design is obscure, in the sense that it usually hides existing dependencies and interactions between classes [Butler, 1997].

Good documentation significantly improves the process of learning and understanding new frameworks. By guiding users on the customization
process and making explicit the framework design principles and details, effective documentation contribute to frameworks easier to reuse.

For all these reasons, framework documentation is one of the most important activities in framework development, although its importance is not always recognized.

The remainder of this section presents the typical activities of documenting frameworks. After a brief characterization of the specificities of framework documentation, it is presented a brief review of the most important types of documents used for frameworks, some general-purpose techniques to document software, and the key issues of documenting frameworks.

3.4.1 Typical Activities of Framework Documentation

Framework documentation is produced mainly during framework development, resulting in user manuals teaching how to use the framework, and design documents to explain how it works by describing its underlying design principles and mechanisms.

Once produced, framework documentation is then used and reviewed during all phases of framework development. It is probably at framework instantiation, during application development, that documentation is used in a more intensive way. It acts as a means of communicating important information from the original framework designers, primarily to framework users, but also to other framework designers and framework maintainers.

The incorporation of comments and feedback from readers is very important for improving the quality of future revisions of the documentation, so it is important to establish an effective bidirectional communication mechanism between documentation authors and readers.

As a framework evolves during the expected long life of the respective framework, the accompanying documentation must evolve as well, and therefore the maintenance of documentation is an activity to be taken in consideration during all framework's life.

The life cycle of framework documentation can be organized in five basic activities: configuration, production, organization, usage, and maintenance (Figure 3.7).

**Configuration**

The configuration consists on the setup of the technical infrastructure, if existent, devoted to support the activities of documentation. Usually it consists on defining template documents, on the setup of editors, integration mechanisms, storage mechanisms, and other tools eventually used to support documentation activities. Ideally, documentation configuration should
precede all other activities to avoid a disorganized production of documents.

Production
The production of contents is the main activity of documentation. It may include not only the writing of technical documents and elaboration of models, but also the integration and formatting of documents and source code. The exhaustive cross-referencing of all kinds of contents is a very important goal of this activity in order to support easy navigation in the contents.

Organization
The organization of contents is an activity that can be realized in parallel with the production. It consists on (re)defining the structure of the contents as they evolve along the project life, according to the needs of the users. This may require transformations of contents to fit the new structure, and the addition of meta-information to help its management. The main goal of this activity is to keep all the contents consistent, well structured, integrated, and easy to maintain.

Usage
The usage is the ultimate activity of documentation, the reason why it was produced and organized. To use documentation means not only to read contents in a presentation format, but also to browse, search, select, and navigate through the contents. Sometimes, the usage of documentation requires processing of contents (transformations, filtering, composition, etc.), in order to present them in a format convenient for the user.

Maintenance
Maintenance of contents basically consists on controlling the revision, evolution, and release of contents, while keeping track of their history, and ensuring a reliable storage and archiving.

Figure 3.7  Typical activities of framework documentation.
Figure 3.7 represents the typical activities of framework documentation described before, and the roles assigned to such activities. In addition to the roles of author and reader, we can find also the role of manager and the role of tools. The manager role must be assigned to the responsible for the configuration activity. The tools role must be assigned obviously to tools able to support the automation of the documentation process.

### 3.4.2 Classifying Framework Documentation

The documentation for an ordinary object-oriented application usually needs to be described at different levels of abstraction, and the contents tailored to different categories of software engineers. This is equally true for a class library or a framework. Additionally to all the aspects that are relevant to object-oriented applications and class libraries, the documentation of a framework must satisfy the needs of different audiences and to encompass several purposes. Frameworks can also be (re)used in several ways, thus it is important to appropriately reflect all this variety in the accompanying documentation.

To be effective, the overall documentation of a framework must encompass different purposes by combining a large diversity of contents, which must be tailored to the needs of different audiences. The documentation must explain [Johnson, 1992; Butler et al., 2000]:

- the purpose of the framework;
- how to use the framework;
- the detailed design of the framework;
- the purpose of the application examples.

To address all these needs requires useful, readable, structured, and systematic framework documentation, containing much more than source code.

**The purpose of the framework.** The first thing that the documentation must describe in a clear way is the domain covered by the framework, i.e. the application domain and the range of solutions for which the framework was designed and is applicable.

This information is of great value for potential users specially during the selection phase (Section 3.3.1) because it helps them to evaluate the appropriateness of the framework for the application at hands, and thus fundamentally the selection, or rejection.

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.

**How to use the framework.** The quality of how-to-use information is very important to effective framework reuse because the informal communication channels between framework developers and framework users are not available, in most of the cases. Therefore, all information has to be communicated in alternative ways and delivered with the framework, typically contained in different kinds of manuals, eventually using different media formats (documents, animations, videos).

To help application developers being effective in the reuse of the selected framework, ideally, the documentation should be able to conduct the readers directly to the points where it explains how to use the framework, not all of it, but only the parts strictly required to implement the specific features of the application at hands.

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). 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. To complicate even more, this perfect balance may vary with the reader of the documentation, and thus the "one size fits all" solution doesn't work here.

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. Considering that the main purpose of a framework is to reuse design, if it is well designed, then there must exist large parts of its design easy to reuse even without knowing them.

The first usage of a framework is always the hardest one because it may involve the learning of the problem domain and of the fundamental concepts of object-orientation, depending on the user's previous knowledge. It usually consists on picking a small application well suited for the framework, to study similar examples and copy them, and to use the documentation to learn how to implement individual features. To get a real sense of the control flow, it is interesting to use a debugger to single-step the application, but sometimes this is not easy to do due to the complexity of the framework.

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 requires that theory follows practice [Johnson, 1992]. After the first use, the framework user has a much better understanding of what the framework does and thus is more capable to understand how it works, i.e. to understand its internal design details.

The design of the framework. In addition to the framework purpose and usage instructions, the framework user also needs to understand the underlying principles or basic architecture of the framework in order to develop applications that conform to the framework.

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. Framework understanding is often a prerequisite for evaluating the applicability of a framework and the amount of adaptation required, and how to carry out the adaptation.

Framework design groups are often small and use different ad-hoc techniques to interchange information among the individuals in the team. Although informal communication is a convenient means to spread the knowledge about the developed framework, it doesn't ensure the correctness of the information, and is not efficient for larger groups.

But 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 includes not only the different classes of the framework, but also the strategic roles and collaborations of their instances, and rules and constraints, such as cardinality of framework objects, creation and destruction of static and dynamic framework objects, instantiation order, synchronization and performance issues.

As an example, a change in a framework may involve the understanding of different kinds of information, at varying levels of abstraction and detail, including:

- understanding the application domain;
- understanding the basic architecture in terms of the key abstractions and their static and dynamic relationships;
- determining where is most appropriate to make a change in an inheritance hierarchy;
identifying which methods must be overridden, may be overridden, and can't be overridden in the context of inheritance;

and determining the dynamic behavior of a class in the context of inheritance and polymorphism.

The amount of comprehension required to understand a framework is much greater than that required to understand a simple application. The code of a framework implicitly contains a lot of design decisions that are very difficult to uncover simply by program inspection. Unless explained, the rules and constraints defined by framework developers makes no sense and the framework is in risk of not being used as intended. The effective understanding of a framework requires the recovery of a lot of comprehension, which can be achieved with good documentation.

In summary, framework documentation must be able to preserve important design information and comprehension that would take too much effort to gather from scratch. The documentation must describe the internal details of the framework, to explain how it was designed and how it works, and to include any other information produced during framework development considered useful to transmit to users, developers and maintainers of frameworks.

As most of the research work on framework documentation has focused on documenting the design and architecture of frameworks, there are a lot of complementary approaches useful for documenting framework design. The biggest problem is on conveying this information in a concise form to the framework user, otherwise the problem becomes not how to understand frameworks but how to understand framework documentation.

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

The examples must be used to show what the framework is good for, and not for showing how to use the framework, or for explaining how the framework is designed.

A small, but representative, graded set of examples can serve as a live catalogue for the key features of the framework, constituting a perfect complement to all other purposes of documentation mentioned before: the domain covered, how-to-use information and design internals.

Because good examples help new users to get 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.

The documentation of many successful frameworks provide a lot of examples: MVC [Krasner and Pope, 1988], ET++ [Weinand et al., 1989], Java Swing [Gosling et al., 1996]. Unfortunately, for many frameworks, the source code of example applications is the first and only documentation provided to framework users, as they can be produced during framework development without extra effort, but despite their value, they are not sufficient.

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, having varying levels of experience, and therefore requiring different information.

There are five main kinds of framework users, each with different documentation requirements: framework selectors, application developers, framework developers, framework maintainers, and developers of other frameworks [Butler, 1997].

**Framework selector.** A framework selector is someone (manager, project leader, developer) who is responsible for deciding which frameworks to use in an application development project. Framework selectors will look for a short description of the framework's purpose, the domain covered, and an explanation of the most important features of the framework, possibly illustrated with examples.

**Application developer.** An application developer is a software engineer who is responsible for customizing a framework to produce the application at hands. In a first place, application developers want to identify which customizations are needed to produce the desired application, and to know how to implement such customizations, rather than to understand why it must be done that way.

The application developer needs prescriptive documentation able to guide her find out which hot spots must be used, which set of classes to subclass, which methods to override, and which objects to interconnect. It is expected that the application developer possibly is not knowledgeable on the application domain and not an experienced software developer.

**Framework developer.** A framework developer is a software engineer who is responsible for the design and implementation of a framework. Framework developers must have a good understanding of the overall architecture and its rationale, and the most detailed vision over the internals of the framework design, and the hot spots that support its flexibility. They simultaneously produce and consume information at several levels of abstraction, from a
high level of abstraction to a concrete level of detail.

Their needs in terms of documentation are very similar to those of framework maintainers.

**Framework maintainer.** A framework maintainer is a software engineer who is responsible for the maintenance and evolution of a framework. Usually, framework maintainers are the original framework developers, but this is not always the case. Framework maintainers must have a good understanding of the overall architecture and its rationale, the internals of the framework design, the application domain, and the hot spots that support the flexibility offered by the framework. The information needed must be described at several levels of abstraction, from a high level of abstraction to a concrete level of detail. It usually contains several kinds of artifacts ranging from architectural models and design patterns to abstract algorithms and concrete source code.

The documentation has to be descriptive, instead of prescriptive, because original framework designers can't predict how the framework might be extended in the future through additional flexibility on existing hot spots, or in additional hot spots. It is expected that the framework maintainers are both domain experts and software experts.

**Developer of another framework.** Framework developers usually study existing frameworks, even frameworks for another domain, to find ways of providing flexibility at the hot spots of the framework they are developing. They have special interest on information at a high level of abstraction, such as abstract solutions and design patterns.

The documentation requirements are similar to those of framework maintainers, except that they don't need the concrete details about the framework, but rather the abstract ideas. It is expected that framework developers are expert software designers but not necessarily domain experts for the framework they are mining for ideas.

From all these audiences, application developers represent the majority. Framework developers are also a very important audience in the context of this thesis work because they are simultaneously authors and intensive users of framework documentation.

Different audiences (re)use frameworks in different ways, according to their interests. In [Butler, 1997] are identified six categories of framework (re)use, each with their own demands on documentation, and requiring increasingly more knowledge about the structure and behavior of the framework.

These categories of framework reuse are intimately related with the kinds of framework reusers, i.e. the documentation audiences enumerated before.
Figure 3.8 shows these relations in a use case diagram, where the actors represent the documentation audiences, and the use cases represent categories of framework reuse.

Figure 3.8  Relating documentation audiences with categories of reuse.

A brief description of these categories is presented below. A more detailed description can be found in [Butler, 1997] and [Butler et al., 2000].

Selecting. Framework selection consists on deciding whether to reuse or not a framework, after evaluating its appropriateness for an intended application. The selection process requires matching the description of the problem at hands to the framework description. This matching may involve a simple matching of structures or a more complex matching of behaviors.

Instantiating. Framework instantiation for a particular application consists on customizing hot spots in a way planned by framework designers. Instantiating can be achieved by selecting concrete classes from an existing library to customize the hot spots to the needs of the application at hands, or by extending framework abstract classes in a way planned by framework designers. The instantiation requires matching of interfaces and behaviors, and writing code to implement new behaviors.

Flexing. Flexing a framework consists on customizing a hot spot in a way not initially planned by framework designers, i.e. the customization is consistent with the framework design, structurally but not semantically. This often involves abstracting the domain to a level higher than the original.

Composing. Framework composition consists on using two frameworks in conjunction, thus leading to the sharing of participants and messages.
Evolving. Evolving a framework consists on refining a framework either to increase the flexibility of existing hot spots, or to add new ones. This requires abstracting the domain and splitting existing participants into a fixed part and a variable part.

Mining. Mining a framework consists on identifying parts of a framework that might be useful to reuse in other contexts, such as other frameworks or applications in other domains. This requires the identification of a subset of participants, separating those participants from the rest, and finally transferring them to another application domain.

Different aspects. Framework documentation use different types of documents that differ on several aspects. While some documents are more prescriptive and provide task-oriented information, others are more descriptive and provide reference information, specifications of interfaces, interactions, and architectural constructs.

Depending on the concrete framework construct to document, it may be convenient to use and mix contents differing in many aspects [Butler, 1997]:

- **Notation.** Framework documentation contents are usually represented in different notations: free text, structured text, source code, object models, images, formal specifications, storyboards, tables, etc.

- **Linguistic.** The kind of content is considered by the linguistic aspect. Contents may provide information about purpose or intent (pragmatics), about meaning (semantics) or about structure (syntax).

- **Abstraction.** Documentation can describe framework constructs at different levels of abstraction, ranging from pure abstractions (abstract) to implementation of abstractions (concrete) or mappings from an implementation to an abstraction (mappings).

- **Granularity.** This aspect considers the scale of the information described. It can range from a global organization of responsibility (architecture), local organization of responsibility (micro-architecture), inter-objects communication (interaction), a single class (class), a single method (method), a single object (object), or a language primitive (primitive).

- **Variability.** This aspect considers the time-varying nature of contents: static or dynamic. Contents may represent time-invariant or time-variant properties of entities and their associations (static or dynamic).

To be effective and easy to use, the documentation environment must support contents varying in all these different aspects in an integrated manner. In addition, to provide a good navigation, the contents must be also...
extensively cross referenced, a challenging task considering the need to support different notations.

To satisfy the different needs of all kinds of (re)use and (re)users, several approaches are normally used to document frameworks, which include: framework overviews, example applications, cookbooks and recipes [Pree, 1995], motifs [Lajoie and Keller, 1995], interface contracts [Meyer, 1992], interaction contracts [Helm et al., 1990], reference manuals, pattern languages and design patterns [Johnson, 1992; Beck and Johnson, 1994; Lajoie and Keller, 1995].

Each of the major types of framework documents is later presented with the help of concrete examples taken from the documentation of the JUnit framework [Beck and Gamma, 1997], when existent.

### 3.4.3 Framework Overview

The first kind of information that a framework user looks for is contextual information about the framework. This introductory information is usually presented in the form of a framework overview that briefly describes the domain, the scope of the framework, and the flexibility offered.

Typically, a framework overview defines the common vocabulary of the problem domain of the framework and 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. It is typical to review a simple application, and to present an overview of all the documentation.

Figure 3.9 presents the framework overview of JUnit. It was extracted from the document “Frequently Asked Questions” (FAQ) [Clark, 2003], which is the JUnit’s document, from all the documents delivered with the framework, that most clearly presents the information typical of a framework overview, although its placement in a FAQ is not evident in a first look at the documentation.
3.4.4 **Cookbooks and recipes**

Cookbooks and recipes were historically the first technique used to document frameworks, namely the MVC framework [Krasner and Pope, 1988] and the MacApp framework [Apple Computer, 1986].

The Figure 3.10 presents a recipe for writing a simple test with JUnit, the first of the five recipes contained in the cookbook provided with JUnit, the "Cookbook" document [Beck and Gamma, 2003b].
Simple Test Case

How do you write testing code?

The simplest way is as an expression in a debugger. You can change debug expressions without recompiling, and you can wait to decide what to write until you have seen the running objects. You can also write test expressions as statements which print to the standard output stream. Both styles of tests are limited because they require human judgment to analyze their results. Also, they don’t compare nicely: you can only execute one debug expression at a time and a program with too many print statements causes the dreaded “Scroll blindness”.

JUnit tests do not require human judgment to interpret, and it is easy to run many of them at the same time. When you need to test something, here is what you do:

1. Create an instance of Test Case
2. Create a constructor which accepts a string as a parameter and passes it to the superclass.
3. Override the method runTest()
4. When you want to check a value, call assertEquals() and pass a boolean that is true if the test succeeds

For example, to test that the sum of two Moneys with the same currency contains a value which is the sum of the values of the two Money, write:

```java
public void testSimpleAdd() {
    Money allCHF= new Money(12, "CHF");
    Money allCHF2= new Money(14, "CHF");
    Money expected= new Money(26, "CHF");
    Money result= allCHF.add(allCHF2);
    assertEquals(expected.equals(result));
}
```

If you want to write a test similar to one you have already written, write a Fixture instead. When you want to run more than one test, create a Suite.

Figure 3.10 Example of a recipe for writing a simple test with JUnit.

Recipe

A recipe is a document that informally describes how to use the framework to solve a specific problem of application development [Pree, 1995]. Recipes present information in natural language, perhaps with some pictures and source code. Although recipes are rather informal documents, they are usually structured in a purpose section, a how to do section containing a sequence of steps to follow, and explicit, or implicit, references to other recipes, models and fragments of source code examples.

Cookbook

A cookbook is a collection of recipes acting as a guide to the contents of all recipes. Cookbooks 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 [Krasner and Pope, 1988].

A cookbook is specific to a framework. The users search the cookbook for the recipe that is most appropriate for their needs, and when found, they follow the steps described in the respective recipes.

The most important kind of information provided by cookbooks and recipes is prescriptive information, which instruct 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 amount strictly needed to help users understand minimally what they are doing.

Active cookbook

The active cookbook is a specialization of the cookbook concept that provides active guidance to framework users [Schappert et al., 1995]. Active cookbooks extend the cookbook idea with an hypertext representation, a visual development environment and supporting tools. The tool support provided by active cookbooks help 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 [Johnson, 1992] 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.

Motif

To avoid confusion with design patterns, the term motif was later introduced in [Lajoie and Keller, 1995] to name Johnson's patterns [Johnson, 1992]. 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

Hooks present knowledge about the usage of the framework, and provide an alternative view to design documentation [Froehlich et al., 1997]. 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: a hot spot tends to have several hooks within it. The usage of hooks can be semi-automated.

3.4.5 Patterns

The pattern concept was introduced in the software community by the influence of the Christopher Alexander’s work [Alexander et al., 1977; Alexander, 1979; Lea, 1994]. A pattern presents a generic solution to a recurring problem that might arise in a given context. The description of a pattern explains the problem and its context, the solution, and a discussion of the consequences of adopting the solution.

There are several documentation techniques around the pattern concept,
being design patterns, pattern languages and meta-patterns the most popular
eamples.

**Design Patterns**

A design pattern is a specialization of the pattern concept for the domain of
software design. Design patterns capture expert solutions to recurring design
problems. In [Buschmann et al., 1996] a design pattern is defined as the
following: “A pattern for software architecture describes a particular
recurring design problem that arises in specific design contexts, and presents
a well-proven generic scheme for its solution. The solution scheme is
specified by describing its constituent components, their responsibilities and
relationships, and the ways in which they collaborate.”

![Template Method Diagram](image_url)

**Figure 3.11** The Template Method (Gamma et al., 1995b).
Figure 3.11 presents an extract from the documentation of the Template Method pattern presented in [Gamma et al., 1995b], showing the structure of solution proposed by the pattern, the participants involved and their roles, and the consequences of instantiating the pattern.

As design patterns provide an abstraction above the level of classes and objects, they are suggested as a natural way of documenting frameworks [Johnson, 1992], for describing 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 enclose meta-knowledge about how to incorporate flexibility [Beck and Johnson, 1994; 1]. In fact, design patterns are able to illuminate and motivate architectures, to preserve design decisions made by original designers, to provide a common vocabulary that improve designers communication, and to help on the understanding of the dynamics of control flow.

3.6 Summary

We are at the end of our cook's tour through JUnit. The following figure shows the design of JUnit at a glance explained with patterns.

![JUnit Pattern Summary](http://junit.sourceforge.net/doc/junit-overview.png)

Figure 6. JUnit Pattern Summary

Notice how TestCase, the central abstraction in the framework, is involved in four patterns. Pictures of mature object designs show this same 'pattern density'. The star of the design has a rich set of relationships with the supporting players.

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

Figure 3.12 presents an extract from the document "A Cook’s Tour" [Beck
and Gamma, 2003a] 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.

Searching, selecting and applying design patterns are the necessary steps of the cognitive process for assigning the roles defined in an abstract pattern to concrete classes, responsibilities, methods and attributes of the concrete design. This process is generally called pattern instantiation [Odenthal and Quibeldy-Cirkel, 1997].

3.2 Blanks to fill in - run()

The next problem to solve is giving the developer a convenient "place" to put their future code and their test code. The declaration of Test Case as abstract says that the developer is expected to create Test Case by subclassing. However, if all we could do was provide a superclass with one variable and no behavior, we wouldn't be doing much to satisfy our test goal, making tests easier to write.

Fortunately, there is a concrete structure to fill in - they set up a test fixture, run some tests against the fixture, check some results, and then clean up the fixture. This means that each test will run with a fresh fixture and the results of one test can't influence the result of another. This supports the goal of maximizing the value of the tests.

Template Method addresses our problem quite nicely. Quoting from the intent, "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."

This is exactly right. We want the developer to be able to separately consider how to write the fixture (set up and tear down) code and how to write the testing code. The execution of this sequence, however, will remain the same for all tests, no matter how the fixture code is written or how the testing code is written.

Here is the template method:

```java
public void run() {
    setup();
    runTest();
    tearDown();
}
```

The default implementations of these methods do nothing:

```java
protected void setUp() {}
protected void tearDown() {}
```

Since setUp and tearDown are intended to be overridden but will be called by the framework we declare them as protected. The second snapshot of our tour is depicted in Figure 2.

![Figure 2 Test Case run() applies Template Method](image)

Figure 3.13 Template Method is instantiated by the Test Case class.

Figure 3.13 presents an extract from the document "A Cook's Tour" [Beck
and Gamma, 2003a] that informally documents how the class TestCase instantiates the Template Method design pattern, using natural language, models, and fragments of source code.

Documenting pattern instances is an important activity because it will help other developers to better understand 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 himself that he is taking the right decision. In general terms, 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, to justify the selection of the pattern, to explain how the pattern roles, operations and associations were mapped to the concrete design classes, and to state the benefits and liabilities of instantiating the pattern, eventually in comparison with other alternatives.

A framework typically encompasses several design patterns. The whole set of patterns for a specific application domain, together with their relationship principles, becomes a high level language, called a pattern language [Alexander et al., 1977]. A pattern language represents the essential design knowledge of a specific application domain, i.e. the accumulated experience gained by designers in solving a class of similar problems.

A pattern language can be used for documenting a framework in a natural language. Patterns provide a format for each recipe, and an organization for the cookbook [Beck and Johnson, 1994; Johnson, 1992; 1; Lajoie and Keller, 1995]. A pattern language acts as an explicit route-map through the architecture, which is considered “one of the most important skills possessed by experts” [Beck and Johnson, 1994].

Frameworks provide their flexibility at hot spots using two essential constructs: templates and hooks (Section 2.4.2). 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 [Pree, 1995; Pree, 1999].

Meta-patterns are a useful abstraction that can be applied to categorize and describe framework hot spots on a metalevel.

Although meta-patterns can be used to document the roles of framework participants, the level of detail is too fine to be useful. But they are extremely useful to document the roles of the participants involved in a design pattern.
For example, Figure 3.14 presents three participants of JUnit framework, the interface Test and the abstract classes TestCase and TestSuite, and attach one of the five design patterns they are involved, the Composite pattern [1], to the meta-pattern it follows, the 1:N Recursive Connection meta-pattern. Similar attachments can be made to the other patterns instantiated by these three participants. While this information would fit in a document intended to describe in very high detail the design of JUnit, it is preferred to attach meta-patterns to design patterns, and design-patterns to concrete participants that instantiate them, instead of attaching meta-patterns directly to concrete participants, mainly due to the redundancy introduced, and the level of detail being very fine.

![Diagram](image)

**Figure 3.14** Example of attaching a meta-pattern to a hot spot from JUnit.

### 3.4.6 Architectural models

Object-oriented architectures, and frameworks in particular, can be conveniently documented using object-oriented modelling languages, such as the Unified Modeling Language (UML) [Rumbaugh et al., 1999], and architecture description languages (ADLs).

**UML**

The Unified Modeling Language is a general-purpose visual modeling language to specify, visualize, construct, and document the artifacts of a
software system [Rumbaugh et al., 1999]. UML was standardized by the Object Management Group (OMG) [OMG, 2003] in November 1997, and a draft for version 2.0 is available in 2003. UML offers a rich set of concepts and models to describe the static structure and dynamic behavior of a software system using different views:

- **structural views**: logical, use case, implementation and deployment views can be modeled using class diagrams, use case diagrams, component diagrams and deployment diagrams, respectively;

- **dynamic views**: state machine, activity, and interaction views can be modeled using statechart diagrams, activity diagrams, and sequence and collaboration diagrams, respectively.

In addition to these diagrams, the UML provides elements for model management (packages) and for extending the notation (profiles, stereotypes and tagged values).

UML is a result of many years of maturation on object-oriented modeling techniques and is simple to learn (the most of it) and easy to integrate in the software development life-cycle, one of the reasons of its widespread use in the software community (but not only).

UML models can be used across all the life-cycle, from earlier phases of requirements analysis (with use-cases) to the final phases of deployment [Kruchten, 1999]. By using the same set of concepts and notation in different stages of development, translation of models is unnecessary, which is critical for iterative and incremental development.

UML models are well suited to describe both high-level constructs, at the conceptual level, and low-level constructs, at the implementation level, thus being very useful to document object-oriented frameworks: the application domain, the logical architecture (macro and micro-architectures), the implementation, and the physical architecture.

**UML-F**

The constructs provided by standard UML are however not enough to assist framework development. UML-F [Fontoura et al., 2001] is a new profile for UML that provides a set of extensions (graphical and textual) for explicitly modelling framework variation points in UML, which capture the semantics of the most common kinds of hot spots in object-oriented frameworks.

The usage of UML-F is simple, and consists on annotating classes and methods with tags and stereotypes that precise the role and flexibility provided at each hot spot.

The example previously presented in Figure 3.14 is used in Figure 3.15 to illustrate the UML-F profile for annotating part of JUnit classes and methods. It can be seen that for representing the Composite pattern instance
at Test, TestCase and TestSuite classes, were used a total of eight tags to annotate the roles and responsibilities: three class stereotypes, one association stereotype, and four tagged values. A drawback of UML-F is that complex representations would require a large number of tags, which all together may decrease the readability and understandability of the resulting UML model.

![UML-F representation for the Composite pattern instance in JUnit.](image)

**ADLs**

Architecture description languages (ADLs) have been proposed as an answer to the growing need to support the shift in software development from lines-of-code to coarser-grained architectural elements (software components and connectors) and their overall interconnection structure.

In [Medvidovic and Taylor, 2000] an ADL is defined as “a language that provides features for modeling a software system’s conceptual architecture rather than the system’s implementation. ADLs provide both a concrete syntax and a conceptual framework for characterizing architectures. The conceptual framework typically reflects characteristics of the domain for which the ADL is intended and/or the architectural style”.

ADLs have become an area of intense research in the software architecture community, and therefore a large number of different ADLs can be found: Rapide [Luckham et al., 1995], SADL [Moriconi et al., 1995], or Wrigth [Allen and Garlan, 1992] (see [SEI, 2003] for a longer list of ADLs).

Despite differing on the formalisms used, on the target domain, and on the supporting tools, an ADL has components, connectors, and architectural configurations as building blocks. In addition, an ADL provides means for their explicit specification, and eventually also provides tool support to render it more usable and useful.

Because ADLs are intended to specify a wide range of software
architectures, and not frameworks in special, ADLs offer more flexibility than needed to document object-oriented frameworks, resulting not very attractive for framework designers.

### 3.4.7 Contracts

A contract is a specification of obligations [Meyer, 1992]. Contracts provide a formal way of textually documenting a framework. They focus on describing with precision the syntax and semantics of individual class interfaces and the cooperative behavior of several interacting classes. The fundamental goal of using contracts and other formal approaches is to enable the specification of components, or complete applications, and the generation, or verification, of conforming implementations.

The interface contract of a class provides a specification of the class interface and class invariants in isolation. It specifies the type constraints given by the signature of a method and the interface semantics of the method [Meyer, 1992].

An interaction contract deals with the cooperative behavior of several participants that interact to achieve a joint goal [Helm et al., 1990]. The contract specifies a set of communicating participants and their obligations: preconditions on participants required to establish the contract, and the invariant to be maintained by these participants.

This kind of contracts are used by Mili and Sahraoui for representing object frameworks able to describe interobject behavior, how and when to use the framework, and how to extend the framework. An example of a generic specification can be found in [Mili and Sahraoui, 1999] and reproduced in Figure 3.16.

```plaintext
Framework <FM₁> extends <FM₂> {
    Variables: ...
    Participants:
        Part'₁ renames Part₁;
        Part₂: Interface'₂ specializes Interface₂; ...
    Constraints: ...
    Tasks:
        Task'₁ (I'₁: IN T'₁₁, ..., O'₁: OUT T'₀₁, ...) overrides
        Task₁ (I₁: IN T₁₁, ..., O₁: OUT T₀₁, ...) (...) 
        Task'₂ (...) (...) 
    Instantiations:
        Scenario'₁ (P'₁: T'₁₁, ..., P'₃: T'₃) overrides
        Scenario₁ (P₁: T₁₁, ..., P₁: T₁) (...) 
        Scenario'₂ (...) (...) 
}
```

**Figure 3.16** Example of a framework specification using Helm contracts.
3.4.8 Reference Manual

A reference manual for an object-oriented system is the document that contains the descriptions of all classes. A description of a class typically presents the purpose or responsibility of the class, the role of each field, and some information about each method. In a description of a method it is important to describe the functionality provided, the kind of operation (query, accessor, modifier, utility), pre-conditions and post-conditions, and eventual side-effects of executing the method.

For framework documentation, the descriptions can include additional material concerning the role of a class or method in providing flexibility for a hot spot, whether a class is intended to be sub-classed or a method to be overridden, and eventual pattern instances associated with the class or method.

Reference manuals are usually not intended to be easy to read, but instead to be easier to consult, and therefore they are not appropriate for learning a framework by new users. Reference manuals are most useful for advanced users when looking for descriptive information about the artifacts and constructs of the framework.

Because reference manuals are highly structured and must contain the complete information about the framework, they are usually automatically produced with the help of general-purpose or customized documentation generation tools, such as Javadoc [Sun Microsystems, 2003] for Java source code, Doxygen [van Heesch, 2002] for C++ source code, or CASE tools [Borland, 2003; Quatrani, 1998].

3.4.9 Example Applications

The source code of example applications constructed using the framework is often the first documentation provided to application developers. Documentation requires a graded set of training examples, each illustrating a single new way of customization, and providing eventually complete coverage. The usage of hypertext links in the source code and the availability of executable code are a valuable help for the understanding of the examples.

The cost of producing the examples to deliver with the framework is usually very reduced, if well planned, as the examples can be used to drive the development, to verify the real reusability of the framework, and to help document the framework.

Figure 3.17 presents an extract from the article “Test Infected: Programmers Love Writing Tests” [Beck and Gamma, 2003c] 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.

```java
class Money {
    private int amount;
    private String currency;

    public Money(int amount, String currency) {
        this.amount = amount;
        this.currency = currency;
    }

    public int amount() { return amount; }
    public String currency() { return currency; }
}
```

When you add two Money of the same currency, the resulting Money has as its amount the sum of the other two amounts.

Figure 3.17 MoneyTest: an example provided with JUnit.

3.4.10 Software Documentation Techniques

The techniques useful to document frameworks include not only techniques to document in-the-large (architectures, subsystems and interactions), but also techniques to document in-the-small, i.e. to document local understanding, and these techniques have not really changed very much in thirty years.

So, in addition to the techniques specifically used to document frameworks, these other techniques, not framework-specific, must be also considered when addressing the issue of framework documentation. Framework documentation techniques and these other techniques, hereinafter referred as general-purpose software documentation techniques, address the same fundamental problem, i.e. the software comprehension problem, although with different levels of concern in terms of generality and applicability.

Software is inherently complex [Booch, 1994]. Without adequate documentation, software is difficult to understand. It was observed that
maintenance programmers need to spend up to half of their time trying to understand existing software, in most cases, directly in program code [Parikh and Zvegintsov, 1983].

Developing software is a highly demanding activity. Developers need to constantly map their own mental representation of a solution to other representations usable by computers. Specially with modern software architectures, such mappings often require the usage of multiple languages and different levels of abstraction at the same time. In addition to the programming activities, developers must also preserve and communicate their understanding in the form of documents to potential users, which usually play different roles and have different needs.

Developers produce the documents they need often using techniques and tools that don’t fully fit their needs and therefore end up introducing new difficulties in the overall development process. Because documenting is hard, and developers don’t see short term benefits on it, they simply focus on programming, unless mandated, letting documentation almost neglected.

The many artifacts produced during software development include concrete source code files, abstract models, and informal documents, all of which require continual review and modification throughout the life cycle in order to stay consistent and preserve their value.

Internal documentation. The understanding of a software system can be documented internally in the source code or externally in documents. Typical internal documentation captures the understanding of a program in order to preserve it over time, but is often limited to low-level, textual explanations usually included in source code comments, which are not convenient to document global system understanding, i.e. one that crosses several sections of a software system.

External documentation. On the other hand, higher-level external documentation is capable of capturing the components and connectors of an architecture, and the interactions of cooperating classes, but the consistency between external documents and source code can be difficult to maintain as the system evolves over time.

Literate programming was invented by Donald Knuth [Knuth, 1984] and is a well known possible solution to the problem of preserving the consistency of source code and documentation. The main idea of literate programming is that when writing programs, we should not try to instruct the computer what to do, but rather we should try to tell humans what we want the computer to do.

The technique involves writing documentation and code in a single source
document (verisimilitude), psychologically organized for comprehension by humans rather than computers. The technique has the following distinguishing characteristics:

- **Verisimilitude.** Code and documentation are written together in a same source document. This ensures that documentation can evolve independently of the code but always in consistency with it.

- **Psychological arrangement.** A literate program can be organized in the way considered most appropriate for human comprehension, without being restricted to follow the structure expected by the compiler. In fact, the primary ordering of a literate program depends on the documentation, with code interleaved as necessary.

- **Good readability.** Knuth suggests considering programs as works of literature, easy to read and understand, including cross-references and indexes, and beautifully typeset as a book, such as the full source of TeX [Knuth, 1986].

A literate program is a unified source document containing program fragments and explanatory text, organized into small sections or chunks, laid out in the order that the author seems to be the best for describing the program. This combined form is neither source code nor documentation and is traditionally called a web document, derived from the notion of web of information (the name is not derived from the World Wide Web because it predates it at least 15 years).

Literate programs can include both internal and external documentation. They can include not only the description of the function and purpose of the code, but also high-level information, such as problem statement, design decisions, how to use, background details, and any other information able to improve the understanding of the program.

Literate programming systems, such as the web [Knuth, 1983], the cweb [Knuth and Levy, 1994], or the noweb [Ramsey, 1994], provide two tools: one called tangle that allows the automatic extraction of computer-understandable code, and another called weave that produces human readable, comprehensible, and beautifully typeset documents. Neither of these extracted forms is intended to be modified. The tools and processes of weaving and tangling are shown in Figure 3.18.

The technique provides significant incentives for developers to document while they code, potentially leading to programs of higher quality and maintainability.
Despite its advantages, the technique is not widely used beyond academics or expert programmers working on personal projects. This is mainly due to the integration difficulties of literate programming tools in mainstream development environments, i.e. the lack of real world usage by ordinary programmers and teams who have not themselves designed the tools. These difficulties result from the following reasons:

- literate programming requires the combined use of three languages: the programming language, the formatting language, and the interconnection language, i.e. the language that enables the definition, referencing and inclusion of chunks;

- literate programming introduces too much overhead for small programs;

- the format of literate files is complex, what seriously compromises their on-screen readability and understanding during development;

- most literate programming tools support the documentation of new software, but are not well suited to document already existent software;

- and finally, the organization of the source code seen by the compiler is different from the original, as written by the programmer, what often cause problems when using tools that manipulate source code files, such as debuggers, generators, or refactoring tools.

**Single source**

Alternative documentation techniques to literate programming can be classified into either single source or multiple source methods.

Single source methods integrate code and documentation together in the same file, so there are no consistency problems between code and documentation as there is no replication of contents in the whole documentation bundle.
The Sun's Javadoc utility for interface documentation [Friendly, 1995] is an example of single source method that *weaves* Java source files to produce interface documentation in HTML, using simple Java commenting conventions. However, Javadoc cannot be considered a literate programming tool because it lacks the support for psychological arrangement of documentation. Javadoc documentation is targeted only for users but not for maintainers of the program code. It follows the structure of the code, and is limited to the abstractions supported by the programming language, typically classes and methods.

Other examples of similar tools are XDoclet [XDoclet, 2003], Doxygen [van Heesch, 2002] and Doc++ [Wunderling and Zockler, 2002]. Although some of them provide powerful extraction and typeset features, they also lack the support for psychological arrangement.

**Multiple source**

Multiple source methods maintain documentation and code in separate files. Traditional documentation written externally to source code is the most common example of such technique. Although these methods support psychological arrangement, they lack (real) verisimilitude. Due to the separation of code and documents, authors usually copy source code into document editors, or refer to a specific range of lines of a program file, but as soon as the code changes, the documents become inconsistent with the code.

More sophisticated systems simulate the verisimilitude characteristic of literate programming paradigm, with the help of hypertext systems and tools [Conklin, 1987; Ashman and Simpson, 1999] that automatically manage the strict relationships between code and documents. The most typical problem with this kind of systems is requiring the edition of code and documents inside their tools, a serious obstacle for easy integration in mainstream development environments.

**DOgMA.** An example is the DOgMA tool [Sametinger and Pomberger, 1992]. DOgMA is an enhanced hypertext system that aims to support literate programming in C++. Unlike typical literate programming tools, the source code is not really mixed with the documentation text, but stored in separate files and linked together using identifiers. DOgMA is in fact a multiple source tool.

The documentation consists of a set of documentation chapters, spreading several files organized in a hierarchy, where each chapter contains documentation text (real) and program text (actually a link to source code). The tool merges the source code into the documentation for browsing purposes. The tool provides an integrated environment supporting both the edition and browsing of documentation and source code using hypertext facilities that ease the navigation through the classes and documents of the
software system. DOgMA supports documenting both new and already existing systems.

Elucidative Programming. A more recent example is the Elucidative Programming tool, called the elucidator, a development of Nørmark [Nørmark, 2000a] that enables the linking between source code and documents by inserting special directives. A special characteristic of elucidators is the use of two synchronized windows laid out side-by-side to simultaneously present two parts of the documentation.

Initial Scheme and Java elucidators [Nørmark, 2000b; Nørmark et al., 2000] use separate applications for editing (Emacs) and browsing documentation (web browser). The last prototype tool [Vestdam, 2003] integrates an elucidator in the Borland’s Together IDE [Borland, 2003] thus enabling both activities inside the IDE.

Elucidators support the definition of relations between code and documents based on knowledge extracted from the documents and source code, which is stored in data structures, in the Scheme elucidator [Nørmark, 2000b], or in a relational database, in the Java elucidator [Nørmark et al., 2000].

Similarly to literate programming, elucidative programming is primarily intended for internal documentation [Nørmark, 2000c], but also supports external documents structured using a very simple, special purpose markup language.

Although elucidators do not provide the same level of incentives to document program understanding as literate programming, because code and documents are edited and browsed in separate windows, they have the advantage of not requiring the specific tangling process, thus enabling direct edition and compilation of code, which improves their real world usage and integrability in a larger universe of developers.

3.4.11 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, specially to new users.

Task-oriented, accurate, complete, understandable, and easy to navigate documentation are common attributes of good quality documentation, often hard to achieve, and eventually impossible for poorly designed frameworks, as good design is a necessary but not sufficient prerequisite for good
documentation.

There are several issues associated with framework documentation, both technical and non-technical. They include fundamental issues of technical documentation, such as quality assessment, common issues of software documentation, and several issues specific to framework documentation.

This section discusses a set of key issues, organized in: technical documentation issues, software documentation issues, and specific issues of framework documentation related with the documentation product itself, the process used to create it, and the tools used to support the process. The set of issues presented do not pretend to be mutually independent, and in fact some of them have overlapping sub-issues and requirements. In addition, the set of issues presented does not pretend either to be exhaustive, but only to enumerate the most important issues experienced by the author, and the most relevant for this dissertation, selected from a list of several issues mentioned in the vast literature on the topic, namely [Johnson, 1992; Taligent Inc., 1993; Taligent Press, 1994; Taligent Press, 1995; Fayad and Schmidt, 1997; Butler, 1997; Butler et al., 2000; Fayad et al., 1999; Clements et al., 2002].

Perhaps the greatest challenge of framework documentation is to obtain the appropriate knowledge from framework developers [Butler et al., 2000].

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.

Quality was defined by Wright as “an elusive concept with many meanings” [Wright, 1994]. Quality documentation has a multidimensional nature, and through the years, a variety of dimensions have been discussed, including aspects such as: usability, accessibility, readability, and consistency.

The book Producing Quality Technical Information (PQTI) [IBM Corporation, 1983] is considered by many to contain one of the earliest comprehensive discussions about the multidimensional nature of quality documentation [Waite, 2002; Smart, 2002; Grice, 2002]. PQTI suggests seven measurable dimensions of documentation quality, which all together provide a way of measuring documentation quality, although in a simplistic way. The dimensions suggested are: task-orientation, organization, entry points, clarity, visual communication, accuracy, and completeness. For each dimension, the book identifies a set of requirements (a total of 29). To help writers, or reviewers, to judge more objectively the quality of the documentation, each requirement is discussed with the help of a good example and a bad example.
In the revised edition of PQTI, *Developing Quality Technical Information* (DQTI) [Hargis, 1997], 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 3.19). DQTI is targeted for general use and takes into account online information (help, tutorials, etc.).

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

Figure 3.19 Quality dimensions for technical information [Hargis, 1997].

Despite the valuable contributions, PQTI (and DQTI) doesn’t consider the contextual nature of the quality aspects. The relative importance of individual dimensions is not static, it changes, and usually differ from document to document, depending upon the specific audience, context, and purpose of the document [Smart, 2002]. As a result, the quality of a technical document can’t be entirely assessed independently of the context in which it will be used, what suggests that producing effective documentation needs to be grounded in user data and user-centered design processes. In order to differentiate the relative importance of quality dimensions, Karl Smart suggests to classify quality attributes into three categories: essential, conventional, and attractive.

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, thus they won’t be discussed in more detail.
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. 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.

**Semantic consistency.** The many artifacts produced during development require continual review and modification throughout the lifecycle. To maintain the semantic consistency of all the different software documents is often difficult. One of the reasons is that formal documents and tools usually don't interoperate well with informal documents and their tools. Another reason is that the relationships between software documents are often implicit in the contents and thus their relationships cannot be browsed, navigated, queried, or analyzed systematically [Gupta et al., 2002].

**Appropriate documentation environments.** Typically, different kinds of contents are maintained independently using different environments and editors (text editors, source code editors, visual model editors, document editors). Such configurations often require constant switching between working environments, thus resulting inappropriate for maintaining the semantic consistency between several documents, as they disturb developers and cause significant process inefficiencies. There is thus a need for integrated documentation environments, powerful, widely applicable, and highly usable tools able to interoperate with the most common development environments. The existence of tools interoperable with typical development environments supporting all the software development lifecycle would constitute an excellent incentive for documenting software with quality.

**Knowledge acquisition.** The most complete mental model of the problem understanding and design of the solution is formed by the developer at implementation time. However, most of this understanding is often lost in the code, or in the developer's head because it was not adequately documented at the right moment, thus requiring a large effort to recover it later. The development environment must provide incentives for developers to document what they need, when they need, be it before, while, or after
implementation.

**Programmer's attitude.** Inadequate documentation is also a consequence of an attitude problem. Parnas suggests [Parnas and Clements, 1986] that “most programmers regard documentation as a necessary evil, writing it as an afterthought only because some bureaucrat requires it. They do not expect it to be useful.”. Often, documenting is postponed indefinitely, an attitude that must be changed. Good software documentation should be seen as a challenge, genuinely useful, and appreciated in its own right, as happens in other engineering disciplines. This requires to tell students early in software engineering courses what are the benefits of documenting software, and, most importantly, it requires also to provide developers with appropriate tools and guidelines that help reduce the difficulty of documenting, otherwise they don't get convinced.

**Product Issues**

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

Similarly to other technical documentation, the overall product quality of framework documentation is complex to determine and assess, and this is perhaps the first issue. Documentation must have quality, that is, it must be easy to find, easy to understand, and easy to use.

Using the quality dimensions proposed in DQTI [Hargis, 1997] (Figure 3.19), and the categories defined by Smart [Smart, 2002], we can define a possible arrangement of quality attributes that reflects their relative importance for framework documentation:

- **essential quality attributes** (attributes necessary to achieve minimal levels of reader satisfaction [Smart, 2002]): task-orientation, accuracy, organization;
- **conventional quality attributes** (attributes that cause satisfaction when present and dissatisfaction when not present, also known as *the-more-the-better* attributes [Smart, 2002]): retrievability, visual effectiveness, clarity;
- **attractive quality attributes** (attributes that go beyond readers expectations [Smart, 2002]): style, concreteness, completeness.

The difficulties more specific to framework documentation are related with essential and conventional quality attributes, as considered by the author, being concretely the following: 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, although easy to produce and maintain.

Combining prescriptive and descriptive information. Framework users want practical information, explaining how-to-use the framework. To enable users quickly start using the framework the documentation must combine prescriptive information, and concrete examples, easy to learn and experiment. However, related descriptive information (patterns, contracts, etc.) is also important to be available, to explain how the framework works, or even to show the concrete details of the implementations, for those readers looking for a better understanding of the framework internals. Documentation must be written from the reader's point of view, ordered in a way that reflects the order of use, and to explain the importance of the information presented. The titles and headings must reveal the task that readers are being told how to do.

Managing consistency and redundancy. Readers comprehension is fast when information is provided with accuracy, without inconsistencies between their different kinds of contents: textual descriptions, source code, models, cross-references, auxiliary information, writing (grammar, spelling,...), and consistency of terms. Sometimes it is useful to include redundant information (to repeat an item of information) instead of using a reference, to avoid forcing readers to branch unnecessarily. This redundancy adds extra difficulty to the typical semantic consistency issue of software documentation (page 95).

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. This is particularly important for the large volume and diversity of contents of framework documentation. In addition, as framework documentation can be classified using different and independent criteria (Section 3.4.2), it would be useful to support an orthogonal organization of the contents, so that they can be extracted, filtered and merged in multiple ways by the readers with reduced effort.
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 [Shull et al., 2000]. 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, such as architectural documents (abstract contents), or source code examples (concrete contents). Another important requirement is the ability to follow the logical thread of thought in the documentation through the pertinent parts using search facilities and references.

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. Through folding (i.e. expand and collapse), filtering and transformation of contents, documentation views must be able to convey the most valuable information for a specific objective.

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. In order to support new kinds of documents as inputs, the integration mechanism must be flexible. Literate programming suggests merging documents and code in a same file, but the most common practice in software development is to keep documents separated from source code [Kramer, 1999; Sun Microsystems, 2003].

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.

The quality of the documentation has a direct link with the maturity of the process used to create it. Mature documentation processes would lead to less documentation defects and therefore to less software development defects [Huang and Tilley, 2003; Visconti and Cookand, 2002].
The ultimate reason for producing documentation is to help framework users on learning how to instantiate the framework, and how the framework works.

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.

**Lack of standards.** Defining suitable methods for framework documentation is an important first step to reduce the high cost of providing suitable framework documentation. 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. As mentioned before, the seminal report that IBM produced in 1983 (PQTI) [IBM Corporation, 1983] is a good example of using guidelines and checklists to help assess and improve documentation quality; PQTI “is one of the earliest and perhaps best attempt at laying out the characteristics that constitutes quality documents” [Smart, 2002]. As a result, it is important to get from documentation experts such checklists and guidelines, so that writers, specially novices, can find guidance when documenting a framework.

**Cost.** To be useful in practice, the cost of documentation must not significantly outweigh its benefits. Specially 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, Sametinger suggests [Sametinger, 1994] that documentation should be reusable in the same way as object-oriented software systems do, thus allowing to extend and modify 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).

**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 a 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, specially 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.

This long enumeration of issues implicitly suggests that the topic of framework documentation is a rich research topic, despite the associated difficulties related with its broadness in terms of the knowledge areas involved, and its fuzzy boundaries as a research topic.

### 3.5 Maximizing Benefits of Frameworks

The benefits from frameworks are very significant, but they result from developing and using successful frameworks, which usually is a challenging task for many organizations.

To maximize the benefits of frameworks it is useful to understand what are the key ingredients for developing, supporting, and maintaining successful frameworks.

#### 3.5.1 Aspects to Consider When Building Frameworks

When designing and implementing frameworks it is important to consider the aspects that can help maximize the benefits of frameworks. They are the following:

- **Documentation:** to explain well the framework, from all relevant points of view (purpose, how-to-use, how it works, and how to extend).
- **Standards:** to follow common coding standards.
- **Change management:** to devise strategies for managing framework change.
The theory of minimalist instruction [Carroll, 1990] is a theory with foundations in the psychology of learning and problem solving. The minimalist instruction intends to help on the design of instruction material so that people can learn faster and for longer. If learning to do something new is always a difficult undertaking, to help someone else to learn is even more challenging.

Minimalist instruction is an alternative to the standard systems approach. The major theme of standard systems approach is the hierarchical decomposition of learning objectives, building skills from the bottom up, by step-by-step drill and practice.

One goal of the minimalist approach is to teach people what they need to know in order to do what they wish to do. Among other characteristics, a minimalist manual motivates people to train on real tasks and getting started fast, presents topics very briefly in the order that seems best for the reader, supports error recognition and recovery, and tries to explore readers' prior knowledge.

This chapter reviews the fundamentals of minimalist documentation based on [Carroll, 1990; Carroll, 1998], the best references on minimalism.
4.1 Minimalist Documentation

In this approach to documentation, the instruction material is designed in a way that suits the learning strategies that people spontaneously adopt, where the reader is offered with the minimal amount of information needed to get the task done. The key idea in the minimalist instruction approach is to minimize the obtrusiveness to the learner of training material, hence the term. There are three main aspects that characterize the approach: more to do, less to read, and help with errors.

4.1.1 More to Do

The most important factor in learning is the learner motivation, but this is also the least amenable factor to control through design. If learners want to undertake a particular activity, to let them try to do it is perhaps the best design step to take. Therefore, minimalist instruction aims to allow learners to start immediately on meaningfully realistic tasks.

4.1.2 Less to Read

New users are not inclined to read training material [Rettig, 1991]. People seem to be more interested in action, in working on real tasks, than in reading. Therefore, minimalist instruction aims to reduce the amount of reading and other passive activity in training.

4.1.3 Help with Errors

In training situations, to make errors is best regarded as inevitable and perhaps intrinsic to human learning. Training materials must therefore explicitly support the recognition of errors, and the recovery from errors both to make the materials robust with respect to user error and to train error recovery skill. Particularly in self-initiated forays of exploration, errors may play a unique constructive role in facilitating the discovery of new knowledge. Therefore, minimalist instruction aims to help making errors and error recovery less traumatic and more pedagogically productive.

4.2 Learning systems

Frameworks are often large and complex. New users can rapidly go from being excited and expectant reusing a framework to being lost and frustrated.
4.2.1 Being Overwhelmed

It is wonderful to be excited and expectant at the prospect of learning a new skill, but often people allow themselves to be too optimistic about learning new things. They expect to be easy to learn how to reuse a framework and with little effort.

An initial optimism by the new user can fade into self-blame or cynicism after a few mistakes and getting lost following the instruction materials. The learning task begins with the problem of identifying appropriate goals, means for attaining them, and drawing rudimentary connections. One consequence of these difficulties may be that learners lose track of what they are trying to do or what they have learned.

4.2.2 Following Directions

In general, people do not follow directions willingly or well. People typically create and respond to their own agenda of goals and concerns and not to the careful ordering of steps in a training procedure. Learners don't seem to appreciate overviews, reviews, and previews; they want to do their work.

The failure to coordinate the system and training takes the form of opportunistic reading: browsing ahead for a topic that seems interesting or relevant to a current concern, without much regard to its prerequisites.

4.2.3 Interacting with the system

People want to work on meaningful tasks. No system can be so simple as to make reasoning unnecessary. Additionally, people tend to follow directions unpredictably. People often refer to their prior knowledge and experience under an assumption of consistency, what in fact may create false expectations and result in a lot of inconsistencies.

4.2.4 Recovering from Errors

New users make many errors. Training materials usually do not include specific error recovery procedures. The unpredictable reading strategies and the learner's own goal-directed activities deter the learner from following step-by-step instructions correctly, and by consequence a lot of errors may occur in the process, due to synchronization problems between training material state and system state.

Many errors could tangle with one another, and recovering from a tangled error can be so complicated that learners can lose their place in what they
are trying to do. Learners often are unable to articulate their problems to make effective use of the interactive help; they often lack the background knowledge required to make sense of the help.

Help is easiest to give for specific and isolated questions of information, but learners don’t typically confront this kind of question.

Designers of help systems have to decide whether they are providing tutorial information for the learner or reference information for the skilled user; they can’t effectively do both.

4.3 Minimalist Instruction

Carroll takes up the possibility that user problems occur when people adopt reasonable approaches to learning in situations that fail to support those approaches.

4.3.1 The Weaknesses of Systems Approach

The first step in designing such systems instruction material is a fine-grained decomposition of target skills. This decomposition can be used to derive a curriculum sequence; simpler component skills are practiced and mastered before the learner attempts the more complex skills they comprise. The systems approach’s instruction is designed with little consideration of the learners and no consideration for the contexts within which learning will occur.

All the learner needs to do is to follow the steps, but this may be both too much and too little to ask for the people. The problem is not that people cannot follow simple steps; it is that they do not. People are always trying things out, thinking things through, trying to relate what they already know to what is going on, and recovering from errors. They are too busy learning to make much use of the instruction. This is the paradox of sense-making.

Working toward the achievement of a personal goal increases the chance that the activity will be personally rewarding, and hence that the person will remember it and want to do more. The motivation to interact meaningfully in a situation is also the root of a learning paradox: to be able to interact meaningfully, one must acquire relevant skills and understanding, what can only be done through meaningful interaction. When a learner reads through a manual, skipping things and reading in any order, this can be seen as desperate effort to inject meaning into a training experience. Learners surely need to acquire the skills and knowledge that standard training structures into a step-by-step curriculum for them. The key obstacle is the paradox of
sense-making: to learn, they must interact meaningfully with the system, but to interact with the system, they must first learn.

4.3.2 Design Principles

The key idea in the minimalist approach is to present the smallest possible obstacle to learners' efforts, to accommodate, even to exploit, the learning strategies that cause problems for learners using systematic instructional materials. This approach does not solve the sense-making paradox; rather it compromises in the direction of accommodating the learner's desire for meaningful interaction at the expense of providing a less comprehensive curriculum.

As a design theory, the minimalist instruction doesn't prescribe ways of producing minimalist manuals, but instead it defines a set of design principles, which are presented below.

Learners want to get something done, and an instructional manual that fails to support this motivational orientation fails as instruction.

People personalize their own knowledge and skill through their activities. The identification with an activity is a powerful source of motivation that can enhance learning. This can be achieved when the learning task is the learner's task, when the learning situation is under the learner's control and when that learner personalizes the activities of learning.

Human problem solving is neither confined to, nor always well served by hierarchical control. The structuring discipline of the systems approach specifically misfit human learners' organizational propensities.

Activities undertaken directly in the service of a meaningful goal are remembered concretely, accurately, more effectively and more durably when contrasted to those undertaken merely as means to other ends. The understanding of the overall goal of an activity can be a condition for even being able to carry out its component steps reliably.

The minimalist approach urges allowing the user to select meaningful tasks to work on, to personalize instructional projects. It is committed to allowing the user to pursue a sequence of goals, each a meaningful end in itself, not a mere step meaningful only as part of something else.

It is quite common for training manuals to present a welcome to the system preface. This, even in the end, does much to facilitate the user's desire to get started on meaningful activity. Rather, it obstructs this goal.

The natural energies that sustain spontaneous learning are curiosity, a desire for competence, aspiration to emulate a model, and a deep-sensed
commitment to the web of social reciprocity.

Learning by reading often conflicts with the more basic human need for getting something done.

Studies of skill learning indicate that people learn by induction from concrete examples, not from being told how to do things. The minimalist approach urges getting learners started doing projects as quickly as possible, allowing meaningful and concrete activity to provide intrinsic learning guidance, rather than relying on extrinsic guidance of conceptual elaboration and practice with numbered steps, techniques that too often become obstacles to learning.

Learners are always in the state of just getting started, and needing to get started fast on projects they could recognize as realistic. This suggests that an appropriate way to view all skill development is as iterative getting started.

A key role of instruction is to guide the learner to pose productive questions and to adopt appropriate methods to investigate them.

Allowing people to reason about what they are doing, allowing learners in part to create, and not merely to consume, instructional material, can enhance cognitive processes in a variety of ways.

Incompletely specified, open-ended learning tasks are also often more motivating. Studies suggest that tasks are better remembered and more likely to be returned to if they are incomplete on first encounter.

Permitting self-directed reasoning and improvising to play a more central role in instruction allows, indeed it requires, substantial reductions in the volume and verbiage of training materials.

People who are reasoning out the meaning of training tasks do not need to read as much about what they are doing.

The rule of thumb in minimalist design is to try to cut and condense text and other passive components; the goal is to enrich the training experience.

The learner needs more to think and less to overcome.

When people do try to learn by reading, they are frequently unsuccessful, apparently due to material design errors, like sequencing problems. However, when people refer to instruction opportunistically in support of their own goal-directed activities, it becomes difficult or impossible to predict what sequencing will be appropriate even when the logical dependencies are merely linear.

To provide material that can be sensibly read in any order, necessitates a different approach to organizational instruction, requiring a high degree of
modularity, a structure of small, self-contained units. The orderly accumulation of prerequisite skill and understanding that can be assumed when material is embedded in a sequenced curriculum cannot be assumed if learners use the material in any order they wish. But this is just what learners do anyway. One cannot eliminate the need for prerequisite knowledge in learning, but one can try to minimize the tangles and side effects that occur when prerequisite relationships are disregarded.

The linkage between the system and the training needs to be flexible and robust. It is required to incorporate rich implicit and explicit linkages into the training materials. The learner's task goals and task-oriented reasoning strategies provide an implicit structure for coordinating attention to the system and training.

The pervasiveness of error in learning makes it unrealistic to imagine that learner error can be eliminated. Moreover, it was concluded that error can play an important and productive role in learning and intellectual development.

Errors can be of diverse categories:

- the learner has an appropriate goal but takes an inappropriate action;
- the goal and action are appropriate, but a preexisting condition precludes the usual and intended effect;
- the goal and action are appropriate but still be wrong because the instruction material currently in effect specified other activities.

A more appropriate design goal is to support the recognition of and recovery from errors in order to help ensure that when errors are committed they are productive for the learner.

One of the main activities in the design of minimalist instruction is compiling an inventory of user errors, particularly those that are typical or have serious consequences.

The prior knowledge suggests goals and interpretations to systems and instruction materials.

The importance of prior knowledge is key to the failure of systematic instruction. Not having specific goals and expectations might make learners more amenable to drill and practice techniques, more willing to read, less likely to reason and improvise on their own.

One of the most important aspects of training design is understanding the users' prior knowledge and motivation and then finding ways to exploit it.
There is a need to experience and exploit the fine detail of actual situations in order to create practical solutions. The fine details of real situations are an individual's preferences and choices, the contents of a given memo, the possibilities for learning enrichment, and personal autonomy on a given weekday morning.

The above principles are one of several sets that have been used to define minimalist instruction. Like any other design philosophy there is no deductive theory of minimalist instruction.

Given a set of minimalist principles, it is not possible to crank out a training manual. Design never works in this way.

It is not possible to mechanically generate effective training. The development of instructional materials is largely a matter of direct empirical analysis and iterative adjustment of particular trade-offs.

The starting point for such design is an eclectic synthesis of available design elements (instructional metaphors, techniques for coordinating learner attention, tasks that learners want to accomplish) marshaled to address instructional objectives and usability objectives.

At the beginning, it's always better to include more ideas and approaches, and as the design attain the integrity of a whole, it is easier to cut than to add.

The analysis of instructional objectives is decompositional, as in systematic training design, but at a far coarser grain of analysis, and requires that all target objectives be meaningful to learners a priori. The typical questions are:

- What are the tasks users will want to learn and need to learn?
- What are the skills they will need in order to perform these tasks?

The analysis of usability objectives is heuristically guided by the minimalist principles but is always largely empirical.

One common mistake at this stage is to rely inordinately on expert judgements.

The final stage of design is traditionally called summative testing. This final test is, of course, also the starting point for further design and evaluation.

### 4.3.3 The Limits of Learning

Human learning is paradoxical, in the sense that no simple, comprehensive, logical treatment of the sense-making paradox is possible. One cannot design training that is both usable and comprehensive.

Minimalist instruction pushes toward a compromise in the trade-off,
promoting personally meaningful activities but sacrificing some of the comprehensiveness of systems-style curricula.

Experienced users are by definition chiefly working on their own and thereby escape learning problems aggravated by systems-style instructional materials. Reference and help material for more experienced users would also benefit from the minimalist model.

In order to reason successfully about a situation, the situation must be interpretable in some preexisting schema. Only when one has been able to make sense of the situation in this manner can one reason fluently about it. The paradox of sense-making may be a fundamental constraint on human learning and reasoning.

The minimalist training model was outlined as a vehicle for addressing the problems of new learners in ways that accommodate their need for sense-making. The minimalist instruction aims at trying to exploit the sense-making capabilities and propensities people bring to a learning situation, in order to produce more efficient learning.

4.4 The Minimal Manual

The elaboration of a minimal manual often requires an iterative development process, which consists of three stages of activity:

- design analysis – analysis of the instructional situation in view of available instructional models, being the outcome a first-draft design;
- subskill evaluation – detailed empirical assessment of typical users attempting typical tasks, which is directed to redesign;
- and criterion testing – summative evaluation of the entire design.

4.4.1 Design Analysis

The design should be based on an inventory of user problems, the minimalist training model, and past experience with guided exploration. Below are presented the key aspects of concern when designing minimalist manuals, organized by design principle:

- **Training on Real Tasks.** The topic chapters must be clearly labeled with topics of interest to learners. The chapters broken into brief subsections providing meaningful components of basic tasks. Material not task oriented must be eliminated in many cases.
• **Getting Started Fast.** In order to the learner start the hands-on portion of the instruction very fast, it is needed to be ruthless about cutting verbiage.

• **Reasoning and Improvising.** Many opportunities must be built for learners to think about things and to do things on their own. Instead of rote exercises, must be suggested open-ended “On Your Own” projects, which might engage the learners by presenting more personally meaningful challenges. It can be included some explanatory material, observed important for guided exploration, but very briefly.

• **Reading in Any Order.** Important procedures must be streamlined so that learners can initially skip complicated options. The topic chapters must be designed to be highly modular. The objective is to limit their interactions as much to the level of topic goals and the coarse-grained subgoals. The fact that the various topic goals and subgoals are introduced succinctly, with much of the situational detail left to the learner to elaborate, means that the topic chapters and subsections can more easily be adapted for such cross-referenced use. This helps to minimize the risk of skipping around.

• **Coordinating System and Training.** Coordination of the system and training can be encouraged using different techniques: incomplete specification of procedures when required information can be found by the learner in the system; or incorporation of explicit interrogative or imperative checkpointing into the manual.

• **Supporting Error Recognition and Recovery:** The principal errors of new users must be inventoried and included with specific error recovery information to address those problems.

• **Exploiting Prior Knowledge.** When introducing system concepts it is important to exploit users’ prior knowledge (domain or technical), by referring to them when there is some similarity or calling the attention to some behavior of the system anticipated as not natural for the learner, due to his prior knowledge and experience.

• **Using the Situation.** Learning situations can be anticipated and plan specific material that uses them.

### 4.4.2 Subskill Evaluation

After the design analysis, design elements can be tested in qualitative detail. Sometimes, the recognition of errors by the learner is not easy because the system may provide an obscure feedback to some errors. Therefore, the inclusion of specific information to help recognize situations of error and
further information for recovery is important.

4.4.3 Using the Minimal Manual

Minimal manuals usually make the learning faster for the basic topics and also for the more advanced topics. The learning efficiency (time and success) is also usually better. Minimal manuals seems to help participants learn how to learn.

4.4.4 Evaluating the Minimal Manual

After using the manual, it is important to evaluate if and why the minimal manual worked or not. Some questions to consider in this evaluation are:

- Do learners get started faster?
- Do they coordinated attention better?
- Do they made fewer errors, in particular, errors the minimal manual targeted and trained against?
- Do they made better use of error recovery methods?
- Do they made better use of the training manual for later reference?

4.4.5 Inference and Rehearsal

Learner inference can play a productive role in instruction. There are three types of inference, to consider:

- **Analogy** is the inference from a specific example to a specific procedure;
- **Proceduralization** is the inference from a general explanation to a specific procedure;
- **Instantiation** is the inference from a general procedure to a specific procedure.

4.4.6 Developing Minimal Manuals

Carroll wanted to design a training manual in the self-instruction genre but one that allowed users to get started doing recognizably real work, one that de-emphasized reading in favor of action, and one that helped learners to avoid making errors and to recognize and recover from errors committed.
Better training material can be produced by:

- presenting real tasks that learners already understand and are motivated to work on;
- helping them to get started rapidly on these tasks;
- allowing them to rely on their own reasoning and improvising;
- reducing the instructional verbiage they must passively read;
- organizing material to support skipping around and to facilitate the coordination of attention between the training and the system;
- and addressing important user errors.

4.5 Task Intelligence

The exploration of approaches to provide systems with task-intelligent support to real users is of importance for minimalist instruction manuals. Systems that know about the tasks learners already understand, and use this knowledge to help guide learning can be very valuable.

Three different approaches, believed to be mutually reinforcing, can be developed:

- a scenario machine – the system allows the user with very few options and can therefore more intelligently diagnose and advise errors;
- a smart help – utility for flexible task-intelligent error recovery support;
- and a task mapping tool – incorporation of task intelligence directly into the design of the system interface itself, to facilitate the basic tasks users can understand based on their prior knowledge, to integrate and coordinate error recovery support and other on-line information with the organization of user tasks, and to ensure better that the user's reasoning about the system will more often be productive than misleading.

Future research on minimalist instruction must explore the incorporation of task intelligence in more effective ways.
4.6 Developing Minimalist Technology

Minimalist instruction tries to capitalize on what the learner already knows and on transforming possibilities that are inherent to the activities that the learner already wishes to undertake.

The fundamental strategy is to minimize the obtrusiveness of the instruction and other support, and to allow familiar tasks and situations and the learner's own reasoning to permit the intellectual transformations of sense-making.

There is a need to know more about learning in the context of meaningful and self-initiated tasks, in the context of error recovery, in the context of coordinating multiple, rich sources of information, and arenas of interaction.

Designing minimalist instruction requires a more detailed understanding of the instructional domain and an orientation toward producing less instructional volume whenever possible, in comparison with a systematic instructional curriculum.

4.6.1 Science and Design

The concrete goal of a design theory is to help people design better solutions, better interfaces, better instruction, by understanding how we do what we do in design so that we can do it deliberately and repeatedly in diverse and novel situations. Moreover, to externalize the understanding of design practice so that it can be taught to others and work with it directly to improve it. The challenge is to get from the why to the how.

The perspective of design-based theory conceives of the relation between science and design not as “one way” and deductive but as interactive and reciprocal. Design-based theory usually develops through a “task-artifact cycle”. People want to engage in certain tasks. In doing so, they make discoveries and incur problems; they experience insight and satisfaction, frustration and failure. Analysis of these tasks is the raw material for the invention of new tools, constrained by technological feasibility. New tools, in turn, alter the tasks for which they were designed, indeed alter the situations in which the tasks occur and even the conditions that cause people to want to engage in the tasks. This creates the need for further task analysis and, in time, for the design of further artifacts, and so on.

To work within the framework of design-based theory, scientists must be able to understand, indeed to generate, the examples, the designed artifacts that are the intellectual currency of the domain. And beyond this, they must develop tools for interpreting and working more effectively with artifacts.
4.6.2 Technology Transaction

The horizon for minimalist instructional technology is real design and product development and careful analysis and interpretation of what goes on in the design process and in what results.

Design is already well under way when it officially “begins”. Designers are living within a design zeitgeist: a set of shared values, goals and assumptions, commitments to practice. It is unrealistic to think that this context can be discarded merely because a new technology becomes available.

Technology development takes place through a series of small steps and depends on identifying and exploiting niches in which a new technology can survive and develop. Carroll states that: “Simple things are hard to keep secret, and complex things like technology are hard to give away”

4.6.3 Minimalist Know-How

Design is a highly creative process and is difficult. Design is inventing something and cannot be both effective and mechanical. Seriously taking checklists is perhaps the most typical and debilitating design fallacy. There three things that are useful to inventors: having a goal, having an inventory of good examples, and having empirical techniques to assessing them.

4.7 Principles for Designing Minimalist Documentation

The major design principles followed by authors of minimalist instruction material were the minimalist principles proposed in [van der Meij and Carroll, 1998]:

- **choose an action-oriented approach**: provide immediate opportunity to act, encourage and support exploration and innovation, respect the integrity of the user’s activity;

- **anchor the tool in the task domain**: select or design instructional activities that are real tasks, the components of the instruction should reflect the task structure;

- **support error recognition and recovery**: prevent mistakes whenever possible, provide error information when actions are error prone or when correction is difficult, provide error information that supports detection, diagnosis, and recovery, provide on-the-spot error information;
• **support reading to do, study and locate**: be brief and don’t spell out everything, provide closure for chapters.

This chapter reviewed the fundamentals of minimalist documentation based on [Carroll, 1990; Carroll, 1998], the original (and best) references on minimalism.
5

Research Problem

Good quality documentation is crucial for the effective reuse of object-oriented frameworks. To fulfill the needs of different audiences, framework documentation combines several kinds of documents and contents that must be produced, organized, and maintained consistent. But documenting a framework is a non-trivial task that comes with many problems, often resulting in hard and costly, specially when not supported by appropriate tools and methods.

This chapter defines the set of key problems of framework documentation addressed by this dissertation, and justifies their importance by discussing to which extent they are or not addressed by related work. The thesis statement of this dissertation is then defined and explained in detail. The chapter concludes by presenting the methodology devised to validate the thesis.

5.1 Research Focus

Software engineering is an engineering discipline concerned with all aspects of software production, namely theories, methods and tools for cost-effective software development [Sommerville, 2000]. Software reuse is an
established sub-discipline of software engineering specially concerned with the aspects of developing software systems starting from existing software artifacts rather than building them from scratch [Naur and Randell, 1968].

There are many different kinds of reusable software artifacts and reuse techniques (Section 2.2.1), object-oriented frameworks being considered one of the most powerful (Section 2.2.3) because they enable large-scale reuse (Section 2.2), among other benefits (Section 3.3.4). However, the development of software based on frameworks (Section 3.1) can be problematic (Section 1.2) due to the many issues that must be addressed when building (Section 3.2.3), using (Section 3.3.3), and documenting a framework (Section 3.4.11).

This dissertation investigates the problem of documenting frameworks, a problem with many different issues (Section 3.4.11), ranging from general issues inherited from technical documentation (page 93) and software documentation (page 95), to framework-specific issues related with the quality of the documentation (page 96), the maturity of the documentation process (page 98), and the appropriateness of the supporting tools (page 100). Figure 5.1 summarizes all these issues organized by categories, c.f. described in Section 3.4.11.

<table>
<thead>
<tr>
<th>General documentation issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technical</td>
</tr>
<tr>
<td>Quality assessment</td>
</tr>
<tr>
<td>Software</td>
</tr>
<tr>
<td>semantic consistency, appropriate documentation environments, knowledge acquisition, programmer's attitude</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Framework-specific documentation issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
</tr>
<tr>
<td>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</td>
</tr>
<tr>
<td>Process</td>
</tr>
<tr>
<td>Lack of standards, guidelines, cost, user-centered process, documentation reuse</td>
</tr>
<tr>
<td>Tools</td>
</tr>
<tr>
<td>Authoring tools, contents synchronization, extraction tools, browse and retrieval facilities</td>
</tr>
</tbody>
</table>

Figure 5.1 Enumeration of framework documentation issues.

The focus of this dissertation is on the documentation issues specific to frameworks. Although the dissertation doesn't intend to address the still open issues related with technical documentation and software documentation, such issues cannot be ignored and therefore they will be considered whenever appropriate.
5.2 Related Work

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.

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 communicating usage information to help new users quickly learn 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.

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.

In the 1980s, minimalism has emerged as an approach to technical documentation that leverages task-orientation in order to achieve easier and faster learning. Research proved repeatedly the effectiveness of minimalist documentation, and therefore, this dissertation investigates whether minimalism is able to provide means for achieving framework documentation easier to use, and easier to understand.

The analysis of the related work is organized in three main areas: software documentation, framework documentation, and minimalist documentation. The areas are discussed individually, and for each one is presented related work that seems to be capable of alleviating some of problems raised by the issues shown in Figure 5.1, mentioning exactly which issues are supposed to
be addressed by the work.

5.2.1 Software Documentation

Good software documentation benefits every software development project, specially large projects. Ad-hoc management of software documentation can consume a high proportion of the software process costs, directly in its production, and most importantly, indirectly in the usage failures and errors originated by omissions and errors in the documentation [Sommerville, 2000].

The production of documentation for a software project often starts well before the product development, with strategic documents, requirements specifications, drafts, etc. The documentation usually covers both the development process, with schedules, conventions, guidelines, memos, etc., and the product, usually separated in user documentation, and system documentation.

User documentation describes the system from the user’s point of view (possibly for many different kinds of users), a black-box view of the system, and explains how to use it.

Research on user documentation started many years ago, in the mid 1970s, as an answer to the needs of delivering instructional documentation to teach how to use software systems. The systems approach of late 1970s provided detailed sequences of instructions, emphasizing structural decomposition and completeness of information, but completely ignored the context of use, thus leading to de-coordination between the system and its documentation, and resulting difficult to use for unexperienced users. After that, the work on user documentation started centering on the user needs and goals.

In 1983, the 53-pages book *Producing Quality Technical Information* (PQTI) [IBM Corporation, 1983] introduced a quasi scientific, analytical approach to help IBM technical writers on evaluating the effectiveness of user documentation, through the identification of nine quality requirements, and a checklist of 29 quality characteristics. For more detail on PQTI see page 93. Due to its success, a revised edition of PQTI, named *Developing Quality Technical Information* (DQTI) was published in 1997 [Hargis, 1997], PQTI was reprinted by ACM [Dean, 2002], and a second edition of DQTI is planned for 2004 [Hargis, 2004].

Another important book is *The User Manual Manual* [Bremer, 1999], which explains the process of writing good user manuals: a step-by-step guide illustrated with examples to plan and create manuals.

Agile processes for software development [Agile Alliance, 2001] are very restrictive with documentation suggesting that: “documentation should be
lean and mean” [Ambler, 2003]. Considering the importance of
documentation for the success of software projects, Andreas Rüping
[Rüping, 2003] advises developers on how to produce lean and lightweight
software documentation, using the patterns format. Although not focused
on documentation, the work on “Agile Modeling” [Ambler, 2002] is very
relevant to the process part of this dissertation work, because the
problematic of creating cost-effective models is very similar to the
problematic of this dissertation, which is to create cost-effective
documentation for frameworks.

Good structure is an important quality aspect of documentation. The IEEE
standard for user documentation [IEEE, 2001] proposes a structure for user
documents, identifying the components of a document, organized in
desirable and essential features. The standard is mostly used as a guide.

Although extremely valuable and useful for this dissertation, all the work
referred above has the drawback of being targeted for general technical
documentation, and thus its application to framework usage documentation
requires adaptation.

System documentation describes the system from the developer’s point of
view, a white-box view of the system, and explains how it works.

The objective of system documentation is to preserve information gathered
and produced during all phases of development, from requirements analysis
to final acceptance testing. It usually includes requirement documents,
ar Constitutional and design decisions, program source code, program
explanations, test plans, and maintenance guides. Documentation is of
critical importance for system maintenance and evolution, specially the
design, implementation and testing, because they support program
understanding, the ability to understand and maintain the system by current
and future developers.

The preservation of consistency between different representations
(documents, source code, models) when the system changes is a problem
common both to system documentation and framework documentation, as
described before in “Semantic consistency”, on page 95, and in “Managing
consistency and redundancy”, on page 97, respectively.

Literate programming systems and other alternative techniques, described in
Section 3.4.10, promise to be very useful to alleviate the consistency
problem of simple documents with source code, but they are not sufficient
for framework documentation. In addition to simple documents, such as
framework overviews (Section 3.4.3), framework documentation also uses
complex structured documents, such as cookbooks (Section 3.4.4) and
pattern instances (Section 3.4.5), which include and reference not only
source code, but also models (Section 3.4.6), which poses more consistency difficulties.

In addition, most literate programming systems (such as the \texttt{WEB} [Knuth, 1983], \texttt{CWEB} [Knuth and Levy, 1994], \texttt{noweb} [Ramsey, 1994], \texttt{nweb} [Briggs, 1993], \texttt{dotnoweb} [Sousa, 1999]) simply don't integrate with modern development environments, source code editors, or software engineering tools, and they don't interoperate well with each other, to mention only the two most restrictive characteristics of literate programming systems that negate their \textit{real-world usage} and wide adoption.

These integration problems were experienced by the author in two software development projects [Pires et al., 1999; ParadigmaXis, 2001] using the \texttt{dotnoweb} system. Despite the reduction of programming productivity, globally the usage of literate programming was considered advantageous.

\texttt{Leo} is a promising recent development [Ream, 2003] that provides a modern environment for editing literate programs, and is compatible with the \texttt{noweb} and \texttt{CWEB} files. The system is still under development but seems to be a good attempt to produce a more widely applicable and usable tool.

\begin{flushleft}
\textbf{Elucidators}
\end{flushleft}

Elucidative programming (page 92) [Nörmärk, 2000a] is an attractive variation of the original literate programming that favors the satisfaction of the major needs of software engineers (program understanding, online presentation, source code preservation) at the cost of separating source code files from documentation files. In order to preserve the semantic consistency, the relationships between source code and documents are managed by automatic mechanisms. By keeping source code intact (almost [Nörmärk, 2000b]), and using XML-based documents, elucidators (i.e. elucidative programming tools) integrate well with source code editors and tools thus obviating the integration problems of traditional literate programming systems.

In order to conveniently support all the requirements of framework documentation, the Java elucidator [Nörmärk et al., 2000] has, however, some internal limitations. The documents (\texttt{edocs}) are limited to have only simple structures based on chapters, sections, and paragraphs. The relationships between source code and documents are also limited to allow only three kinds of relationships: inside documents, inside code, and between source code and documentation. An additional limitation is the lack of support for integrating contents in other representations, of which the UML is the most important.

Tools for supporting framework documentation should be able to accommodate \textit{more complex structures}, and more importantly, to support \textit{different structures}, possibly defined by the writer, without following a pre-
defined tool format. In short, the major limitation of elucidators for usage in this dissertation work (at least the version for Java) resides on the limitations of the internal model used to integrate documents and source code.

In order to obviate this apparently small problem, the author and Thomas Vestdam, one of the developers of the Java elucidator, have collaborated [Aguir and Vestdam, 2001; Aguiar and Vestdam, 2002] with the goals of flexing the elucidator internal model, to support the integration of new kinds of contents, and also to prototype the integration of the Java elucidator in the Borland’s Together tool [Borland, 2003]. The internal model has been considered too difficult to evolve due to the de-localized modifications it would require. The integration in Together was concluded by Vestdam [Vestdam, 2003].

Since the Berners-Lee proposal in 1989 for a “distributed hypertext system” for the “management of general information (...) at CERN”, the exchange of information gained a new dimension. Web-based systems are now the rule for exchange information, both locally and remotely.

In addition to paper-based documentation, most of the documentation delivered with software systems is now online documentation, ranging from simple textual files served by hypertext-based help systems to complete online documentation sets. The advantages are several, including: low cost, immediate accessibility, always up-to-date information, search and query facilities to quickly locate information, attractive presentation, multiple navigation, and multimedia. Effective and attractive online documentation increases the speed with which users acquire proficiency.

However, the paper-format is still important and have its advantages: paper is tangible, more comfortable to read, customizable, off-line and off-screen, and is calm. Research revealed that people retain more if the information is presented as printed text rather than displayed on a screen.

One of the problems with the production of online documentation (web-based), is that web browsers are not capable of editing pages, and don’t support yet the rich structuring, navigation, and annotation features [Vitali and Bieber, 1999].

A WikiWikiWeb, or simply a Wiki, is a very innovative and appealing collaboration tool capable of presenting and editing online information. A Wiki can be defined as a web platform for the cooperative edition of documents, through which everyone can edit any page, using a simple web browser. By simply invoking an Edit command available on the page, often at the top or bottom, the reader is allowed to edit the page [Cunningham, 1999]. After saving, the modifications done will be uploaded to the Wiki server and will be immediately made available online.
A Wiki normally uses a very simple markup language to support text formatting, and a simple mechanism based on Wiki names to automatically link pages. A Wiki name is a name that joins two or more capitalized words, such as AnExample, or WikiName. Despite its simplicity, Wiki names are very powerful because they provide a kind of dynamic-linking mechanism [Bodner et al., 1999; Bodner and Chignell, 1999], where the link is not statically defined, but calculated on the fly, based on the user context (for example), what supports the notion of adaptive web pages. In addition, other kinds of linking can be defined using lexical conventions, such as prefixes, suffixes, and name patterns, in general.

There are a lot of Wiki engines available to use and install [Cunningham, 1999]. Due to the attractiveness of Wiki engines, specially in the software community, they are sometimes used to produce informal software documents (drafts of designs and implementations, design-trade-off discussions, requirements gathering, etc.). Wiki documents are open, evolve incrementally and organically, are easy to edit and organize, promote convergence of contents and consistency of terms, are tolerant, and are easily observable by other users.

In terms of software documentation requirements, among the many different implementations of Wiki engines, we can find some with features to support software development, such as: bug tracking, tests, and source code formatting [Cronin and Barnett, 2003; Jugel and Schmidt, 2003].

However, the existing Wiki engines simply support language-specific formatting of text directly written or copied-pasted from source code files that were marked-up as code (Java, C++, SQL, etc.); as soon as the code changes, the document becomes inconsistent (see “Semantic consistency” on p. 95). No Wiki has been found supporting more useful features for preserving the semantic consistency between source code and documentation, as those provided by the software documentation techniques described in Section 3.4.10.

A Wiki-based environment specifically enhanced for documenting software would help alleviate the issues related with knowledge acquisition, integration of documentation environments in development environments, and the lack of appropriate authoring tools. For all these reasons, Wiki engines are a promising tool for supporting framework documentation, and software documentation in general.

The hierarchical nature of XML documents is very useful to implement structuring mechanisms for software documents. There is abundant work applying XML to software documentation, which include representations of source code (JavaML [Badros, 2000], cppML [Mamas and Kontogiannis, 2000], srcML [Collard et al., 2002]), UML [AlphaWorks, 1999], mechanisms
for integrating heterogeneous documents from different sources [Hartmann et al., 2001; Gupta et al., 2002; Anderson et al., 2002], and literate programming systems [Walsh, 2002; Coates and Rendon, 2002].

XML is an open format, it is widely available in many platforms, and provides powerful querying capabilities with XML standards and tools, such as XSLT [World Wide Web Consortium, 1999] and XQuery [World Wide Web Consortium, 2002]. Therefore, XML provides a good means for integrating the heterogeneous contents of framework documentation: source code, models, and documents.

In order to increase the productivity of the documentation process, Sametinger suggests reusing documentation, in the same way we reuse object-oriented software, by applying object-oriented technology for documentation, too [Sametinger, 1994]. The documentation scheme for object-oriented software systems proposed organizes documentation in six parts, along two dimensions (Figure 5.2). One dimension is related with the static and dynamic aspect. The other dimension is related with the intended use of the information: overview, for reusers and maintainers; external view, for reusers, and an internal view, primarily intended for maintainers.

<table>
<thead>
<tr>
<th>static view</th>
<th>overview</th>
<th>external view</th>
<th>internal view</th>
</tr>
</thead>
<tbody>
<tr>
<td>static overview</td>
<td>static overview</td>
<td>class interface description</td>
<td>class implementation description</td>
</tr>
<tr>
<td>dynamic view</td>
<td>dynamic overview</td>
<td>task interface description</td>
<td>task implementation description</td>
</tr>
</tbody>
</table>

**Figure 5.2** Documentation scheme for object-oriented systems [Sametinger, 1994].

With this scheme, Sametinger highlighted the importance of structuring the documentation of object-oriented systems according to the different needs of different kinds of users. This structure is not appropriate for the overall framework documentation, but can be used as starting point for structuring the low-level contents of framework documentation.

Documenting large systems is significantly different from documenting small systems. In [Tilley, 1993] Tilley uses virtual subsystem stratifications to represent multiple abstract views of a software system. The virtual subsystem is a four dimensional hypercube. The dimensions are: artifacts, layers, views, and time (Figure 5.3).
Figure 5.3 Virtual subsystem stratifications [Tilley, 1993].

Considering that frameworks are usually very large systems, and that framework documentation requires a good organization of contents, this multidimensional virtual hypercube constitutes an interesting feature.

The related work in the area of software documentation is vast, so only the work considered more relevant for the issues of framework documentation addressed in this dissertation has been presented.

5.2.2 Framework Documentation

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 (Section 3.3.2). Therefore, framework usage documentation must mix information both from typical user documentation and system documentation, a situation that poses integration and organization difficulties.

Most of the work directly related with framework documentation was already presented in Section 3.4. That work and some additional efforts are analyzed here in relation to the issues previously enumerated.

Background The research conducted for this thesis was primarily motivated by the Mohamed Fayad's article entitled “Object-Oriented Application Frameworks” [Fayad and Schmidt, 1997], which points trends for future research on object-oriented frameworks.
The first important papers about frameworks were authored by Ralph Johnson with other colleagues [Johnson and Foote, 1988; Johnson and Russo, 1991; Johnson, 1992; Beck and Johnson, 1994]. However, the most complete source of information about object-oriented frameworks technology is authored by Fayad, Johnson, and Schmidt, in a set of three volumes that compiles several articles and chapters from different authors [Fayad et al., 1999a; Fayad et al., 1999b; Fayad and Johnson, 2000]. Older work can be found in TaliGent's publications [TaliGent Inc., 1993; TaliGent Press, 1994; TaliGent Press, 1995; Cotter and Potel, 1995] and [LAC+95], to mention a few.

In the article “Documenting frameworks using patterns” [Johnson, 1992] Johnson describes a documentation technique that organizes patterns in a pattern language, in a similar way that recipes are organized in a cookbook (Section 3.4.4). 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 in Chapter 3, from Section 3.4.3 (p. 74) to Section 3.4.9 (p. 86).

An extensive list that includes these and other techniques can be found in the work of Pascal Rapicault [Rapicault, 2002], where they are classified based on the kind of dependencies they represent: structural or behavioral.

Greg Butler’s work on framework documentation is perhaps the most complete and comprehensive existing theoretical work, in the opinion of the author. Butler introduces the concept of reuse cases (page 71) to catalog the properties of the existing documenting approaches [Butler, 1997]. Butler et al. define the concept of documentation primitive, an elementary unit of documentation, and defines a “task-oriented framework for framework documentation” that relates documentation approaches with documentation.
primitives [Butler et al., 2000]. In [Butler and Denommée, 1999] (similar to [Butler and Dénommée, 1997]), is proposed a small set of guidelines to document a framework. In summary, Butler suggests combining a framework overview (Section 3.4.3) 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 research of this dissertation.

Research and experience have proved the effectiveness of some techniques even when used in isolation, namely: cookbooks in [Krasner and Pope, 1988; Apple Computer, 1986]; patterns in [Johnson, 1992; Beck et al., 1996; Prechelt and Unger, 1998]; and examples in [Shull et al., 2000]. 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 [Kirk et al., 2001; Kirk, 2001].

[Meusel et al., 1997] proposes 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 (page 71): selecting, using, and extending a framework. The model is based on the pyramid principle [Minto, 1991], 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., 2000] that reports on the use of open hypermedia systems to keep framework cookbooks up-to-date and consistent with framework source code.

Another area with related work, not so relevant, is on approaches for framework design, which propose complementary representations that can be used to describe framework design. To reduce the complexity of frameworks it is useful to follow a separation of concerns approach. The work [Silva et al., 2000] proposes describing a framework using the composition of concern-specific design patterns. Dirk Riehle’s role modeling
approach for framework design also uses a separation of concerns approach, but instead of being centered on domain semantics, uses the role concept as a key element for modeling collaborative behavior of the framework objects [Riehle, 2000]. Sherif Yacoub’s pattern-oriented approach for analysis and design of software uses the pattern concept as the key modeling element [Yacoub, 1999; Yacoub, 2003]. Like other high-level design representations (Section 3.4.5, Section 3.4.6) these approaches are useful for reducing the complexity of framework design descriptions, by introducing concepts (concerns, roles, and patterns) at an higher level of abstraction than classes.

In addition, to satisfy the personal needs of different users, framework documentation should also support different learning strategies. For that purpose it is important to consider the work on program understanding carried on in the area of software maintenance and evolution. The most typical cognition models are the following: top-down, bottom-up, and knowledge-based [von Mayrhauser and Vans, 1994; Storey et al., 1999].

### 5.2.3 Minimalist Documentation

In few words, minimalism is based on the principle that people don’t want irrelevant information when trying to learn how to perform a particular task. Using different words, Brockmann considers that the basic message of minimalists is: “Get out of the way of the learner as much as possible” [Brockmann, 1990].

The fundamentals about minimalist documentation were already presented in Chapter 4, which can be regarded as a summary of the first book on minimalism, named “The Nurnberg Funnel” [Carroll, 1990]. A second book on minimalism was published in 1998, named “Minimalism Beyond the Nurnberg Funnel” [Carroll, 1998], which compiles a series of chapters from different authors summarizing the state-of-the-art of minimalism, reflecting about problems and solutions found when applying minimalism, and posing new challenges for the near future. Since 1998, a lot of work has been published reporting experiences of applying minimalism. For example, a few articles commenting the second book can be found in [Haramundanis, 1999].

Considering that the role of minimalism in this dissertation is to be applied to framework documentation, and not to be researched by itself, this section presents only relevant work that has also applied minimalism to document frameworks, or at least to document software.

As a design theory, the minimalist instruction only defines a set of design principles, which must be applied to concrete situations. In [Oatey and Cawood, 1997], the authors identify various minimalist techniques, which,
although conflicting with each other, together provide a setting for designing computer documentation. Using these techniques, the application of minimalism involves making trade-offs, instead of following a set of prescriptive techniques. The techniques are the following: word and page count, duplication, selective documentation of facilities, elaboration, task orientation, guided exploration, error recovery, and access. The common-sense for writing minimalist documentation is to be brief, which conflict with the ideas of supporting user exploration and inference. Authors must decide based on users’ needs and the purposes and context of the documentation.

The ideas of the minimalism were initially applied in a case study for the development of a curriculum to introduce users to the Smalltalk object-oriented environment [Rosson et al., 1990]. A set of example-based learning scenarios were developed aimed at supporting real work, getting started fast, reasoning and improvising, coordinating system and text, and satisfying other minimalist principles. The results of the case study were encouraging, and the programmers were able to progress 1-2 orders of magnitude faster than the usual. The work was also reported in [Rosson and Carroll, 1996].

Another application has been to produce instruction material to teach programmers learning object-oriented design through scaffolded examples.

The first reference to minimalism found by the author in work related with frameworks was in [Johnson, 1992], and is cited here: “Most users of a framework want to know as little as possible about the framework. (...) This is similar to the minimalist instruction of Carroll(...), which tries to show how to solve particular problems and make a system useful to a user as soon as possible”.

My own experience and observations of others using graphical frameworks motivated further research on the application of minimalism to framework documentation.

Ian Chai in [Chai, 1999] has investigated how effectively new users have learned to use a framework, when using three different philosophies of documentation: step-by-step documentation, minimalist documentation, and pattern-based documentation. Based on experiments, Chai observed that minimalist documentation helped people get the task done faster, while patterns helped people to understand better the internal workings, but he suggests that more research needs to be done to confirm the results. In his work, the focus was on comparing the minimalist documentation produced with other styles of documentation, and not on finding effective ways to produce it.

Based on Carroll’s work on minimalism, Kasper Østerbye in [Østerbye,
1995] discusses how reference material, tutorial and run-time error messages are combined into a hypertext-based documentation for the BetaSIM framework with the support of the noweb literate programming system [Ramsey, 1994]. As a result of applying minimalism, the tutorial is based on full running examples that enable users getting started fast, and the framework integrates extensive run-time checks that support error recovery. In addition, the documentation has hyperlinks between the reference manual and the examples, and the runtime errors are hyperlinked to the manual and examples. Østerbye argues that the resulting documentation follows almost all the principles of minimalism. Although he explains parts of the process used to produce the minimalist parts of the documentation, the process and the example presented is not easy to generalize to other frameworks.

In [Silveira et al., 2003] is presented a semiotic engineering method for building online help systems. One of the fundamental points of the method is the layering technique of minimalist documentation [Farkas, 1998], which means providing extra information via pop-up windows, tabs, buttons, tooltips, or balloon help. Using the layering technique, the user initially accesses small pieces of information about the topic selected, and is able to drill-down to the level of detail desired. The usage of this layering technique is very useful also for framework documentation, due to the wide range of abstraction levels of the contents, ranging from abstract architectural descriptions to concrete source code implementations.

5.2.4 Summary

Despite the existing work in the areas of framework documentation, software documentation, and minimalist documentation, there are still obstacles to an effective production and usage of framework documentation.

To enable a cost-effective production of framework documentation, I highlight the need to define and build tools capable of integrating the many different contents typical of framework documentation. Such tools must be open and widely available, easy to use, and easy to integrate in common development environments.

To reduce the effort of learning frameworks from documentation, I stress the need to apply the ideas of minimalism, namely: task orientation, learning by doing, to support guided discovery and self-exploration, allow for reading in any order, fading and layering contents, in order to reduce the risk of the reader become lost in the documentation, and to help her quickly locate what she needs.
5.3 Research Issues

Considering the extension and complexity of the problem, this dissertation doesn’t intend to address all the issues previously identified for framework documentation, but to focus only on a subset of them. However, the strong interdependency of some issues may influence the research to address other issues as well.

All the issues previously presented in Figure 5.1 are again tabulated in Figure 5.4, now annotated with the kind of investigation intended to do in the scope of this dissertation work (intentional, occasional), and the respective level of priority (high, medium, low).

<table>
<thead>
<tr>
<th>Category</th>
<th>Issue</th>
<th>Scope</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>General documentation issues</td>
<td>Quality assessment</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>Technical</td>
<td>Semantic consistency</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>Software</td>
<td>Appropriate documentation environments</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Knowledge acquisition</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Programmer’s attitude</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>Framework-specific documentation issues</td>
<td>Combining prescriptive and descriptive information</td>
<td>*</td>
<td>**</td>
</tr>
<tr>
<td></td>
<td>Managing consistency and redundancy</td>
<td>*</td>
<td>***</td>
</tr>
<tr>
<td></td>
<td>Good organization of contents</td>
<td>*</td>
<td>***</td>
</tr>
<tr>
<td></td>
<td>Graded presentation of contents</td>
<td>*</td>
<td>**</td>
</tr>
<tr>
<td></td>
<td>Several entry points and traversal paths</td>
<td>*</td>
<td>**</td>
</tr>
<tr>
<td></td>
<td>Multiple views and outputs</td>
<td>*</td>
<td>**</td>
</tr>
<tr>
<td></td>
<td>Contents integration</td>
<td>*</td>
<td>***</td>
</tr>
<tr>
<td>Product</td>
<td>Lack of standards</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Guidelines</td>
<td>*</td>
<td>**</td>
</tr>
<tr>
<td></td>
<td>Cost</td>
<td>*</td>
<td>***</td>
</tr>
<tr>
<td></td>
<td>User-centered process</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Documentation reuse</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>Process</td>
<td>Authoring tools</td>
<td>*</td>
<td>***</td>
</tr>
<tr>
<td></td>
<td>Contents synchronization</td>
<td>*</td>
<td>***</td>
</tr>
<tr>
<td></td>
<td>Extraction tools</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Browse and retrieval facilities</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

**research scope** * intentional = occasional  
**priority** * low ** medium *** high

* Figure 5.4 Prioritization of framework documentation issues for research.

In order to achieve practical results useful for framework developers and reusers, the research aims to address almost all important issues, instead of investigating only one or two isolated issues very deeply but whose
contributions won’t allow an immediate and practical application. Therefore, this dissertation intends to address the issues annotated as intensional. Issues not annotated as well as those annotated as occasional are not intended to be addressed.

5.4 Research Goals

This research investigates answers to the following questions, as previously mentioned in Section 1.4:

- What are the attributes of “high-quality framework documentation”?
- How to streamline the production of “high-quality framework documentation”?
- How to organize and present the documentation of a framework in order to improve its understandability and usability, especially by first-time users?

5.4.1 Primary Goal

The most important problems of framework documentation were identified, and we know of partial solutions for many of them. What is still lacking is to “bundle” all the good practices in a simple and flexible approach, supported by open and interoperable tools, so that we can use it, test it, refine it, customize it for the specific needs of our frameworks, and love it, or hate it.

Therefore, the primary goal of the research is to define an approach for producing high-quality framework documentation capable of reducing the costs typically high associated with its production. The most important requirements of such approach are the following:

- easy to use by all kinds of authors (developers, technical writers, testers, etc.), so that the activity of documentation can act as a means to improve development productivity and quality, instead of being considered an obstacle, as happens in many development environments;
- flexible enough to be easily adaptable to the needs of different projects and development environments;
- economical, to reduce the typical high-costs associated with the production of good quality documentation.
5.4.2 Secondary Goal

Taking advantage of the qualities and research on minimalist documentation, and the benefits of an adequate combination of task-oriented information with design information in framework documentation, the secondary goal is to define a way to organize and present framework documentation so that it can improve the understandability and usability of the respective framework. To achieve such goal, the documentation should be organized and presented in a way that fulfills the following requirements:

- easy to use by all kinds of audience, so that the readers can locate, read and understand, with little effort, the parts of the documentation containing the knowledge they need to perform their particular engineering tasks;
- task-oriented organization of contents, to favor searches on how-to-do information over how-it-works information.

5.5 Thesis Statement

This section presents and explains the thesis statement of this dissertation. It starts with the initial version defined in the beginning of the dissertation, and after precisings the meaning of the terms considered ambiguous, the revised final version is then presented.

5.5.1 Preliminary Version

The initial version of the thesis statement of this dissertation was defined in Section 1.6 as the following:

- The minimalist approach to framework documentation makes the production and usage of framework documentation easier than is possible with traditional approaches.

This initial thesis statement uses terms whose meaning is not precise, and therefore lead to questions that deserve further discussion:

- what is the scope of “production” and “usage”?
- what is meant by “easier”?
- which are the “traditional approaches”?
- “who” does benefit from these improvements?
The following sections examine these questions.

5.5.2 What is the scope of “production” and “usage”?

The minimalist approach proposed in this dissertation covers all the life cycle of framework documentation. In the thesis statement, the term “production” is used in a wide sense, and applies to all the typical activities of framework documentation described in Section 3.4.1, with the exception of the usage activity, which is applied by the term “usage”. The Figure 5.5 shows the clear separation between the activities in scope of both terms.

![Diagram showing separation of "production" and "usage" documentation activities.]

**Figure 5.5** Separation of "production" and "usage" documentation activities.

5.5.3 What is meant by “easier”?

The term “easier” is used in the thesis statement with the sense of “more effective” in coping with the issues of framework documentation summarized in Figure 5.1, and discussed earlier in Section 3.4.11.

Because the term is applied both to “production” and “usage” activities, it is necessary to identify the issues related with both categories of activities. Figure 5.6 shows the list of issues with a classification that reflects the categories of activities on which the issues manifest themselves. In addition to the “production” and “usage” categories, another column is used to mark the issues that depend on “tool-support”.

---

A Minimalist Approach to Framework Documentation
Almost all the issues identified are considered “production”, from which some would benefit from tool support. The “usage” issues are only a few.

<table>
<thead>
<tr>
<th>Category</th>
<th>Issue</th>
<th>P</th>
<th>U</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General documentation issues</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Technical</td>
<td>Quality assessment</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software</td>
<td>Semantic consistency</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Appropriate documentation environments</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Knowledge acquisition</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Programmer’s attitude</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Framework-specific documentation issues</td>
<td>Combining prescriptive and descriptive information</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Managing consistency and redundancy</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Good organization of contents</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Graded presentation of contents</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Several entry points and traversal paths</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Multiple views and outputs</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Contents integration</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>Product</td>
<td>Lack of standards</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Guidelines</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cost</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>User-centered process</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Documentation reuse</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process</td>
<td>Authoring tools</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contents synchronization</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Extraction tools</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Browse and retrieval facilities</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

P - production  U - usage  T - tool-supported

Figure 5.6 Classification of issues based on the activities they are related.

5.5.4 Which are the “traditional approaches”?  
The “traditional approaches” for documenting frameworks are the kinds of documentation techniques and document styles (Section 3.4) more frequently delivered with object-oriented frameworks.

A survey of application frameworks [Yassin and Fayad, 1999] investigated the document types commonly provided with the framework, among other aspects. More than 95 percent of the 21 framework samples provide a framework overview. The second documentation type most commonly
provided are examples, because they are usually used as a second step in the learning process.

In addition to framework overviews and examples, although not referred in the survey, I also consider reference manuals, at least containing the application programming interface, to be another kind of documentation commonly provided with the framework.

Therefore, “traditional approaches” means “framework overview, examples, and a reference manual”. The process and tools used are considered to be general-purpose software documentation ones.

5.5.5 “Who” does benefit from these improvements?

The minimalist approach to framework documentation aims to help documentation managers, authors, and readers. The authors include framework developers and technical writers, while the readers include also framework developers, application developers, and framework selectors (page 70).

The authors use the minimalist approach to produce and organize documentation. Therefore, they benefit from the contributions made by the minimalist approach on the “production” issues, which include: easy to use authoring tools, automatic preservation of contents consistency, contents organization, contents integration, and reduction of costs, to mention a few.

The readers use the documentation produced with the minimalist approach. Therefore, they benefit from the browsing tools that support the approach, and benefit also from the qualities of the minimalist documentation itself, namely: task-orientation, good organization of contents, graded presentation of contents, multiple views and several entry points and traversal paths.

5.5.6 Final version

Based on the previous discussions and refinements, the final version of the thesis statement becomes the following:

- The minimalist approach to framework documentation makes the typical activities of producing and using documentation easier to carry out respectively for documentation authors (and managers), and documentation readers, when compared to traditional documentation approaches based on framework overviews, examples and reference manuals. To provide such improvements, the minimalist approach combines several enhancements over traditional approaches that collectively help reduce the severity of many production and usage issues.
of those presented in Figure 5.6.

5.6 Validation Methodology

The best way of validating the thesis would be to define empirical studies and controlled experiments to compare with other approaches, the "easiness", both for production and usage activities, demonstrated by the minimalist approach to framework documentation, as defined in this dissertation. Due to the effort required, and the operational difficulties of conducting such experiments in the field of software engineering (Section 1.5.1), it was decided to use a case study based approach for evaluating the approach for production and usage activities, and to conduct a small experiment for evaluating the usability of the minimalist documentation produced. The research strategy, including the validation phase, was globally described in Section 1.5.

Therefore, for validation purposes, it was followed a methodology based on the decomposition of the thesis into sub-theses, and its overall validation through the sum of sub-validations of its parts. Another example of following this kind of validation methodology can be found in [Riehle, 2000].

The thesis was divided into two sets of sub-theses, one set for "production" activities, and another for "usage" activities, as defined above in Figure 5.6.

Each sub-thesis pairs a category of activities A with an issue I, and states that "the minimalist approach reduces the severity of the issue I when carrying out activities from category A".

In Figure 5.7 is shown the table of sub-theses that must be validated in order to achieve the validation of the overall thesis of this dissertation. Because the issues are sometimes specific to production or usage activities, there are some cells that don't correspond to a sub-thesis, thus being left blank, instead of being marked with an asterisk (*).

In Chapter 8, this table of sub-theses is used as the base for validating the thesis. Each sub-thesis is argued based on the case studies, the controlled experiment, and lessons learned from them.
<table>
<thead>
<tr>
<th>Category</th>
<th>Issue</th>
<th>Scope</th>
<th>P</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>General documentation issues</td>
<td></td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Technical</td>
<td>Quality assessment</td>
<td>~</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>Software</td>
<td>Semantic consistency</td>
<td>~</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Appropriate documentation environments</td>
<td>~</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Knowledge acquisition</td>
<td>~</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Programmer's attitude</td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Framework-specific documentation issues</td>
<td>Combining prescriptive and descriptive information</td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Managing consistency and redundancy</td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Good organization of contents</td>
<td></td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>
|                                     | 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           |       |     | *   |

research scope: * intentional - occasional
P - production  U - usage  * - sub-thesis

Figure 5.7 Sets of sub-theses to validate (non-blank central cells).
Good quality documentation is crucial for the effective reuse of object-oriented frameworks. To satisfy the needs of different audiences, framework documentation combines several kinds of documents and contents, resulting hard, costly and tiresome to produce, specially when not supported by appropriate tools and methods.

This chapter presents a minimalist approach to framework documentation that aims at being simple and economic to use. The approach, from now on simply referred as the minimalist approach, reuses existing documentation styles, techniques and tools, and combines them in a way that follows the design principles of the minimalist instruction theory. The resulting documentation assumes the form of a minimalist framework manual, a task-oriented manual specially concerned on helping readers getting started fast using the framework rather than on simply explaining how the framework works.

The minimalist approach proposes a documentation model, a documentation process, and a set of tools specially built to support the approach in a wide range of software development environments.
6.1 Requirements for the approach

The most complete and accurate model of the problem understanding and solution design is formed in framework developer's mind at implementation time. But this knowledge is difficult to obtain from framework developers, at the right time ("Knowledge acquisition", on page 95). One of the fundamental goals of the minimalist approach is to alleviate the severity of this problem.

This section presents the functional and non-functional requirements defined for the approach.

6.1.1 Helping on "documenting-while-developing"

To be effective in development, developers need good documentation. To be effective in documenting, developers need to document while developing, a situation that often reduces productivity due to constant switching between development and documentation environments. This paradox is usually solved by not documenting at the right time, but only at the end.

For small frameworks or small teams, personal communication is preferable to documentation, so the lack of documentation only manifests itself as a problem at maintenance or evolution phases. On the other hand, for large frameworks or large teams, the problem manifests immediately, because personal communication in large teams is not effective, and more formal communication channels must be used.

This approach to document frameworks is primarily oriented for framework developers to help them document while they do "their work" (to develop). The resulting documentation is assumed to be web-based and useful for all kinds of audiences: application developers, framework selectors, framework maintainers, framework developers, and developers of other frameworks ("Different audiences", on page 70).

While developing, a framework developer needs to write code, to draw UML diagrams, to browse online documentation, possibly of some parts of its own framework, and to document the framework. When using a shell-based development environment, a developer may use a web browser for reading the documentation, and an editor to write the documentation. When using a modern integrated development environment (IDE), a developer may prefer to use a browser integrated in a view of the IDE, and to write documentation also inside the IDE, to avoid switching applications.

The next two subsections describe the author's vision about what could be a "good environment" both for producing and for using minimalist framework documentation, informally written as two scenario narratives.
6.1.2 When producing documentation...

Templates
"The developer starts by creating a new document based on pre-defined templates, picked up from a list with a set of helpful exemplars of the best kinds of documents used in document frameworks. The list helps on deciding which template to choose by explaining what each template is good for."

Edit-view
After writing a few lines, and doing simple formatting of some parts, she saves the document and (pre)view(s) it in the same environment (tool).

Merging
The developer restarts editing the document, and then she wants to include a fragment of source code in the document by dragging (or marking) the source code lines to the document, or its representation in the hierarchical navigator. The source code included is copied by reference, not by value. This means that the contents are always consistent: when she changes those lines of the source code, their "pseudo-copy" in the document changes too. Next, she does the same inclusion of contents now from a UML diagram, a XML document, and a plain-text document.

Linking
After having merged all these contents, she adds links to the same kinds of contents.

Explorer
An outline of the document shows her where she is in the document, and let her navigate inside it. A documentation explorer enables her to visualize the documents she has in the project repository.

Stratification
After editing many different documents, structured and unstructured, she organizes them: she classifies the documents, and stratifies the overall documentation.

Filters
To change the way the documents are presented in different layers she defines filters using a scripting language (XSLT, Javascript, Python, etc.).

Linking rules
To automatically link, for example, all pattern-instantiations to the corresponding pattern document, she defines a linking rule between the attribute (or element) pattern-name stored in the pattern-instantiation document, to the pattern document having that value in name attribute (or element).

Extensions
To support new notations for representing contents, additional to those already supported (e.g. Java, XML, UML, text), such as Java class files (.class) and her own programming language OPL (Own Programming Language), she needs to provide an extension to the tool."
6.1.3 When using documentation...

Side-by-side

"The developer starts placing the browser for the documentation side-by-side with her source code editor."

Entry-points

She wants to know how to reuse the framework to create an application that does "something". Because it is the first time she uses the framework, she selects the main entry point of the documentation, as the other possible entry-points don't mean nothing to her, by now.

Browsing modes

Then she selects the guided-exploration browsing mode, which won't let her get lost in the documentation, by providing pre-defined sequences of topics to browse.

Task-oriented

After reading the framework overview she confirms that the framework supports what she intends to do, so she looks in the cookbook for a recipe able to help on her particular problem.

Suggestive names

The developer selects the recipe with the most suggestive name for the information she is looking for. She confirms that the recipe contains what she needs.

Adaptive information

Once found, she wants to visualize at once all the documentation related with this recipe that explains how the framework implements the flexibility at the hot spot she needs to customize. To achieve that, she adapts the information presented according to her needs, using the browsing controls available to customize the browsing of the documentation. She changes the browsing mode to self-exploration.

Aspects

She changes the audience (an aspect) to framework developer.

Scope

She adjusts the scope to see more related information at once.

Resolution

She increases the resolution to see information with more detail, more concrete, till the level of source code.

Offline format

After that she has all the information she needs to study. She saves it locally for eventually browsing it offline.

Printable format

She also exports it to a printable format (PDF), so that she can study parts of it offline and off-screen.

Running examples

After running the related examples provided, she downloads some of them considered useful to study."
6.1.4 Functional requirements

The scenario narratives presented before list the major functional requirements envisioned for the minimalist approach in a very informal but compact way.

This section presents a less informal overview of the same requirements and complements them with other important functionality not mentioned before. The functionality is described with the help of use case diagrams [Jacobson, 1994] for a possible documentation environment to support the envisioned minimalist approach.

The first diagram, in Figure 6.1, gives a first impression of the documentation environment. It identifies the actors representing the different kinds of framework audiences, and the areas of functionality they are expected to use: readers only use browsing facilities (Figure 6.2), writers use also edition facilities (Figure 6.3), and documentation managers use, in addition, configuration facilities (Figure 6.4). The diagram represents also the relations between the roles identified: the different kinds of readers, which includes the writers, the different kinds of writers, and the documentation manager.

![Diagram showing documentation environment for the minimalist approach.](image)

**Figure 6.1** Use cases for a minimalist framework documentation environment.

**Browsing facilities**

The browsing facilities are intended to help readers on finding the information they look for in the documentation, by searching, querying, following sequences of topics, and adapting the information to their needs and preferences, as suggested by the minimalist instruction theory (Figure 6.2).
Figure 6.2 Use cases for the browsing facilities.

In concrete, the facilities should enable:

- **adapting the information presented** based on adjustable browsing controls:
  - **zoom in/out** control changes the scope of the information, narrowing on a single topic, or widening to other related topics;
  - **resolution** control changes the amount of information about a topic, i.e. same topic, but different levels of detail;
  - **aspect value** controls set values for specific aspects, such as audience, purpose, type of document, notation, abstraction, granularity, as described in Section 3.4.2.
  - **layer** control changes the set of information presented, which defines a particular set of pairs (aspect, value) that identifies a hyperplane of information on the multidimensional space defined by the aspects;

- **searching information** based on free text search of terms, words, etc.;

- **querying information** based on the evaluation of predicates against the information structure;

- **browsing modes** ranging from a very restricted mode such as the “training-wheels” mode, to a “guided-exploration” mode, and to a completely free “self-exploration” mode.

- **exporting information** both for offline (save) and off-screen view (print).
Edition facilities

The edition facilities are intended to help writers on creating and organizing documents (Figure 6.3).

![Edition facilities diagram]

Figure 6.3 Use cases for the edition facilities.

In concrete, the facilities should enable:

- **creating documents** using:
  - *template documents* that provide the structure for different types of documents;
  - *formatting features* that are simple to use but help improve the readability of the document;
  - *including contents* from other documents or files, possibly represented in different notations, of which the most important to support are: source code, UML diagrams, XML documents, and plain-text files;
  - *linking contents* internally in a document, and externally to other documents and files, as well.

- **(pre)viewing documents** in close distance to the editing;

- **organizing documents** implicitly with links, or explicitly through sequences of documents or weavings of topics in documents.
Configuration facilities

The configuration facilities are intended to help documentation managers on controlling the documentation process, and the quality and organization of documentation produced (Figure 6.4).

![Configuration facilities diagram]

Figure 6.4 Use cases for the configuration facilities.

In concrete, the facilities should enable the definition of:

- **template documents**, self-explained, to be easily instantiated by writers;
- **aspects**, the orthogonal dimensions that are used to create a virtual multidimensional documentation structure;
- **layers** to stratify the documentation space according to values of aspects;
- **linking rules** to derive automatic links between different contents based on implicit relationships often hidden in the original contents;
- **filters** to apply when stratifying the documentation in layers;
- **extensions** to the environment to support new notations for representing other kinds of contents.

### 6.1.5 Non-functional requirements

In addition to the functional requirements enumerated before describing *which* functionality should be supported by the approach, it is very important to enumerate the quality attributes considered most important for the approach.
The approach for documenting frameworks must be effective, and to help developers on producing good quality documentation. In addition, the approach must be easy to use, easy to manage, easy to setup, and economic.

Good quality documentation

Good quality framework documentation means documentation that has the following attributes: accurate, modular, well-structured, concise, readable, searchable, and easy to navigate [Arthur and Stevens, 1992].

Effective

The approach is considered “effective” if it is able to produce the intended result, i.e. if it is able to help developers on producing good quality minimalist documentation for frameworks, documentation which being of good quality also helps readers on understanding it.

Easy to use

To be helpful, the approach must be easy to use by all kinds of authors (developers, technical writers, testers, etc.), so that the activity of documenting can be seen as useful, as a means to improve development productivity and quality, instead of being considered an obstacle, as often happens. To be easy to use the approach must be simple, easy to learn, easy to adopt, and adequately supported by tools.

Easy to manage

It is also important that the approach results easy to manage and control along the development life cycle.

Easy to setup

The approach must be easy to setup and configure to be adaptable to the needs of different projects and development environments.

Economic

Most importantly, the approach must be economic, to reduce the typical high-costs associated with the production of good quality documentation. The approach can be considered “economic” if it is “cost-effective”, “pays off” or “provides good value in proportion to the resources spent: money, time, and effort”.

To fulfill all these requirements, the approach must informally provide guidance about what, when, and how to document frameworks in a minimalist way. To reduce the costs, the approach must be supported by appropriate tools able to solve the issues of the overall process of framework documentation, from the initial phases of creation and integration of contents till the last phases of publishing and presentation to target audiences.

After enumerating the requirements, the following sections describe the components of the approach: a documentation model, a documentation process and a set of tools to help make it convenient to use in mainstream development environments.
6.1.6 List of requirements

The requirements earlier presented are summarized below in Figure 6.5, and tagged with an unique identifier (ID) for easy cross-reference in the text.

<table>
<thead>
<tr>
<th>Category</th>
<th>Requirement</th>
<th>Approach</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional requirements</td>
<td>Zoom in/out control</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Resolution control</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Aspects control</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Layers control</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Browsing</td>
<td>Searching</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Querying</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Browsing modes</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Exporting</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Edition</td>
<td>Creating documents</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Templates</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Formatting</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Including contents</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Linking contents</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Preview</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Organizing</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>Templates definition</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Aspects definition</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Layers definition</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Linking rules definition</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Filters definition</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Extensions</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Non-functional requirements</td>
<td>Accurate</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Documentation quality</td>
<td>Modular</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Well-structured</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Concise</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Readable</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Searchable</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Easy to navigate</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Approach quality</td>
<td>Effective</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Easy to use</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Easy to manage</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Easy to setup</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Economic</td>
<td>Cost</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

m - model  p - process  t - tools  * - supported by

Figure 6.5 Summary of requirements for the approach.
6.2 Documentation Model

The documentation model is the core of the approach. The model enumerates, associates, and organizes all the contents considered necessary to produce a minimalist framework manual.

The model supports several types of documents aggregating different kinds of contents. Based on their provenience, the contents of minimalist framework documentation can be grouped in two main categories:

- **framework documentation contents** include contents of framework overviews, cookbooks and recipes, design patterns, design notebooks, contracts, reference manuals, and source code examples, as described in “Different types of framework documents”, on page 74;

- **minimalist documentation contents** include contents of user tasks, usage patterns, task information contexts, error inventories, error recovery guides, and classifications of contents based on the browsing mode allowed (training-wheels, guided-exploration or self-exploration).

With the goal of structuring all the above contents in a powerful and flexible way, the model defines and relates the key concepts involved in minimalist framework documentation.

The model presented in this section is a conceptual model. Implementations of the model would be able to map these concepts to concrete solutions, perhaps using different technologies.

The model is organized in four parts, as represented in Figure 6.6: resources, contents, templates, and structure. The following sections walk through the key concepts defined in each part, starting from tangible concepts, such as projects and resources, and then drilling down to more fundamental concepts, such as layers and aspects.

![Diagram](image)

**Figure 6.6** Parts of the model: resources, contents, templates, structure.
The concepts are presented with the help of conceptual class diagrams represented in UML notation [OMG, 2003].

6.2.1 Resources

The first set of concepts defined by the model are those that map directly to concrete physical resources used in a framework development project, the familiar concepts of projects, resources, folders, and files.

These concepts are represented in Figure 6.7 and described below.

![Diagram of resource concepts](image)

Figure 6.7 Model concepts for physical resources.

**Projects**  Projects contain resources. In the model, projects are used to organize all the artifacts involved in framework documentation: document files, source code files, UML diagram files, library files, API documents, etc. Projects are the biggest unit of modularization for exchanging and sharing framework documentation.

**Resources**  Resources are a common concept to refer to files and folders. The resources of a project usually define a hierarchical structure. A resource can map directly to a location in a storage system, or link to another resource, being a link.

**Folders**  Folders can contain files, links, and other folders. Folders are used to organize the project resources in smaller modularization units, for easiness of management, sharing or exchange.

**Files**  Files are the smallest unit of modularization. Files contain information about the project.

**Links**  Links are resources that point to other resources. Links are used to organize resources logically, without changing their physical location.

**Storage systems**  A storage system stores resources. Examples of storage systems include: file systems, archiving systems (e.g. compressed archive files), version control systems, World Wide Web, or database systems.
6.2.2 Contents

Files contain contents. Contents are units of information used to compose concrete documents. Different types of documents use different kinds of contents, possibly represented in different notations ("Different types of framework documents", on page 74), and include references to other contents.

For example, a recipe usually contains a sequence of steps described in plain text, pictures, fragments of source code taken from examples, and references to other recipes.

These concepts are represented in Figure 6.8 and described below.

![Diagram](image-url)  
**Figure 6.8** Model concepts for contents.

**Documents**

Documents contain contents that are contained in files. Documents are used to logically organize and present contents for an easy comprehension and reading. In addition to other contents, documents use references to help the reader navigate to other parts of the document, or to other documents.

**Contents**

Contents contain units of information. While composite contents can be decomposed in other contents, atomic contents don't. Composite contents may contain recursive hierarchies of contents. The depth of this hierarchy can be limited by Depth. Composite contents are useful to create several types of documents, such as simple sequences of contents, or complex graphs of contents (webs).

**References**

Contents can contain references to other contents.

**Pseudo-references**

Contents can contain pseudo-references [Aguilar, 1999], which act as "pretended references" implicitly stored in contents. Pseudo-references are de-referenced by a reference system. For example, a pseudo-reference can be used to define a reference between a class in a UML class diagram to the
corresponding API documentation, using a reference-system that accepts the pseudo-reference and calculates the desired reference to the API.

**Notations** Contents are represented in a specific notation. Although different notations can often be used to represent the same contents, for a specific purpose, some can be more effective than others.

For example, to understand a very small computer program, it is irrelevant if the information is represented in text, or in HTML, or even in a bitmap graphics. However, to parse the same program, the graphics representation would require a lot of pre-processing, while the textual representation is ready to be parsed.

Examples of notations useful in minimalist framework documentation include: free text, structured text (e.g. XML), source code, UML, and images.

### 6.2.3 Concrete contents

All the model concepts above presented capture the commonalities and specificities of the many kinds of contents found in the different types of documents typically used in minimalist documentation and framework documentation.

The most elementary unit of documentation of the model was already presented as being the contents. However, concrete kinds of contents were not presented yet, contents such as framework domain, steps of a recipe, pattern's problem definition, hook changes, error message, etc.

There are at least two possibilities to model concrete contents: specializing contents, and using contents as a meta-concept. Both alternatives are represented in Figure 6.9 and discussed below.

![Diagram](image)

**Figure 6.9** Modeling concrete contents.
Concrete kinds of contents can be modeled as specializations of the concept contents. For example, the contents of classes in UML diagrams could be modeled as something like UML class contents which would inherit the properties of the general concept of contents. The same would be required for contents of classes in Java language, in C++ language, in C# language, and so on.

In terms of the final model, this approach would lead to an explosion of the model in other dimensions, such as the contents notation, and the type of document that owns the contents. Although satisfying the requirements for a possible model for minimalist framework documentation, the model would be huge, strongly coupled to many fine-grained details of the contents to be covered, and thus it would be rigid, difficult to evolve, hard to extend, costly to implement, and not usable in practice.

To be easy to use and implement, the model must trade-off usefulness with flexibility and extensibility. Therefore, instead of using a specialization approach to model the many different kinds of concrete contents, it was used a meta-modeling approach, i.e. the model is used as a meta-model with abstract concepts, based on which concrete model concepts can be obtained by instantiation.

Using a meta-modeling approach, the model doesn’t need to know nothing about concrete contents. The specificities of representing concrete contents such as those previously mentioned is left to model implementation. The integration of heterogeneous contents and notations is achieved through conformance to the meta-model.

Meta-modeling approaches are often more complex to design and implement, but the benefits in terms of flexibility and simplicity usually pay off in a short term.

### 6.2.4 Templates

Different types of documents use different kinds of contents. The most relevant types of documents are included in the model as document templates. A template is a kind of “active document” that provides the basic structure of a type of document in the form of a good document sample, called exemplar, eventually accompanied with a formatter for handling its contents.

Templates provide one way of customizing the model. Customization and extensibility capabilities are provided through configuration. The set of templates are contained by the configuration.

These concepts are represented in Figure 6.10 and described below.
Templates help on creating documents of a certain type by providing the basic structure and an exemplar document. Templates can also provide formatters to format the contents of their instances using the style pre-defined by the template's author.

Formatters The contents of documents created based on templates can be formatted using the formatters provided by the template's author, thus leading to consistency of formatting. Formatters are a kind of contents transformer.

Exemplars Exemplars are a good or typical example of an instance of the type of document defined by the template. In addition to the typical basic structure of the respective type document, an exemplar usually contains also instructions on how to fill the structure.

The model includes templates for the following types of documents: minimalist framework manuals, framework overviews, cookbooks, recipes, patterns, pattern instances, examples, and error recovery guide. As the model is extensible, new templates can be added.

6.2.5 Multidimensional structure

All the contents must be organized in an appropriate manner, so that users don't become overwhelmed or lost when using the documentation.

For that purpose, the contents can be virtually organized in a multidimensional structure, also known as cube, or hypercube [Agrawal et al., 1997].

The structure contains definitions of documentation aspects and layers. The dimensions of the structure are defined by aspects. The layers enable the stratification of the structure in hyper plans, also known as views. Contents
can be filtered (or transformed) using filters according to their relevance for a certain documentation aspect and the layer in focus. Similarly to the templates (Section 6.2.4) the structure is contained by the configuration. The distribution of contents along the layers is supported by meta-information, either manually annotated, or automatically synthesized from the contents.

These concepts are represented in Figure 6.11 and described below.

![Diagram](image)

**Figure 6.11** Model concepts for structuring the contents.

**Structure**

The structure contains definitions of aspects and definitions of layers. The structure is used to configure how the contents must be organized in a multidimensional structure.

**Aspects**

Aspects are used in the model to define the (orthogonal) dimensions of the multidimensional structure. Aspects represent a measurable property of contents. Each aspect may have its own measuring scale: continuous (e.g. from 1 to 10), discrete (e.g. high, medium, low), or named (e.g. developers, designers, testers).

Examples of aspects were presented in “Different aspects”, on page 73, and include the following: kind of audience, level of abstraction, granularity, purpose, and notation.

**Layers**

Layers define configurations of values aspects. Layers are used to create custom views over the contents, based on the relevance of specific contents for a specific view, measured using aspect values.

A concrete example is a possible “overview” layer defined to include only contents that are abstract, coarse grained, and appropriate for users (audience).
The contents can be transformed using the filters defined in layers, thus enabling easy definition of different views over the contents, based on the layer concept. Filters are a kind of contents transformer.

Examples of filters applicable to textual contents include the following: "show everything", "hide everything", "first sentence", "first paragraph", "hide references". Complex filters for structured contents can be written using query languages or scripting languages.

The relations between contents, layers, aspects, filters and templates are presented in Figure 6.12 in a generic example of a configuration.

![Figure 6.12 Configuration of layers and filters for an uni-dimensional structure.](image)

The structure of this example organizes the different kinds of contents, from a to z, in n layers, and is unidimensional, based on a single aspect measured in values ranging from low to high.

To simplify the representation, the layers defined divide uniformly the range of possible values for the aspect. It is represented a template T that provides several filters $f(a, n)$, $f(b, 3)$, $f(c, 2)$, and $f(c, 1)$, to transform specific contents a, b, and c, when presented at layers n, 3, 2, and 1, respectively.

Figure 6.13 represents a simple configuration of layers for a two-dimensional documentation structure defined by the aspect level of abstraction, measured with a named scale with values abstract, intermediate, and concrete, and the aspect kind of audience, also measured with a named scale with values selector, user, and developer.
Figure 6.13 Example of a configuration of layers.

This configuration defines three layers, named introduction, design and implementation, which were defined coincident with the values of the level of abstraction, to simplify the representation: one layer for each value. The configuration considers also four kinds of contents, framework overview, design pattern, example and class. This configuration defines that:

- the framework overview is relevant to the introduction layer and all kinds of audience,
- examples are relevant to all layers and all audiences,
- design patterns are relevant for the design and implementation layers, and for users and developers,
- classes are relevant for the implementation layer and developers.

6.2.6 The (meta)model

A global diagram with all the concepts defined is presented in Figure 6.14. This diagram corresponds to the meta-model of the approach. Concrete documents and contents would be an instantiation of the documents and contents concepts.

To cope with a vast diversity of documentation requirements, the implementation of the meta-model should use the flexibility and extensibility provided so that new custom types of documents, contents, notations, transformers, and layers, to mention only a few, can be added with a reasonable effort.
Figure 6.14 The complete documentation (meta)model.
6.2.7 Minimalist framework manual

To illustrate the instantiation of the (meta)model for concrete documents, and also to provide a base for its implementation, the documentation model identifies the contents, structure, and interdependencies of the most relevant types of documents used in minimalist framework documentation. All the documents and contents presented instantiate the abstract concepts of document and contents described in Section 6.2.2.

The model combines all these documents in a coherent and integrated bundle of documentation, which is proposed in this dissertation under the name of minimalist framework manual.

Combination of "best practices"

A minimalist framework manual encompasses a set of different types of documents already used in minimalist documentation and framework documentation.

The minimalist framework manual doesn't intend to propose new types of documents, except for itself, but only to combine minimalist versions of the existing types of framework documents by following the principles of minimalism. In short, the minimalist framework manual intends to add value through an appropriate combination of best practices in framework documentation and in minimalist documentation.

Design principles

The major design principles followed were to produce minimalist versions for the existing types of framework documents and combine them using the minimalist design principles proposed in [van der Meij and Carroll, 1998]:

- "choose an action-oriented approach";
- "anchor the tool in the task domain";
- "support error recognition and recovery";
- "support reading to do, study and locate".

Another design principle followed was to provide different kinds of entry points, ranging from a general framework overview to source code examples. Because the contents of the manual are cross-referenced, considering the associations between contents, the manual supports a good navigability between different types of documents and different kinds of contents. Easy navigability helps on supporting different ways of reading the documentation, and therefore minimizes the obstacles to learning strategies that readers spontaneously adopt [Carroll, 1990].

Contents

The types of framework documents to combine include those that have proved to be effective (Section 5.2.2), namely cookbooks, patterns, and examples, and the types most frequently used to document frameworks
[Yassin and Fayad, 1999], which adds framework overviews to the previous.

Influenced by the minimalist documentation, task-oriented information was enforced by including hot spot descriptions and hook descriptions [Pree, 1999; Froehlich et al., 1997] in addition to the cookbook and recipes.

To support error recognition and recovery, the minimalist framework manual includes an error inventory and an error recovery guide. To be useful, frameworks themselves must provide error messages containing references to information supporting the detection, diagnosis, and correction of the errors.

The basic structure of the resulting manual is presented in Figure 6.15. All contents shown are concrete instances of contents and documents metaconcepts.

![Diagram of minimalist framework manual structure](image)

**Figure 6.15** Contents and structure of a minimalist framework manual.

The manual encompasses the following kinds of documents:

- **a framework overview** provides introductory information useful for new users and framework selectors; the framework overview is defined as the first recipe of the cookbook;

- **a cookbook and recipes** provide task-oriented information on how to use the framework; recipes include references to running examples in source code, and references to hook descriptions;

- **an error inventory** and **an error recovery guide** helps framework users to
understand and solve the errors they encountered; these documents include references to parts of the documentation that are likely to contain information misunderstood by the user;

- **running examples** encourage users on getting started fast using the framework, on training on real tasks, and on learning by self-exploration;

- **hot spot** and **hook descriptions** provide task-oriented information similar to recipes but are more precise and contain more design information, and help quickly locating information that explains how the framework works to support the desired customization; hooks reference the pattern instances used for its implementation;

- **patterns** and **pattern instances** help understand in detail what can be adapted and how the adaptation is supported.

From the types of documents included in the minimalist framework manual, patterns and pattern instances are those that contain more design information. The patterns and pattern instances used in a framework, showing where and how they are instantiated help readers on understanding the internals of the framework design and on locating the corresponding source code implementations, when available. In Figure 6.16 are represented the contents and structure of patterns and pattern instances, showing the references they have directly to the corresponding implementations in source code.

![Diagram](image)

**Figure 6.16** Contents and structure of patterns and pattern instances.

Although the minimalist framework manual emphasizes task-oriented information ("how to use") it includes also sufficient design information ("how it works") to help users understand its design, organized from a usage perspective. The contents of the manual are modular, referenced, and allow...
reading in any order, so it is left to the reader to decide what to read first, what to skip, what is important, and what is irrelevant.

Layers

A possible configuration of layers for the contents of the minimalist framework is represented in Figure 6.17.

The information is stratified in seven layers along a unique aspect. The figure represents the distribution of the contents along one single dimension, the abstraction level. The contents are represented as areas laid out over the layers they are considered to be relevant. References between contents are represented by arrows. Finally, for each audience, framework user, framework developer, and framework maintainer, it is represented the main entry point to the documentation.

Figure 6.17 Configuration of layers for framework documentation.

Overview. Contents intended to communicate the purpose of the framework to potential users, in a clear and concise way: framework overview, and snapshots.

Domain. Contents that define the application domain covered by the framework, namely the products that can be developed with the framework, their variability aspects (hot spots), and how the framework can and should be reused: use cases, scenarios, examples, cookbooks, recipes, and design
patterns.

**Components.** Contents that formally define a black-box view, i.e. the properties and behavior of the products that can be developed with the framework, and how they must interact with custom code: type specifications, operation specifications, state-charts, contracts, and design patterns.

**Design.** Contents that present the design principles of the framework, and describe its micro-architectures and mechanisms of cooperation between components: technical architecture, application architecture, design patterns, design notebooks, and refinements from specification level to design level.

**Public view of the implementation.** Contents representing an external view of the implementation of framework components: detailed use cases, collaborations, roles, interfaces, classes.

**Protected view of the implementation.** Contents representing the view available for developers of components through extension of classes provided by the framework. In addition to public view, this protected view must include also subclass/superclass contracts: detailed use cases, collaborations, roles, interfaces, classes.

**Private view of the implementation.** Contents that present a white-box view over the implementation of the framework, usually in the form of source code.

Despite using only one dimension, this example emphasizes the complexity and difficulty associated with organizing framework documentation, specially when done without a well-defined process and adequate tool support.

The next sections introduce the process and tools defined in the approach that implements the documentation model presented in this section.
6.3 Documentation Process

When documenting a framework, besides knowing what to document, described in the previous section, it is also important to know who, how, and when to document. These are the three main questions typically addressed by processes and methodologies, independently of being processes for documentation, or processes to help perform mechanically and effectively other activities such as, analysis and design, or testing.

Although production processes are not capable of ensuring with total confidence the quality of an activity or a final product, there is however a direct link between process maturity and product quality. Mature processes help on improving productivity, by avoiding errors, and on identifying likely causes of systematic errors [Huang and Tilley, 2003; Visconti and Cookand, 2002].

To be useful, framework documentation must include a lot of contents, gathered from different types of documents, at different moments of the development life cycle, 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.

This section presents the process outlined to guide framework developers on writing minimalist framework documentation of good quality, in an effective and economical way.

This documentation process defines the roles and activities involved in the production of the minimalist framework manuals and defines a set of guidelines to help writers on producing good quality documentation.

6.3.1 Rationale: "agile documentation"

The fundamental goal of a process is to improve quality and to reduce costs, which is often achievable by process formalization, appropriate mechanization of human activities, and automation of repetitive tasks.

While these ideas apply considerably well to many engineering activities, they are not very effective for intensively creative activities, such as software design, programming, and software engineering activities.

Heavyweight processes [Kruchten, 1999] usually use many prescriptive guidelines, too much formality, and a lot of documentation as means to solve the lack of skill, discipline, and understanding, respectively. Despite their problems [Highsmith, 2000], this kind of processes are useful in development environments where the more flexible processes don't fit well,
such as large organizations, large teams, low technical skills, or teams with communication problems.

Agile processes [Agile Alliance, 2001; Beck, 2000; Highsmith, 2000], on the other hand, favor on adopting simple but effective practices that help improve skill and productivity, such as "pair-programming" for example [Williams and Kessler, 2000; Nawrocki and Wojciechowski, 2001], to promote discipline instead of formality, and to improve team communication instead of forcing the production of too much documentation.

This documentation process was designed considering agile processes, which typically are very restrictive on adopting a documentation approach, because documentation is regarded as an expensive activity, which is often true, indeed. To avoid the program understanding problems at maintenance and evolution phases, with "no documentation", agile processes suggests several practices, such as: to write code as simple as possible, to do code refactoring continuously [Opdyke, 1992; Fowler, 1999], and to use collective code ownership, which forces continuous code reviews, thus leading to better and simpler code.

While the extreme practice of "no documentation" can lead to success in small software projects, this is not true for frameworks, because framework users require documentation, which is software documentation, not a simple user manual. For example, JUnit [Beck and Gamma, 1997] a simple and very successful framework for unit testing developed by XP practitioners, concretely by Kent Beck, one of the pioneers of eXtreme Programming, is accompanied with simple but effective documentation.

Recent work [Ambler, 2003; Rueping, 2003] addressing the problems of documentation on agile processes are not so extreme anymore and suggests that "documentation should be lean and mean" [Ambler, 2003].

Therefore, this documentation process was abstracted from concrete documentation processes and designed with a special concern on lightweight processes. The resulting process aims at being simple, flexible, almost neutral, and easy to adapt to different development processes and environments, ranging from literate programming environments [Knuth, 1984; Williams, 1992] to mainstream integrated development environments.

6.3.2 Goals

Defining suitable methods for framework documentation is an important first step to reduce the high cost of providing suitable framework documentation. Although the combination of patterns, hooks, hot spots, examples, and architectural illustrations, has proven to be very effective, a
commonly accepted method for documenting frameworks, would help produce and combine the contents of framework documentation.

This process aims at addressing some of the issues enumerated before in “Process issues”, on page 98. The goals of this process 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 an initial set of such guidelines, so that writers, specially 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 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 goals, to design the documentation, to write, test, and refine, until it is ready for usage.

- **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).

Based on these goals, the roles, techniques, activities, and guidelines for the production of the minimalist framework manuals are defined.

### 6.3.3 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. The roles identified are the following:

- **developers**, such as framework developers, and framework maintainers, which are responsible for content creation mostly during the development phase;

- **technical writers**, which are responsible to structure, guide, review and conclude the documentation;

- **documentation managers**, which are responsible for configuring and maintaining the documentation base, namely the template documents, template instances, and the filtering, transformation and formatting of documents according to the layers configured, etc.
Depending on the writers discipline, documentation managers can enforce or flexibilize the way documentation is produced with the goal of achieving good quality. The more flexible and informal the process, the more attractive it will be for the writers, because formality often compromises creativity. However, too much flexibility may result in inconsistent writing styles and presentation, if the writers are not well disciplined.

6.3.4 Activities

Because the production of framework documentation (Section 3.4.1) 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 of producing framework documentation are:

- **configuration** of the documentation base;
- **creation and cross-referencing** of the various kinds of contents;
- **normalization, integration and storage** of contents;
- **publishing and presentation** of contents to target audiences.

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 (Section 3.4.1 “Typical Activities of Framework Documentation”, page 64).

6.3.5 “Good practices”

In terms of techniques and practices prescribing how-to write minimalist framework documentation, the process is very open, in order to be flexible and adaptable to different kinds of development environments, personal motivation, and needs.

However, the process intends to combine good practices of technical documentation and software documentation considered helpful to achieve cost-effective documentation whose benefits can be greater than the cost of creating and maintaining it. The good practices were selected from the following work, ordered from the fundamental to the specific:

- **the PQT1 set of guidelines** found in [IBM Corporation, 1983; Dean, 2002; Hargis, 1997], previously presented in “Quality assessment”, on
page 93, and discussed in “User documentation”, on page 124;

- the design principles of minimalist instruction presented in [van der Meij and Carroll, 1998] and summarized in Section 4.7;

- the practices of agile processes related with documentation [Agile Alliance, 2001; Ambler, 2002; Rueping, 2003; Ambler, 2003], discussed in “User documentation”, on page 124.

From these references, the Ambler’s work on “Agile Modeling” [Ambler, 2002] is considered very useful to define a documentation process aiming at being adaptable to agile processes, as discussed before in “User documentation”, on page 124, because, in the same vein as this dissertation work, Ambler’s work recognizes the importance of software documentation (in models), even for agile processes, and proposes a set of practices to create cost-effective models using an agile approach.

Although “Agile Modeling” is devoted only to models, models are a form of documentation, and thus processes for cost-effective modeling and processes for cost-effective documentation have in common similar fundamental issues related with cost-effectiveness.

Therefore, the Ambler’s work was used as a starting base, providing a skeleton to define and organize the practices of this documentation process into which the other practices and guidelines above mentioned are combined.

The practices are presented below organized in core practices, and supplementary practices. It is important to mention that these practices assume that the contents of the several documents are maintained in consistency with the support of an appropriate documentation environment. Without such support, the costs of adopting some of these practices would be totally different, e.g. “Create several documents at once.”

Core practices

Collective ownership. By default, all documents are readable and editable by anyone involved in the project.

Collective ownership of documents usually leads to better documents, because everyone can contribute, resulting in richer and more complete documents. The documents can be reviewed later by a technical writer to improve its homogeneity, consistency of terms, writing style and formatting.

Collaborative writing. Write in collaboration with other people, to assess the understandability, completeness, and accuracy of the document.

Create simple documents, but just simple enough. A minimalist document must be
brief, it shouldn’t contain everything, but just the enough information that fulfills its purpose and the intended audience. The simplicity and understandability of contents must be evaluated by the readers.

Create several documents at once. To represent all the aspects of a framework, and to serve all the audiences and purposes, it is necessary to use different documents (e.g. recipe, example, hook description, and pattern), which when edited in parallel help writers on “dumping” their knowledge more effectively, as writers can document almost every aspect they have in mind without switching contexts. Cross-references must be used to link the separated but related documents.

Publish documents publicly. Publicly available documents, published for everyone to see, supports knowledge transfer, and improves communication and understanding. The feedback from readers is improved and the overall quality of documents is quickly improved.

Document and update only when needed. To be cost-effective, documents should be created and iteratively refined only when needed, not when desired.

Reuse documentation. Reuse contents and structure of existing documentation in order to improve the productivity and quality of the documentation. Reusable contents must be modular, closed, and readable in any order.

Use simple tools. The usage of simple tools help focus more on the contents, rather than on the presentation.

Define and follow documentation standards. Writers must agree and follow a common set of documentation conventions and standards on a project.

Document it, to understand it. To document helps on formalizing ideas focused on single aspects, in isolation from many others.

6.3.6 Guidelines

In addition to the practices proposed above, the process provides a list of guidelines to help writers on producing good quality documentation, and a table summarizing the different types of documents, their purpose and typical applicability.

Because the fundamental issue of assessing the quality of technical documentation is well beyond the scope of this dissertation, the guidelines provided are a synthesis of the quality requirements and characteristics proposed in [IBM Corporation, 1983], also available in [Dean, 2002]. These guidelines in conjunction with the practices and the quality template
documents help improve the quality of the documentation produced.

<table>
<thead>
<tr>
<th>Quality dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Easy to Use</strong></td>
<td></td>
</tr>
<tr>
<td>Task orientation</td>
<td>Present information from the reader's point of view</td>
</tr>
<tr>
<td></td>
<td>Indicate a practical reason for information</td>
</tr>
<tr>
<td></td>
<td>Order the presentation to reflect the order of use</td>
</tr>
<tr>
<td></td>
<td>Devise titles and headings to reveal the task</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Provide accurate technical information</td>
</tr>
<tr>
<td></td>
<td>Provide accurate references and other auxiliary information</td>
</tr>
<tr>
<td></td>
<td>Use correct grammar, spelling, and punctuation</td>
</tr>
<tr>
<td></td>
<td>Use sexually neutral terms, unless other terms are appropriate</td>
</tr>
<tr>
<td>Completeness</td>
<td>Cover all the topics that readers need, and only those topics</td>
</tr>
<tr>
<td></td>
<td>Cover each topic in just as much detail as readers need</td>
</tr>
<tr>
<td></td>
<td>Include all standard parts and all promised information</td>
</tr>
<tr>
<td></td>
<td>Repeat information only when readers will benefit from it</td>
</tr>
<tr>
<td><strong>Easy to Understand</strong></td>
<td></td>
</tr>
<tr>
<td>Clarity</td>
<td>Present material so readers can understand it the first time</td>
</tr>
<tr>
<td></td>
<td>Pace the presentation to be neither too fast nor too slow</td>
</tr>
<tr>
<td></td>
<td>Write directly and economically</td>
</tr>
<tr>
<td></td>
<td>Use only technical terms that are necessary and appropriate</td>
</tr>
<tr>
<td></td>
<td>Define each term new to the intended reader</td>
</tr>
<tr>
<td></td>
<td>Provide appropriate examples to communicate effectively</td>
</tr>
<tr>
<td>Visual communication</td>
<td>Attract and motivate your readers with graphic techniques</td>
</tr>
<tr>
<td></td>
<td>Employ visual techniques to communicate effectively</td>
</tr>
<tr>
<td><strong>Easy to Find</strong></td>
<td></td>
</tr>
<tr>
<td>Organization</td>
<td>Reveal how the pieces fit together</td>
</tr>
<tr>
<td></td>
<td>Emphasize main points: subordinate secondary points</td>
</tr>
<tr>
<td></td>
<td>Don't force readers to branch unnecessarily</td>
</tr>
<tr>
<td></td>
<td>Present similar topics in a similar way</td>
</tr>
<tr>
<td>Entry points</td>
<td>In introductory sections, reveal the order of topics to follow</td>
</tr>
<tr>
<td></td>
<td>Stock the index with predictable entries for the topics covered</td>
</tr>
<tr>
<td></td>
<td>Highlight key terms - including new terms being defined</td>
</tr>
<tr>
<td></td>
<td>Rarely run text for half a page without a heading</td>
</tr>
<tr>
<td></td>
<td>Rarely run a paragraph beyond a dozen lines</td>
</tr>
</tbody>
</table>

Figure 6.18 Quality aspects of technical documentation [IBM Corporation, 1983].

The table in Figure 6.19 lists the major types of documents used in minimalist framework documentation 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 typically considered the best for writing it.

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

<table>
<thead>
<tr>
<th>What to write</th>
<th>Why</th>
<th>By who</th>
<th>To whom</th>
<th>When</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework documentation</td>
<td>to give an overview of the framework: domain, customizations, documentation</td>
<td>writer, developer</td>
<td>all audiences</td>
<td>first release</td>
</tr>
<tr>
<td>Cookbooks, recipes</td>
<td>to explain how to use the framework: the points of customization and the steps required to perform</td>
<td>developer, writer</td>
<td>framework users</td>
<td>development</td>
</tr>
<tr>
<td>Examples</td>
<td>to provide concrete examples of usage of the framework</td>
<td>developer</td>
<td>framework users</td>
<td>development</td>
</tr>
<tr>
<td>Hot spot</td>
<td>to provide information about the points of the framework that accept customization</td>
<td>developer</td>
<td>framework users, framework developers, framework maintainers</td>
<td>development</td>
</tr>
<tr>
<td>Book</td>
<td>to provide detailed design information about how to customize the framework in a specific point of flexibility</td>
<td>developer</td>
<td>framework users, framework developers, framework maintainers</td>
<td>development</td>
</tr>
<tr>
<td>Patterns</td>
<td>to provide design information about how the framework works to offer a specific functionality: classes, methods, rely, collaborations</td>
<td>developer, writer</td>
<td>framework users, framework developers, framework maintainers</td>
<td>design</td>
</tr>
<tr>
<td>Pattern instances</td>
<td>to provide detailed design information about how the framework was implemented to offer the feasibility and extendibility</td>
<td>developer</td>
<td>framework users, framework developers, framework maintainers</td>
<td>development</td>
</tr>
<tr>
<td>Minimalist documentation</td>
<td>Error inventory</td>
<td>to provide an organized list of error messages emitted by the framework, pointing to error recovery guidelines, to assist on the identification and detection of errors</td>
<td>developer, maintainer</td>
<td>framework users</td>
</tr>
<tr>
<td>Error recovery guide</td>
<td>to provide an organized list of guidelines to assist on recovering from errors</td>
<td>developer, maintainer, writer</td>
<td>framework users</td>
<td>development, first release, maintenance</td>
</tr>
</tbody>
</table>

Figure 6.19 Typical authors for the different types of documents.

In summary, the process aims at being very simple to follow because it is based on well-known concepts and practices that bundle together good practices of minimalism and agile documentation.

Therefore, when compared with ad-hoc processes commonly used to document software or frameworks, the process is better because it is defined and provides guidance on writing good documentation for frameworks.
6.4 Tools and Utilities — XSDoc

The previous sections presented the fundamental components of the approach, concretely the documentation model, and the documentation process. The model identified the contents required to produce a minimalist framework manual, and how they are related and organized. The process defined a flexible way of producing the intended documentation, by identifying activities and roles, and suggested a set of good practices that aim at guiding writers on how to document a framework in a minimalist way. Despite the good properties demonstrated by these two components, they are however very difficult to use in practice without appropriate tool support, an issue of framework documentation ("Tools issues", on page 100).

In fact, a cost-effective production and maintenance of good quality framework documentation requires integrated software documentation environments easy to use, powerful, widely available, interoperable with the most common development environments, and supporting the major kinds of documents.

The minimalist approach contributes for this issue with XSDoc, an extensible documentation infrastructure, which, through an attractive documentation environment, significantly helps authors and readers on producing and using minimalist framework documentation.

The main goal of XSDoc is to support the minimalist approach by making convenient its usage in a wide range of software development environments. XSDoc implements the conceptual documentation model defined in Section 6.2, and conforms to the process defined in Section 6.3, and therefore acts as a concrete vehicle to deliver to framework documentation authors and readers the powerful structuring features of the model and the guidance provided by the process.

XSDoc was created to overcome the limitations in terms of tool interoperability, integrability of heterogeneous contents, and extensibility (Section 5.2.1) found on existing literate programming systems, and alternative software documentation techniques used to solve the semantic consistency problem of software documentation.
6.4.1 The XSDoc documentation infrastructure

XSDoc is an open and extensible documentation infrastructure based on a Wiki engine [Cunningham, 1999] and XML technology [Bray et al., 1998], which uses a multiple source approach ("Multiple source", on page 91) to ensure the semantic consistency between different kinds of contents, namely source code, models, and documents.

The implementation of XSDoc at the moment of this writing supports only the integration of source code in Java and C++, UML diagrams via graphic files, plain text documents, and XML documents.

XSDoc provides a simple cooperative web-based environment for the creation, integration, publishing and presentation of minimalist framework documentation. XSDoc covers the typical functionality of a content management system specific for the production and usage of framework documentation.

XSDoc aims at closing the gap "development - documentation", to help make documentation convenient and attractive to framework developers, by assisting them on documenting while they code and design.

The XSDoc infrastructure results from the reification of two previous prototypes. The FrameDocMS [Aguiar and David, 2003] was a first prototype developed with the goal of evaluating the appropriateness of using a Wiki [Cunningham, 1999] ("Wiki-based editors", on page 127) and XML technology to implement a content management system for framework documentation.

As the key idea of "Wiki-based software documentation" behind FrameDocMS was considered attractive, it was then generalized to object-oriented software documentation and evaluated within the WID\(^1\) prototype [WID, 2003; edeias, 2003]. XSDoc combines the best features of both prototypes and in addition has an evolved architecture, more flexible and easier to extend.

The XSDoc infrastructure is composed by one Wiki engine, plugins for seamless integration in open IDEs (currently exists one for the Eclipse IDE), and a set of document templates (cookbooks, design patterns, framework overview, etc.), markup languages, and converters of contents to and from XML.

The Figure 6.20 shows the components of XSDoc as well as their interconnections. A first overview of how XSDoc works is now presented.

1. Wiki-based Integrated Documentation
Creation. Documentation contents can be created *internally* with the XSDocWiki or *externally* with editors not included with XSDoc. External contents, e.g., source code contents and UML diagrams, always require the use of external editors. When integrated into an IDE, the edition of all kinds of contents can be made inside the IDE, which interoperates with XSDoc through a IDE-specific plugin.

Integration. Once created, the contents are logically converted to XML and stored in the XSDoc contents repository, which in the present implementation is the file system. The contents are preserved intact as originally created.

Presentation. When requested for presentation in the Wiki, the contents are retrieved from the repository and assembled together on the fly, possibly both external contents and Wiki documents, and converted for the format requested. The way the contents are assembled is defined in Wiki documents, using tags specific to each kind of contents.

### 6.4.2 XSDocWiki

The main component of the XSDoc infrastructure is XSDocWiki, a Wiki engine that extends a typical Wiki engine with several features useful for documenting object-oriented frameworks, and software in general.
The Wiki concept was already presented before in “Wiki-based editors”, on page 127, but because it is a central concept of XSDoc, the concept is revisited here again, now detailing its most important properties for the implementation of XSDoc.

A Wiki is a very attractive tool capable of implementing the powerful concept of editable web pages, still lacking in typical web browsers. A Wiki can be defined as a web platform for the collaborative edition of documents, where everyone can edit any page, using a simple web browser. Using an Edit command available on the page, the reader is allowed to edit the page [Cunningham, 1999]. After saving, the modifications done are uploaded to the Wiki server and are immediately made available online. With a Wiki it is very quick to edit and publish web pages, hence the name WikiWiki, which is the Hawaiian word for “quick”.

A Wiki normally uses a very simple markup language to support text formatting, and a simple mechanism based on Wiki names to automatically link pages. A Wiki name is a name that joins two or more capitalized words, such as AnExample, or WikiName. Despite its simplicity, Wiki names are very powerful because they provide a kind of dynamic-linking mechanism [Bodner et al., 1999; Bodner and Chignell, 1999], where the link is not statically defined, but calculated on the fly, based on the user context (for example), what supports the notion of adaptive web pages. In addition, other kinds of linking can be defined using lexical conventions, such as prefixes, suffixes, and name patterns, in general.

There are a lot of Wiki engines available to use and install [Cunningham, 1999]. Due to the attractiveness of Wiki engines, specially in the software community, sometimes they are used to produce informal software documents (drafts of designs and implementations, design-trade-off discussions, requirements gathering, etc.). Wiki documents are open, evolve incrementally and organically, are easy to edit and organize, promote convergence of contents and consistency of terms, are tolerant, and are easily observable by other users.

Therefore, Wiki editors satisfy most of the requirements previously enumerated in “Tools issues”, on page 100, both to edit and present contents. Because Wiki engines were not originally designed to document software, only a few Wiki engines support source code highlighting features, and none was found supporting synchronization between source code and documents, one of the crucial requirements for an appropriate documentation environment for minimalist framework documentation.

However, typical implementations for a Wiki engine are “lean” and easy to extend. So, instead of improving existing software documentation tools to be attractive and widely available (perhaps impossible to do in a short term), it
was decided to extend a Wiki with the functionality required to do Wiki-based software documentation.

The XSDocWiki was developed using the VeryQuickWiki engine [Cronin and Barnett, 2003] as a starting base, which was then extended with several features to make convenient the edition and visualization of typical software documentation contents, including the support for:

- linking and inlining of source code fragments;
- linking and inlining of UML diagrams;
- instantiation and validation of XML documents;
- accessing version control systems repositories;
- adding new types of documents (templates) using a plugin mechanism;
- a few browsing controls to enable users to adapt the presented contents to their needs.

These extensions enhance the original automatic linking mechanism restricted to Wiki pages to support also linking of source code, models and structured contents, using simple naming conventions, such as prefixes, suffixes, and naming patterns, which are very easy to learn and use.

To be flexible, the XSDocWiki provides a plugin mechanism that supports the addition of new types of documents (e.g. use cases, examples, patterns, cookbook, minimalist framework manual).

A XSDoc plugin for new types of documents typically includes:

- an example of the corresponding type of document, called exemplar (page 160);
- a set of converters to map that type of document to and from XML, if necessary, planned to be accepted in a scripting language (XSL, Javascript, Python, etc.);
- a declaration of the contents (XML elements) possibly containing Wiki text to be parsed for automatic linking of WikiNames;
- and some lexical rules to use during the automatic linking phase.

For example, for Java source files, it is declared that Javadoc comments may contain Wiki text, what enables the usage of Wiki names in Javadoc to link to other source code contents, UML models, or documents.

So configured, the XSDocWiki promotes the collaboration of technical and non-technical people on an incremental edition and revision of framework documents, ensuring high availability of contents (always online), using
simple features, automated archiving, and only requiring a simple web browser, a tool currently very easy to integrate in a vast majority of development environments.

6.4.3 **XML converters and presentation processors**

As most of the contents can be comfortably edited and linked using the Wiki, most of the documentation contents will reside on Wiki pages stored in a file system, a version control system (currently, only CVS is supported), or a database.

However, non-Wiki contents, such as source code programs and UML diagrams need special processing because they must be converted from their original format to XML using XSLT transformations [World Wide Web Consortium, 1999], respectively using JavaML2 [Aguiar et al., 2003], Doxygen [van Heesch, 2002], and SVG/XMI [AlphaWorks, 1999; Ferraiolo et al., 2003] vocabularies.

At a later stage, the contents are filtered and formatted accordingly to be published and presented. At the moment of this writing, XSDoc is able to output HTML files for online browsing, and PDF files for high-quality printing.

6.4.4 **Integration of multiple sources of contents**

The components of XSDoc are closely integrated, in terms of functionality and of the information they exchange.

The integration of Wiki contents with source code, UML models and structured documents is achieved through a multiple source approach, what means that source code and documentation reside in separate files. While this separation preserves source code files and UML files, it requires a way of managing the relationships between their contents.

These relations are supported in XSDoc by small extensions to the original hyperlinking and inlining mechanisms of the Wiki engine, which enable writers to link and inline source code and models using simple textual references. These extensions are implemented using source code parsers, and XML technology. The information exchanged between the tools uses a textual format, both pure text files and XML files. A markup language is also used internally to normalize all the contents in a unique schema, when necessary.

Integration language

In software documentation tools capable of merging and preserving the semantic consistency between heterogeneous kinds of contents, it is often
used an integration language in addition to the languages, or notations, of the contents to integrate. For example, to integrate Java source code with TeX, literate programming systems often use a third language, often a macro language that is used to specify how the Java contents are woven in the TeX code.

The integration language used in XSDoc is the markup language of the Wiki, which is very simple to use and learn. In addition to the hyperlinking mechanisms provided by HTML, XSDoc provides two dynamic mechanisms to integrate and synchronize the possible kinds of document contents (source code, UML diagrams, XML files). The mechanisms are inlining of contents, and linking to contents using Wiki names.

**Inlining**

The inlining of contents is defined with a reference to the specific contents, enclosed within the tags predefined for its respective kind. An example is presented below (Figure 6.21):

```
[<javaSource>]
  junit.framework.TestCase
[<javaSource>]
```

![Image](Figure 6.21) Inlining Java source code in a Wiki document.

Whenever possible, the references use standard formats and rules, such as Javadoc references for Java source code.

As an example, it is shown in Figure 6.22, and in Figure 6.24 (top) a reference to a fragment of Java source code, which extracts from the TestCase class of the junit.framework package the first and last line of the method testCase(String), from JUnit framework.

```
[<javaSource>]
  junit.framework.TestCase#testCase(String);
  comments=no;
  lines=first, last
[<javaSource>]
```

![Image](Figure 6.22) Inlining a method of Java source code in a Wiki document.

**Linking**

The definition of links to specific contents are implemented with Wiki names and external references with predefined formats. Here is an example below in Figure 6.21. Figure 6.24 (top) shows other examples.
The line below will link to the source code of the TestCase class, from package junit.framework.

```java
javaSource:junit.framework.TestCase
```

**Figure 6.23**  Linking Java source code in a Wiki document.

TestCase and CommandPattern wiki names link to topics of the overall documentation. CommandPattern doesn’t have its target defined yet, therefore the Wiki, see Figure 6.24 (bottom) presents it with a question mark (?)..

In a similar way, XSDoc supports the inline and linking of C++ source code and UML models, using the tags cppSource and uml.

---

**Figure 6.24**  Command pattern instance: (top) text written; (bottom) output in XSDoc.

### 6.4.5 Wiki-centric functional integration

The functional integration of the Wiki with the converters and processors is done within the Wiki and its specific extensions, using Java, servlets, Java Server Pages, and external programs, such as: a modified version of the Jikes
compiler [IBM, 2003] to generate JavaML files [Aguiar et al., 2003], and the
doxxygen documentation generator for C++ [van Heesch, 2002].

6.4.6 Integration in development environments

The integration of XSDoc in an industrial development environment is very
easy to achieve in almost every situation, considering that XML is widely
supported everywhere and the Wiki engine only needs a browser to run. The
combined use of XML and Wiki makes this integration successful in almost
every industrial development environment only with the low cost of small
configurations in the environment.

Another goal of the infrastructure is its seamless integration in modern
development environments, such as IDEs.

The integration of the XSDoc infrastructure with IDEs is achieved through
specific plugins. The plugin should enable the use of a web browser through
which the XSDocWiki can be accessed, and should also provide a
communication link between the IDE and the XSDocWiki, to support their
interoperation.

6.4.7 Side-by-side edition of all contents

Much tighter integration of the infrastructure in a development environment
can be achieved with open IDEs, such as Borland's Together [Borland, 2003]
or IBM's Eclipse [Eclipse, 2003], which enable in a single environment the
side-by-side edition and synchronization of all kinds of contents, source
code, UML models, and documents, thus eliminating the need to switch
applications during development.

In Figure 6.25 is represented a snapshot of the Eclipse IDE with the XSDoc
plugin.
6.4.8 Extensibility

XSDoc was designed with a special concern on simplicity, flexibility and extensibility, so that it can be adapted to different project environments, ranging from literate programming environments to industrial integrated development environments.

As a result, in addition to the kinds of contents already supported and the template documents provided, XSDoc users are able to extend it with new templates, linking mechanisms, and formatting features. XSDoc administrators can also customize and develop plugins to support new programming languages or other kinds of contents.

Particularly relevant for the documentation of frameworks is the integration of XSDoc with external reverse engineering tools. In order to automate the creation of reference manuals, for example, reverse engineering tools or other source code tools can automate the generation of documentation, textual documents (Wiki documents) or XML documents, that can be integrated on the fly in the documents visualized.
6.4.9 Using XSDoc

XSDoc is very simple to use, and the basics can be learned very fast in few minutes by people already familiar with the use of a web browser.

As an example, it will be briefly presented how XSDoc can be used to write part of a document for the JUnit testing framework. Figure 6.24 shows part of a document that describes the instantiation of the Command pattern by the class TestCase of the JUnit framework. The text that is required to write is shown in Figure 6.24 (top), and the resulting documentation, is represented in Figure 6.24 (bottom). Any change on external contents, such as the code or models, is automatically reflected in the documentation when the web page is refreshed by the browser, or when the involved contents are modified and saved.

Firstly, XSDoc must be installed and configured. In the current state of development, XSDoc is available as a Tomcat's web application archive (xsdoc.war) ready to be automatically deployed and installed in the application server.

After installation, it is required to create an area on the Wiki to store the documents of the project, usually called a "wiki web". The project's wiki web must then be configured to the specificities of the project at hands. The configuration includes:

- the definition of template documents, based on those already provided with XSDoc, or written from scratch;
- additional linking conventions and navigational properties;
- the location of repositories;
- the kind of integration with the development environment (standalone or a specific supported IDE);
- user accounting.

As mentioned before, documentation contents can be created internally with the XSDocWiki or externally with editors not included with XSDoc.

Using the XSDocWiki it is possible to create new documents, and to revise them using a web-based collaborative environment. The creation of new pages are triggered by following a non-existent topic, presented as a link marked with a question mark, as shown in Figure 6.24.

Template documents can be associated with specific topic name patterns, and are instantiated at topic creation time. For example, the wiki name CommandPattern shown in Figure 6.24 (top) is associated with the template DesignPattern. Depending on the level of integration with the
development environment, the XSDocWiki may trigger the creation of source code and UML diagrams using external editors.

With XSDoc configured and integrated in an IDE, the developer has access to a web browser from where she can use the XSDocWiki. When documenting, the developer creates new pages, writes documents, possibly using predefined templates, uses IDE features such as copy-paste and drag-and-drop, browses project resources, and defines links to other pages or special contents, such as Java source code or UML diagrams, using predefined tags and linking mechanisms.

The contents are always available for online browsing through the XSDocWiki, but can also be exported to static HTML, for offline browsing, or to PDF files, for high-quality prints.

Source code is presented with syntax-highlighting, and smart hyperlinking to other source code files, formal documentation contained in Javadoc comments and Doxygen comments, and other related documents.

6.4.10 XSDoc: plan for the future

XSDoc, an infrastructure combining a Wiki engine, document processing with XML technology, and easy integration in development environments, including open IDEs, shows that the production of framework documentation can be done easier with the help of appropriate tools.

XSDoc combines the simplicity, easiness and versatility of the collaborative environment for document edition provided by Wiki engines with the well-known qualities of XML technology for information processing, taking advantage, although not yet fully at the moment, of the powerful development features of modern open IDEs.

From the work done, we conclude that the use of the XSDoc infrastructure can reduce the effort typically needed to document frameworks, specially when integrated in an open IDE, as it combines the simplicity, easiness and versatility of the collaborative document edition in the Wiki, with the powerful development features of IDEs, and the well-known qualities of XML technology in terms of information integration, processing and presentation.

At the moment of this writing, the most fundamental features were implemented in XSDoc, of which the implementation of the meta-model defined in Section 6.2 is the most important.

The second most important feature implemented is the multiple source mechanism for seamless integration of heterogeneous contents through
XML. The Wiki supports the integration of any kind of contents, on the fly, considering that the information is or can be converted to XML.

The combination of both technologies (Wiki and XML) result a very attractive infrastructure, whose best qualities can be summarized as the following:

- easy to integrate in software development environments;
- easy to use by technical and non-technical people;
- promotes the collaboration of all the team in the documentation;
- improves team communication;
- provides an easy way to access, revise and evolve the documentation;
- and finally, enables a smooth integration of contents in a controlled and structured way, while preserving the information in an universal format, the XML format.

At the moment of this writing, XSDoc satisfies most of the requirements defined, although some not completely, which are the following: B5, B8, E1, E2, E3, E4, E5, E6, E7, C1, C4, C6, D1, D2, D3, D4, D5, D6, D7, A1, A2, A3, A4, EC.

Doing... The most important features not yet fully implemented in XSDoc are the support for the layering mechanisms, and the minimalist browsing controls, but the XML-based representation of the contents is already supported.

The zoom in/out control maps in Wiki to the level of depth of the graph of pages presented at once. By default, the Wiki only presents a single topic, but when zooming out the Wiki should present a graph with depth greater, a feature that it is already implemented but not available from the user interface.

The layering and aspects mechanisms are under development, and these would improve significantly the level of “minimalism” of the resulting documentation.

To Do In future work, the XSDoc tools will be improved with more browsing features to help adapt documentation contents to the user needs (exploration mode, extensive search, query capabilities), new plugins for integration with other popular IDEs will be developed, and other popular Wiki engines will be supported.

An interesting feature to add to XSDoc it would be to support literate programming in XML, i.e. to enable the definition of source code fragments in documents. In order to quantitatively and qualitatively evaluate the impact
of these tools on the quality, understandability and usability of the resulting framework documentation, more user tests and experiments are required, namely in industrial settings.
The minimalist approach proposed in this thesis was developed and validated with the help of two case studies and one experiment that evaluated how the approach works in practice. The case studies were mainly used for validating the approach during its development phase. The experiment was performed to test the resulting documentation.

This chapter presents how the case studies were run and how the experiment was conducted to assess the usefulness of the approach. The main case study used the JUnit framework [Beck and Gamma, 1997] and was conducted by the author. The experiment evaluated part of the documentation produced during the JUnit case study. The development team of the COOLFluiD framework [Quintino, 2003] has started, but not concluded (at the time of this writing), the second case study.
7.1 Case Study - JUnit

The case study on the JUnit testing framework was used mostly to evaluate the documenting approach from a framework developer's perspective. This case study was the most important experimentation method for validating the approach during its development phase and it was also the main source of requirements for the refinement of all its components (model, process and tools).

The case study started after the formulation of the hypothesis of the research that an integrated approach to document frameworks following the principles of minimalist instruction would be beneficial.

7.1.1 Model definition

A model for a possible minimalist manual was quickly defined based on literature searches.

The following phase was to start producing the manual, and immediately became clear that in order to be useful the documentation must have different documents containing similar information, derived from source code, from models and from documents. The redundancy was considered useful from the perspective of the reader, but a serious problem from the perspective of writer, because the redundancy would lead to inconsistency, and inconsistent documentation is not effective.

Another difficulty with the design of minimalist manuals is that they require iteration to be fine tuned to users, what means that to write these manuals without tool support is impractical.

7.1.2 Semantic consistency problem

The first tool evaluated to support the edition of the manual was a literate programming tool, called dotnoweb, developed by an element of the author's research group at INESC Porto [Sousa, 1999].

In terms of consistency between source code and documents, the tool was perfect. The problem found was that it wouldn’t be easy to integrate that tool (and others similar) in a typical development environment, because the source code is generated automatically and the generator is not capable of incremental generation, or reverse engineering from the source code files to literate files, again.

Because the author was accessible, an evaluation was made to judge how to evolve the tool to remove the interoperability issue.
After researching for other alternative tools and not founding anyone able to synchronize code and documents, easy to use, and widely available, it was decided to develop one based on a Wiki engine.

7.1.3 Wiki-based software documentation

The idea of documenting software using a Wiki was considered very interesting, and the first manuals started appearing. Initially the goal was to reproduce the original documentation of JUnit with the tool (XSDoc) under development.

The tool was evolved iteratively and reached the point where it is now, as described in Section 6.4.

7.1.4 Main results

In addition to XSDoc, the other tangible results are the documentation produced as an example of minimalist framework documentation for JUnit.

Because the manual is web-based and looks very simple when printed, they are available online in [Aguiar, 2003].

7.2 Case Study – COOLFluiD

The case study on the COOLFluiD was initiated by the developers of the framework with the intent of externally confirming the results of the previous case study, again from a framework developer’s perspective.

Although this case study has not been planned to make part of the thesis work, the interest of the COOLFluiD development team on using the approach motivated the consideration of their specific requirements during the last iterations of the development.

The COOLFluid framework aims at providing Von Karman Institute for Fluid Dynamics [VKI, 2003] with a reusable software package that supports the development of cool fluid dynamics (CFD) tools. COOLFluiD is currently capable of generally treating multi-physics of CFD simulations, and includes a variable transformation facility which allows the implementation numerical methods with variable set independency. A first usable release is expected at the end of October 2003. Multiple physical models are present: linear and non-linear scalar and system of scalar equations, and an euler system solvable in several variable sets. In addition, a whole configuration facility allows for run-time loading of third party software plugins that allow
VKI partners or users in general to extend or override the functionality already present in the kernel, by having only access to the COOLFluiD application interfaces, not its actual implementation.

The main goal of the COOLFluiD team was to use the minimalist approach to produce the documentation for their latest release. From the point of view of this work, the goal was to use the COOLFluiD team critics as a new source of requirements for the approach under development.

At the moment of this writing, the case study is not considered to be concluded. However, the documentation of the COOLFluiD framework is planned to be released by the framework team in [Quintino, 2003].

As a result, this case study served mainly to validate the model and process already defined with the new requirements of the COOLFluiD framework. Perhaps the most relevant result of this case study was the evolution of the the XSDoc tools to support the integration of source code written in the programming language used to implement the COOLFluiD framework, the C++ language.

### 7.3 Controlled Experiment with JUnit

The objective of conducting this experiment was to test whether the minimalist framework documentation produced during the JUnit case study helps programmers to learn how to use JUnit in faster way, when compared with the traditional documentation of JUnit. It was not an experiment's goal to confirm the claimed advantages of minimalist documentation in general, but only to evaluate the specific benefits of traditional framework documentation redesigned in a minimalist way using the approach proposed by this dissertation.

The results of the experiment support the hypothesis that the minimalist framework documentation helps first-time users of JUnit to be more effective on the writing of simple unit tests. Although the experiment has produced the expected results, the data is still too scarce for unquestionable conclusions.

#### 7.3.1 Designing the experiment

In an experiment done using the scientific method, we are often interested in the effect that a method, a tool, or an artifact, called a factor, has on an attribute of interest. An experiment with an effect previously assigned to a factor is called a treatment. The elements we study are called subjects.
The experiment consisted on staffing two groups of programmers (subjects) to perform a similar set of tasks (treatments) using two different sets of documents for the JUnit framework (factors). The goal was to evaluate the effect of the factors on the time needed to complete the tasks (attribute of interest).

Although the experiment was planned to be run only once in the time frame of this thesis work, it was designed to be easy to replicate in different settings, with different subjects, and by different researchers, to help duplicate the results and thereby to increase its level of confidence. The envisioned experiment has the requirements of both a replicated experiment and a synthetic environment experiment [Zelkowitz and Wallace, 1998].

As in a replicated experiment, several subjects were staffed to perform a task in multiple ways (two ways). The control variables were the duration of the experiment, the subjects experience, and the methods used. The cost of this form of experimentation using human subjects usually limits its usefulness. Programmers are difficult to staff and, in most of the experiments, they can participate only in one single replication, because their interaction with the artifacts under study often irreversibly influences them and affect their behavior in later replications.

Because replicated experiments are usually large and very expensive, it is usual preferred to perform software engineering replications in a small artificial setting that only approximates the environment of real projects. These are called synthetic environment experiments. This kind of experiments is good for investigating small and well-defined aspects of interest. All variables are fixed except those of the control method being modified. Instead of weeks or months, these experiments usually take only a few hours. Such experiments often consist on several subjects, randomized from a homogeneous pool, working at some task for the duration of the experiment while researchers monitor as many variables of interest as possible. Although easier to conduct and potentially leading to statistical validity, these experiments don’t relate very well to problems actually found in real industrial settings, originating threats to its validity that reduce their value.

Having in consideration these requirements, the subjects, factors, tasks and attributes that characterize this experiment were defined.

For controllability reasons, the number of programmers to staff for a replication must average 30–40. The subjects must be divided in two homogeneous groups: the control group, and the experiment group. In addition, the subjects must verify the conditions of a potential “first-time user of JUnit”, which are the following:
to have previous programming experience with Java;

- do not have previous experience with JUnit;

- and to be motivated to learn how to use JUnit.

**Tasks**

All subjects were asked to perform a common set of tasks of increasing complexity involving the use of the JUnit framework. The tasks range from a simple download and setup task to more complex tasks requiring the extension of the JUnit framework. The tasks can be categorized in setup (tasks 1, 2, 3), first-usage (tasks 4, 5), free-usage (task 6), and extension tasks (task 7). These categories reflect the typical activities of reusing a previously selected framework. Because setup tasks were crucial for the rest of the experiment, subjects were allowed to ask for help to complete these tasks. Setup tasks were included in the experiment with the goal of normalizing the programming environment of each subject and to avoid the installation of JUnit before the experiment.

After a brief explanation of the experiment and receiving the descriptions of the tasks (Figure 7.1), the subjects had 2.5 hours to complete the maximum number of tasks. The overall experiment is expected to fit well in a 3 hours session.

**Task 1: Download.** In this initial task the subjects are asked to download the JUnit installation archive from the JUnit web site and to unpack it to a folder of their choice.

**Task 2: Classpath.** After download, subjects are asked to add the path of JUnit classes to the CLASSPATH environment variable.

**Task 3: Run.** This task consists on running JUnit for the first time and allows subjects to verify if it is correctly installed. To complete the task, the subjects need to read the documentation, to learn how to run JUnit, and verify it.

**Task 4: First test.** This task consists on writing a simple class with one single unit test containing several kinds of assertions. Subjects are supposed to browse the documentation to learn how to write such class and the kind of assertions provided by JUnit. This is one of the most important tasks of the experiment.

**Task 5: First suite of tests.** This task consists on writing a second class with unit tests and to group both classes in a suite of tests. Once again, for completing this task, subjects are supposed to learn how to do the task from the documentation provided. This is another important task of the experiment.
### "Unit tests in Java"

**A mini-tutorial**

<table>
<thead>
<tr>
<th>Task</th>
<th>Task description</th>
<th>Annotations</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Classpath</td>
<td>Open a command shell and change the CLASSPATH variable to include the path of the chosen folder for JUnit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Run JUnit</td>
<td>Check is JUnit is running.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. First test</td>
<td>Write a first test class including assertions for equality of values, boolean values and nullity of objects.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. First Suite</td>
<td>Write a second test class and group both in a suite.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. IntegersRange</td>
<td>Write a group of 5 tests for the class IntegersRange, for each of the following test situations: the range is only a simple value; the range has a lower limit and an upper limit; and the range has a series of two sub-ranges of values.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. TestRunner</td>
<td>Write a TestRunner subclass that changes the output messages according to your preferences.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7.1** Form of the experiment describing the tasks.

**Task 6: Free testing of IntegersRange.** After having learned how to create unit tests and to group them in suites, subjects are asked to freely define and run tests for a given class (IntegersRange). This task implies to understand the class provided, define, write and run tests to detect possible problems in the given implementation.

**Task 7: Extending TestRunner.** This task consists on extending JUnit with a new class to run unit tests. Based on the original TestRunner of JUnit, a new test runner must be implemented in order to change its output messages. Subjects are supposed to find out the simplest and fastest way to implement...
the extension. This task requires a detailed understanding of TestRunner: how to use it, its internal design, and its implementation. This is the most complex task of the experiment.

Factors

The experiment consists on observing its effect in terms of productivity.

The control variable of the experiment is the type of documentation provided to the subjects. Each group of subjects was exposed to a different treatment. The treatments consist on performing the set of tasks and differ on the set of documents provided. All documents were published in the same web of XSDoc [Aguirar, 2003].

Traditional documentation. The control group was provided with a copy of the traditional framework documentation available at [Beck and Gamma, 1997]. The initial topics in this set of documents are the following:

- “Getting Started” is a document that points to JUnit web resources;
- “Test Infected” is an article that motivates programmers to write tests for their code using JUnit;
- “Frequently Asked Questions” is a document (FAQ) that contains the answers to the most frequent questions;
- “Cookbook” is a document that shows step-by-step how to write and organize tests using JUnit;
- “Cookstour” is a document that presents the framework design;
- and the Javadoc reference documentation of the API.

Minimalist documentation. In addition to the traditional documentation, the other group was provided with part of the minimalist documentation produced during the JUnit case study. Although the contents of this set of minimalist documents are exactly the same as those of the traditional documentation, they follow the design principles of minimalist instruction, i.e. the contents are organized in an user-centric and goal-oriented way. The initial topics are the following:

- “Overview of JUnit” briefly introduces the purpose and domain of JUnit and points to more detailed information.
- “How to use JUnit?” points directly to concrete user-tasks ranging from simple getting started information to more sophisticated usage, such as organizing tests in suites.
- “How to extend JUnit?” enumerates the extensions supported by JUnit;
• the Javadoc reference documentation of the API;

• and "Source Code Examples" points to the source code of several examples provided with JUnit.

The goal of the experiment was to observe the effect of the documentation on the programmers' effectiveness performing the tasks. The programmer's effectiveness can be defined in different ways, considering two extreme measures:

• **Speed**, emphasizing how fast subjects successfully completed her tasks at first time.

• **Amount of knowledge**, emphasizing on how much knowledge relevant for the tasks at hands the programmer was able to retain; perhaps it was slower the first time he performs a new task, but probably will be more autonomous on later occasions.

The strength of the minimalist documentation is that at either extreme there is no conflict. If the documentation is too difficult to understand, the reader neither succeeds at the task nor learns anything, and if the instructions are too detailed and complete, then readers may loose more time and make more errors interpreting them than they gain in extra information [Draper, 1998].

By consequence, there are at least three attributes of interest to observe during the experiment: task completion speed, level of knowledge retention and number of errors.

**Task completion speed.** Task completion speed (S) is a quantitative attribute that is objective and precise. As tasks are equal for all programmers, the task speed can be easily calculated as the inverse of the time required to complete the task (T), i.e. $S = 1/T$.

**Level of knowledge.** The level of knowledge (K) is a much more difficult attribute to measure because it is qualitative and can only be quantified through its indirect effect on later and similar occasions.

**Number of errors.** The number of errors (E) is another quantitative attribute of interest to observe. It is easy to measure but difficult to interpret correctly. Different programmers use different approaches to write programs: some spend more time designing a program and finally write it with few errors, while others start immediately with a simple program and evolve it until no more refinements are needed. As both approaches have advantages, the number of errors made by a programmer until the task is completed is not a good measure for programming effectiveness. To do more errors till completing a task is not necessarily a measure of less efficiency.
With the goal of keeping the experiment simple to run and considering the problems associated with measuring the level of knowledge and the number of errors, the only attribute of interest that was decided to measure was the duration of each task, i.e. the time in minutes between starting a task and having its completion accepted by the researcher monitoring the experiment. A task was considered to be complete only after the results being verified and accepted as sufficient and correct.

The task duration is an objective measure, but, depending on the task, it can reflect other things than the programmer effectiveness.

- **Duration of setup tasks (1,2,3).** The completion time of setup tasks reflects the experience of the programmer on downloading and unpacking archives, and configuring Java software.

- **Duration of first usage tasks (4,5).** The duration of these tasks reflects the effectiveness of the set of documents on providing the knowledge that programmers need to start successfully using JUnit.

- **Duration of the free usage and extension tasks (6,7).** In addition to the effectiveness of the documentation, the duration of these tasks also reflects the skills of the subject on understanding Java source code, namely the code provided for the IntegersRange class and the classes related with TestRunner.

Although the task duration was the only attribute measured, the level of knowledge acquired during task 4 can be considered as being partially reflected in the duration of task 5, because both tasks have a common sub-task, which is the writing of an unit test. It is assumed that programmers after learning how to write its first unit test, they can write the second one much faster.

Several threats to internal and external validity of the experiment were identified and addressed to minimize their effects.

**Internal validity.** The variable of control is only one: the kind of documentation. But the causal effect of independent variables in dependent variables may be threatened by problems on controlling other external variables. Some of the relevant external variables of this experiment, namely the duration, environment, and materials, are guaranteed to have equal values derived from the way the experiment is setup. However, the subjects’ programming experience and motivation are equalized between the groups by a random group assignment, a procedure which may produce unbalanced groups.

To address the possible problem of unbalanced subjects’ programming experience, the experiment includes tasks that help measure this variable, and thus enable its consideration in the analysis of the results, if desired.
The motivation problem is more difficult to control. This was addressed by offering the experiment as a tutorial for which they are free to register, according to their motivation to learn JUnit. Therefore, we assume that every registered subject has similar levels of motivation, although not necessarily equal. Anyway, low motivation may lead to subjects giving up before the end of the experiment, so the rule defined to follow was to ignore such cases to avoid biasing the results.

**External validity.** The generalization of the results of this experiment is limited by the differences between this experimental situation and other real situations, namely industrial situations, where we can found well organized teams of professionals cooperating to solve problems in much bigger and complex projects.

To address these problems, the experiment was directed to a very specific situation, the first-usage of a small framework, a situation that don’t vary significantly from an experimental to a real situation, because: the level of previous knowledge on the framework is null (first-usage) and the complexity of the framework is limited by its size (small framework).

The difference related with working in team (and learning in team) is minimized in the experiment by allowing the subjects to exchange ideas with other subjects of the same group sitting side-by-side.

Anyway, the design of the experiment is rather conservative. The experiment compromises to take advantage of all the claimed benefits of minimalist documentation with the test of a very clean question in a very typical, although simple, learning situation.

### 7.3.2 Selecting the subjects

Once designed the experiment, the next phase consisted on staffing subjects verifying the requirements enumerated before.

Considering that the subjects should be motivated to learn JUnit, the experiment was offered in two organizations as a short hands-on tutorial. As a result, two sets of subjects were staffed.

The first set of candidates were professional programmers from PT Inovação (PTI), a Portugal Telecom’s company for the development of innovative value-added services, products and applications [PT Inovação, 2003].

The PTI management was previously informed about the experience and the offer was accepted. A set of Java programmers were then invited to participate in the event and 16 have registered.
Although a set of subjects from industry was very appealing, the analysis of their profiles indicated that they were very heterogeneous in terms of experience with the Java language. Some programmers were familiar with the language, but had only small experience in writing Java programs. In addition, other programmers had previous experience with JUnit, thus violating important prerequisites of the experiment.

The second set of candidates were undergraduate students from the Informatics and Computing Engineering (LEIC) programme at FEUP [FEUP, 2003]. The LEIC's students were homogeneous in terms of previous experience with Java programming and many of them didn't knew JUnit before the experiment.

At FEUP, about 80 students of the fourth year of LEIC programme were collectively invited by e-mail to attend the mini-tutorial, having registered 35 of them.

Because the PTI group of programmers didn't satisfy all the prerequisites they couldn't be used to run the experiment. Instead, they were used only to test the experiment. Although a second choice, the LEIC students were selected to run the experiment.

In both situations, the subjects didn't knew that they will make part of an experiment, in order to avoid influencing them beforehand. But they were informed that data would be collected during the tutorial for later statistical analysis.

### 7.3.3 Testing the experiment

To take advantage of the 16 PTI programmers being ready to participate but not entirely satisfying the requirements, the experiment was ran at PTI only for testing purposes. The test aimed at evaluating the understandability of the task descriptions, the adequacy of the tasks for the time defined, to validate the XSDoc wiki engine to publish and serve the documentation, and to get a first impression about the effects and relative acceptance of both sets of documentation.

Perhaps due to the heterogeneity of the group, we observed a great discrepancy in completion times. Some programmers completed the tasks very fast while others needed assistance to complete them in time. However, globally it was confirmed that the set of tasks proposed in the experiment fits very well in the time frame defined for the event (2.5 hours) and that the tasks motivated the participants to learn JUnit by exploring the documentation provided, only exposing them to small but motivating obstacles.
In this test, all subjects were provided with both sets of documents, allowing them to freely use the set they found preferable for their needs. It was observed that the minimalist documentation was generally preferred specially as a starting point for exploring the documentation in the beginning of each task, and also to navigate into and inside source code examples.

7.3.4 Running the experiment

The experiment was performed by 25 of the registered subjects in a room with almost the same capacity as the expected number of participants.

The room was laid out in two parts, one for each group. To randomly divide the participants in the control and experiment groups, each one was free to choose any seat available. Following this method, the subjects distributed themselves in two groups of similar sizes. The group assigned to the minimalist documentation had 13 subjects and the other group had 12.

After providing the descriptions of the tasks to realize and briefly outlining the goals and procedure of the tutorial, the subjects were left unattended, learning and performing the tasks by themselves. The experiment timer started.

Exceptionally, due to the importance of setup tasks for the overall experiment, it was provided help to those subjects that demonstrated being unable to complete them by their own in a reasonable time. Although the duration of setup tasks was measured, it was intended to be used only as an indicator to evaluate subject skills with Java.

During the rest of the tasks, no help was provided at all. Each time a subject completed a task, the results produced were verified for acceptance and the respective completion time was registered. This process was repeated along the 2.5 hours of the experiment. Subjects were authorized to skip tasks and to give up at anytime.

7.3.5 Processing the results

At the end of the experiment, the results were tabulated, adding the times for each task and each subject, and then processed. Before automatically processing the results, some tasks times required normalization, as follows:

- **Setup tasks.** Subjects that received help to complete setup tasks had their respective times not considered individually, task by task. Instead it was calculated the sum of all setup tasks times and then increased according to the level of help given, in order to penalize the help provided.
• **First-usage tasks.** No normalization was required.

• **Free-usage task.** Three subjects from the control group didn’t completed this task, so their times were set equal to the maximum value registered in the experiment.

• **Extension task.** Few subjects have tried to do this task, due to its higher difficulty. Because only one subject have succeeded in the time frame of the experiment, these times were not processed.

The complete table of the results is presented in Figure 7.2. Blank table cells correspond to times not considered for the above mentioned reasons.

<table>
<thead>
<tr>
<th>Minimalist group</th>
<th>1. download</th>
<th>2. classpath</th>
<th>3. first run</th>
<th>4. first test</th>
<th>5. first suite</th>
<th>6. free usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>start</td>
<td>end</td>
<td>total</td>
<td>start</td>
<td>end</td>
<td>total</td>
</tr>
<tr>
<td>A</td>
<td>10:23</td>
<td>10:26</td>
<td>0:03</td>
<td>10:29</td>
<td>10:38</td>
<td>0:09</td>
</tr>
<tr>
<td>B</td>
<td>10:27</td>
<td>10:28</td>
<td>0:01</td>
<td>10:39</td>
<td>10:47</td>
<td>0:08</td>
</tr>
<tr>
<td>C</td>
<td>10:26</td>
<td>10:27</td>
<td>0:01</td>
<td>10:41</td>
<td>10:45</td>
<td>0:04</td>
</tr>
<tr>
<td>D</td>
<td>10:27</td>
<td>10:28</td>
<td>0:01</td>
<td>10:43</td>
<td>10:47</td>
<td>0:08</td>
</tr>
<tr>
<td>E</td>
<td>10:27</td>
<td>10:28</td>
<td>0:01</td>
<td>10:49</td>
<td>10:59</td>
<td>0:25</td>
</tr>
<tr>
<td>F</td>
<td>10:27</td>
<td>10:28</td>
<td>0:01</td>
<td>10:49</td>
<td>10:51</td>
<td>0:25</td>
</tr>
<tr>
<td>G</td>
<td>10:26</td>
<td>10:27</td>
<td>0:01</td>
<td>10:41</td>
<td>10:45</td>
<td>0:04</td>
</tr>
<tr>
<td>H</td>
<td>10:25</td>
<td>10:26</td>
<td>0:01</td>
<td>10:47</td>
<td>10:50</td>
<td>0:25</td>
</tr>
<tr>
<td>I</td>
<td>10:28</td>
<td>10:30</td>
<td>0:02</td>
<td>10:35</td>
<td>10:40</td>
<td>0:05</td>
</tr>
<tr>
<td>J</td>
<td>10:20</td>
<td>10:21</td>
<td>0:01</td>
<td>10:48</td>
<td>10:50</td>
<td>0:07</td>
</tr>
<tr>
<td>K</td>
<td>10:25</td>
<td>10:26</td>
<td>0:01</td>
<td>10:47</td>
<td>10:50</td>
<td>0:25</td>
</tr>
<tr>
<td>L</td>
<td>10:27</td>
<td>10:28</td>
<td>0:01</td>
<td>10:42</td>
<td>10:50</td>
<td>0:28</td>
</tr>
<tr>
<td>M</td>
<td>10:23</td>
<td>10:30</td>
<td>0:07</td>
<td>10:47</td>
<td>10:50</td>
<td>0:25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control group</th>
<th>1. download</th>
<th>2. classpath</th>
<th>3. first run</th>
<th>4. first test</th>
<th>5. first suite</th>
<th>6. free usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>start</td>
<td>end</td>
<td>total</td>
<td>start</td>
<td>end</td>
<td>total</td>
</tr>
<tr>
<td>N</td>
<td>10:25</td>
<td>10:26</td>
<td>0:01</td>
<td>10:29</td>
<td>10:33</td>
<td>0:08</td>
</tr>
<tr>
<td>O</td>
<td>10:29</td>
<td>10:30</td>
<td>0:04</td>
<td>10:45</td>
<td>10:47</td>
<td>0:08</td>
</tr>
<tr>
<td>P</td>
<td>10:27</td>
<td>10:29</td>
<td>0:02</td>
<td>10:38</td>
<td>10:40</td>
<td>0:03</td>
</tr>
<tr>
<td>Q</td>
<td>10:26</td>
<td>10:27</td>
<td>0:01</td>
<td>10:50</td>
<td>10:50</td>
<td>0:05</td>
</tr>
<tr>
<td>R</td>
<td>10:25</td>
<td>10:26</td>
<td>0:01</td>
<td>10:44</td>
<td>10:44</td>
<td>0:05</td>
</tr>
<tr>
<td>S</td>
<td>10:25</td>
<td>10:26</td>
<td>0:01</td>
<td>10:54</td>
<td>10:54</td>
<td>0:05</td>
</tr>
<tr>
<td>T</td>
<td>10:29</td>
<td>10:30</td>
<td>0:02</td>
<td>10:45</td>
<td>10:45</td>
<td>0:05</td>
</tr>
<tr>
<td>U</td>
<td>10:28</td>
<td>10:29</td>
<td>0:01</td>
<td>10:54</td>
<td>10:55</td>
<td>0:05</td>
</tr>
<tr>
<td>V</td>
<td>10:28</td>
<td>10:30</td>
<td>0:02</td>
<td>10:50</td>
<td>10:51</td>
<td>0:05</td>
</tr>
<tr>
<td>W</td>
<td>10:28</td>
<td>10:30</td>
<td>0:02</td>
<td>10:49</td>
<td>10:51</td>
<td>0:05</td>
</tr>
<tr>
<td>X</td>
<td>10:25</td>
<td>10:30</td>
<td>0:05</td>
<td>10:32</td>
<td>10:33</td>
<td>0:02</td>
</tr>
</tbody>
</table>

**Figure 7.2** Task completion times registered during the experiment.

To simplify the analysis of the results and highlight the effect of the documentation on the subjects speed completing the tasks, the times were aggregated both by task category and subjects group. The resulting table of values is presented in Figure 7.3.
Figure 7.3  Times aggregated by task category and subjects group.

Figure 7.4  Completion times for setup tasks.

In a first look at the results, we can observe that the average time of the setup tasks is smaller in the control group (Figure 7.4). As times for setup tasks reflect the subject skills on configuring and using software Java, this possibly means that the subjects of the control group were in average slightly better than the others, in relation to this aspect.

Figure 7.5  Completion times for first-usage tasks.

On the other hand, we can observe that the average times of all other tasks are smaller in the experimental group (Figure 7.5). This is a very good evidence that the minimalist documentation contributed for a reduction in
the time required by subjects to complete their tasks. In other words, programmers of the experimental group were more effective than the others, specially in first-usage tasks, where the reduction in time was of 38%, against 13% in the free-usage tasks, as shown in Figure 7.6.

<table>
<thead>
<tr>
<th>Group</th>
<th>4/5. first-usage</th>
<th>6. free-usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>max</td>
</tr>
<tr>
<td>Control</td>
<td>0:42</td>
<td>1:20</td>
</tr>
<tr>
<td>Minimalist</td>
<td>0:19</td>
<td>0:50</td>
</tr>
</tbody>
</table>

| % reduction | 38%   | 13%   |

Figure 7.6  Percentile reductions of the task completion times of the experimental group.

7.3.6 Discussion

In summary, we find that the results support the hypothesis that the minimalist documentation produced using the approach proposed helps programmers be more effective on learning how to use JUnit for the first time.

The experimental group was faster than the control group, despite the evidence of the latter being more skilled than the former ones, as indicated by the lower completion times of the control group for the setup tasks. The average size of the effect is considerable (~38%) for first-time tasks, and lower (~13%) for free-usage tasks, although this may not generalize to other cases.

As the design of this experiment was conservative, not taking advantage of all the claimed qualities of minimalist documentation, we expect that these results can be reproduced in real situations and perhaps even more pronounced using full-featured minimalist framework manuals.

Because the resulting data of this experiment is too scarce for unquestionable conclusions, more replications of this experiment or similar kind of experiments must be performed using other settings to increase the number of subjects and thus the statistical validity of these results.
This chapter validates the thesis presented in this dissertation. The thesis was divided in two sets of sub-theses, each of which pairs a category of activities and an issue.

The first set addresses issues of producing framework documentation, namely the economic, contents consistency, contents organization, and other quality issues. The second set addresses issues of using framework documentation, concretely the understandability, searchability, and effectiveness issues. For each sub-thesis arguments are made based on the properties of the approach, on results of the case studies, the controlled experiment, and lessons learned from them.

The overall validation of the thesis is achieved through validation of all its sub-theses.
8.1 Thesis review

The final version of the thesis statement, as defined in Section 5.5.6, is reproduced below:

- The minimalist approach to framework documentation makes the typical activities of producing and using documentation easier to carry out respectively for documentation authors (and managers), and documentation readers, when compared to traditional documentation approaches based on framework overviews, examples and reference manuals. To provide such improvements, the minimalist approach combines several enhancements over traditional approaches that collectively help reduce the severity of many production and usage issues of those presented in Figure 5.6.

The first important aspect of this thesis is that its validation must be based on a comparison between traditional documentation approaches and the minimalist approach to framework documentation as defined in this dissertation, hereafter shortly referred as "the minimalist approach".

Traditional documentation approaches set the baseline for the comparison and should represent the state-of-the-art, or the most common practice. Such baseline must be defined for both dimensions of the documentation problem: production of the documentation, and usage of the documentation.

8.1.1 Baseline for comparing approaches

In terms of production, it was considered the best combination of techniques and tools for documenting software, reviewed in Section 3.4 (page 74 to page 87), and discussed in Section 5.2, which include: traditional literate programming systems, such as noweb [Ramsey, 1994], their multiple source alternatives, such as D OgMA [Sametinger and Pomberger, 1992] or elucidators [Nørmark et al., 2000; Nørmark, 2000], and single source alternatives, such as Javadoc [Sun Microsystems, 2003], Doxygen [van Heesch, 2002] or Doc++ [Wunderling and Zockler, 2002].

8.1.2 Baseline for comparing documentation

In terms of usage, considering the pragmatic kind of the contributions aimed to achieve with this dissertation, and also the lack of studies on defining the state-of-the-art on integrated documentation approaches for frameworks, it was decided to use the common practice as the baseline for the comparison. In other words, it was preferred to compare the minimalist approach with the best practices on framework documentation, as used in concrete and
industrial frameworks, instead of comparing it with a possible (because it was not found) approach for which there is no empirical or practical evidence of being the best documentation approach.

With the purpose of defining the “most common practice”, the author used the results of a survey of 21 application frameworks including well-known products such as ACE, CLOS, IBM San Francisco, MacApp, to mention just a few [Yassin and Fayad, 1999]. Although significant and relevant, the study can’t be considered the final word on the framework aspects it covers, but only as the most recent and complete study over the common practice of the still immature field of object-oriented frameworks. Better and future studies would set the baseline of the comparison in different points, so the validation should be easy to update according to that new baseline, when they come available.

The study observed that the two most common kinds of documentation delivered with application frameworks include a framework overview (95 percent) and examples (no percentage is given).

To be conservative on the comparison with the minimalist approach, it was also added a reference manual, because it is also common in many frameworks, and usually is not expensive to produce it, at least the application programming interface documentation (API).

### 8.1.3 Thesis decomposition

The thesis decomposition in sub-theses, earlier presented in Section 5.6, is made along two dimensions: **activities**, and **issues**.

The first dimension is related with the activities of framework documentation, grouped in a category for production activities, and another category for usage activities, as defined in Section 5.5.2 “What is the scope of “production” and “usage”?”, page 139.

The second dimension is related with the issues that complicate framework documentation described in Section 3.4.11 “Key Issues of Documenting Frameworks”, page 92.

By decomposing in this way, each sub-thesis pairs a category of activities with an issue, and states that “the minimalist approach reduces the severity of the issue when carrying out activities from category A”. This means that for each sub-thesis, it must be shown that, when compared to traditional approaches, the minimalist approach makes easier (or at least doesn’t complicate) carrying out the activities defined by the **activities** dimension, with respect to the issue defined by the **issues** dimension.

The validation of the dissertation thesis so decomposed in sub-theses is
viewed as the conjunction of all the individual sub-validations.

8.1.4 Organizing the sub-theses for validation

This decomposition can be represented as a two-dimensional table, where the cells represent the sub-theses to validate.

The tabular form of presenting the sub-theses suggests to assess and validate the sub-theses individually, issue by issue. However, to avoid repetition of arguments, a better strategy is to assess and validate multiple sub-theses in conjunction, for example, organized by the component of the minimalist approach (model, process, or tools) with higher impact on the validation.

Figure 8.1 shows how the sub-theses are organized.

<table>
<thead>
<tr>
<th>Category</th>
<th>Issue</th>
<th>S</th>
<th>Production</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>General documentation issues</td>
<td></td>
<td>~</td>
<td>m p t</td>
<td>m p t</td>
</tr>
<tr>
<td>Technical</td>
<td>Quality assessment</td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Semantic consistency</td>
<td></td>
<td>~</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Appropriate documentation environments</td>
<td></td>
<td>~</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Knowledge acquisition</td>
<td></td>
<td>~</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Programmer's attitude</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Framework-specific documentation issues</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Product</td>
<td>Combining prescriptive and descriptive information</td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Managing consistency and redundancy</td>
<td></td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Good organization of contents</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Graded presentation of contents</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Several entry points and traversal paths</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Multiple views and outputs</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contents integration</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Lack of standards</td>
<td>~</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Guidelines</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cost</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>User-centered process</td>
<td>~</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Documentation reuse</td>
<td>~</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Tools</td>
<td>Authoring tools</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contents synchronization</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Extraction tools</td>
<td>~</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Browse and retrieval facilities</td>
<td>*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S - research scope  m - model  p - process  t - tools  * - sub-thesis

Figure 8.1  Impact of the minimalist approach components on the issues.

Each central cell, corresponding to a pair (issue, activities), holds three values
to indicate which components of the minimalist approach contribute for reducing the severity of the issue, and the component with higher impact is highlighted. Because some issues are specific to production or usage activities, and some issues are not influenced by some components, some cells are left blank.

The next subsections assess and validate all the sub-theses of Figure 8.1, organized by the components of the approach: the documentation model, the documentation process, and the tools.

All the arguments and sub-validations presented in the next sections are merged and consolidated later in a second step.

8.2 Assessing the documentation model

To cope with all the complexity of framework documentation, the minimalist approach is based on a powerful and flexible documentation model that enables the organization of all kinds of contents into a uniform, modular, and orthogonal structure (Section 6.2).

The documentation model of the minimalist approach has several features that simplify the organization of contents:

- a meta-model to cope with the heterogeneity of concrete contents, without losing generality;
- powerful structuring mechanisms, such as the layering mechanism parameterized with document aspects;
- capability of defining explicit relationships based on implicit references;
- is extensible.

When producing documentation, the model of the minimalist approach helps authors with:

- combining arbitrary kinds of contents represented in different notations;
- providing template documents for the most frequent types of framework documents, such as cookbooks and pattern instances;
- the contents layering parameterized by predefined document aspects supported by the model;
- providing different views to a document element according to the
layer that is active;

- integrating contents informally and flexibly, at a conceptual level, with
  the support of the meta-model.

When using documentation, the model of the minimalist approach is almost
invisible to the reader, depending on how it is supported by tools, but it
helps readers with:

- understanding how contents are intrinsically organized, because the
  structure is uniform and consistent;
- understanding the implicit relationships between different kinds of
  contents (if configured during production);
- sharing and exchanging parts of the documentation, because the
  contents are loosely integrated (logically), not physically tied together.

Therefore, when compared with documentation models provided by
traditional approaches, such as those of literate programming systems based
on XML, the model of the minimalist approach is better because it is defined
at a meta-level, and addresses specific issues of framework documentation.

8.3 Assessing the documentation process

To document a framework economically and with quality it is necessary to
know what, when, and how to document. The process of the minimalist
approach is very simple and almost neutral, in order to be flexible, and easy
to adapt to different development processes and environments.

When producing documentation, the process of the minimalist approach
provides guidelines to help authors with:

- judging the quality of the documentation;
- combining different kinds of documents, such as task-oriented
  information and how-it-works information;
- organizing the documentation in a minimalist way;
- controlling, defining, managing, and optimizing their own
  documentation process to be more effective and reduce costs;
- refining the documentation to make it more usable, by listening to the
  readers comments and suggestions.

Therefore, when compared with ad-hoc processes commonly used to
document software or frameworks, the process is better because it is defined
and provides guidance on writing good documentation.

8.4 Assessing the documentation tools

To produce and maintain good quality framework documentation at a reasonable cost, there is a strong need for integrated software documentation environments that are easy to use, powerful, widely available, interoperable with the most common development environments, and supporting the major kinds of documents.

The minimalist approach contributes for this issue with XSDoc, an extensible documentation infrastructure that verifies all these requirements, and provides an attractive documentation environment that significantly helps authors and readers cope with many issues of producing and using minimalist framework documentation (Section 6.4 “Tools and Utilities — XSDoc”, page 178).

XSDoc is the most tangible component of the minimalist approach, through which the authors may benefit from the capabilities provided by the model and the process. It is therefore natural that the majority of the contributions to the problem of the minimalist approach are associated with XSDoc, as can be deduced from Figure 8.1.

The model dictates the way contents are integrated, organized, and related, and also supports extensibility and adaptability of information. The process provides guidance on how to document a framework in a minimalist way. Based on the conceptual model, and the process defined, XSDoc acts as a concrete vehicle to deliver to authors and readers of framework documentation the powerful structuring features of the model and the guidance provided by the process.

XSDoc was created to overcome the limitations in terms of tools interoperability, integrability of heterogeneous contents, and extensibility (Section 5.2.1) found on existing literate programming systems, and alternative software documentation techniques used to solve the semantic consistency problem of software documentation.

8.4.1 Flexible integration of contents

To integrate and ensure the semantic consistency between different kinds of software artifacts is an old issue of software documentation. In framework documentation this issue gains more severity because the contents to maintain in consistency in framework documentation are not only source code and documents, but may include also models, contracts and other
formal documents ("Semantic consistency", on page 95).

XSDoc preserves the semantic consistency between different kinds of contents, including source code, models, and documents, using a multiple source approach (Section 3.4.10) based on a dynamic and extensible XML-based integration mechanism, with which the contents are not statically integrated or synchronized when they change, but instead are dynamically integrated "on the fly" only when they are requested. A simple caching service helps optimize the efficiency of the process (Section 6.4.4 "Integration of multiple sources of contents", page 183).

The integration mechanism enables the definition of explicit static relationships between different kinds of contents, usually made by the author, one by one. In addition, the mechanism supports also the definition of rules to derive explicit dynamic relationships based on implicit relationships often hidden in the original artifacts, such as relationships between source code and application programming interface documentation, or reference manuals.

8.4.2 Wiki: effective, attractive, and easy-to-use

Once integrated, the contents are presented using a Wiki-based environment, through dynamic web pages containing dynamic links. All the functionality provided by XSDoc is available to authors and readers using the simplicity typical of a Wiki (Section 6.4.2 "XSDocWiki", page 180).

The Wiki concept (and tool) is used as an "attractive facade" for hiding the complexities of XSDoc internals.

The Wiki's provide perhaps the simplest way of editing and publishing simple contents on the web. As the extensions made to support software documentation follow the Wiki philosophy, XSDoc preserves the original simplicity typical of a Wiki.

The Wiki promotes cooperative edition of documents on the web, supports the appealing concept of "read-write web page", and enables the easy formatting and linking of contents. ("Wiki-based editors", on page 127).

XSDoc promotes the writing of documentation following the Wiki principles [Cunningham, 1999], which enforce the usage of comprehensive and consistent names, extensive cross-referencing of contents, modular documents, and concise writing.

8.4.3 Easy integration in development environments

The combined use of XML and Wiki makes the integration of XSDoc easy to achieve and successful in almost every development environment, only
with the low cost of small configurations in the environment, considering that XML tools are widely supported everywhere, the Wiki engine is multi-platform, and the Wiki pages only require a browser to be displayed.

Because XSDoc is web-based, it is easy to integrate in a wide range of development environments, from shell-based environments to modern integrated development environments, such as Eclipse [Eclipse, 2003] and Together [Borland, 2003].

The integration of XSDoc in open integrated development environments (IDEs) is possible with specific plugins that enable the use of a web browser inside the IDE through which the Wiki of XSDoc is accessed. Another requirement is a communication link between the IDE and the Wiki engine to enable their interoperation. With Eclipse, the interoperation is implemented directly in Java. Such integrations are very attractive for developers because they enable “side-by-side” edition and synchronization of all kinds of contents (source code, UML models, and documents) thus reducing the need to switch applications during development.

Web-based documentation environments, and documentation-enabled IDEs significantly help reduce the typical gap “developing - documenting”, one of the major reasons for the knowledge acquisition issue (“Knowledge acquisition”, on page 95) and programmer's attitude issue (“Programmer’s attitude”, on page 96).

8.4.4 Interoperability

XSDoc is totally integrated in a Wiki engine, through which it provides interoperability with external tools through Java language, client-side scripts, or executable server-side programs (Section 6.4.4 “Integration of multiple sources of contents”, page 183).

8.4.5 Effective guidance through templates and guidelines

Framework documentation requires an adequate mix of prescriptive information, explaining how-to-do, and also descriptive information, explaining how it works (“Combining prescriptive and descriptive information”, on page 97).

In order to guide authors on how to document a framework, XSDoc provides a set of templates and guidelines for the most common types of documents. Because the template mechanism is extensible, new templates can be added in order to easy adapt to different projects and environments.

From the point of minimalism, templates and guidelines can be seen as the
contents of a possible minimalist manual named “How to document a
framework?”, where the guidelines provide task-oriented information that help
authors getting started fast creating the documentation for their real framework
using the ready-to-use examples provided by the templates.

8.4.6 Widely available

XSDoc is implemented in Java language, on top of a Java-based open-source
Wiki engine, the VeryQuickWiki [Cronin and Barnett, 2003], and also
includes external programs developed with C++ and XML. All these
technologies are available in a wide range of platforms.

8.4.7 Main benefits of using XSDoc

By combining a Wiki engine, document processing with XML technology,
and easy integration in development environments, possibly in open IDEs,
XSDoc makes easier the production of framework documentation.

For writers When producing documentation, the tools of the minimalist approach help
authors with:

- preserving the semantic consistency between different kinds of
  contents;
- carrying out activities typical of documentation (Section 3.4.1) by
  providing an appropriate documentation environment;
- documenting their ideas and other arbitrary information they consider
  important to preserve, because it is very easy to do it with the
  environment provided;
- “sensing” that to document software is important, both to help future
  developments, and developments at hands, in the present;
- improving the quality of their documentation by following templates,
  organizing the documentation adequately, and introducing controlled
  redundancy to increase readability;
- adopting “good practices” of framework documentation, such as:
  combining task-oriented information with internal design
  information, “fading” information, providing several entry points to
  the documentation;
- integrating contents without explicit relationships using dynamic
  relationships created based on rules provided;
- reduce the costs of documentation;
- evolving the documentation to fit the real needs of the readers, because is easy to give feedback;

- documenting while they develop, without the effort of switching environments;

- automating the production of parts of the documentation considered repetitive and that doesn’t require human intervention, such as many parts of reference manuals, with the help of author-defined scripts that produce information that can be integrated in the overall documentation;

- sharing and exchanging parts of the documentation, because the contents are loosely integrated (logically), not physically tied together (also in Section 8.2).

For readers When using documentation, the tools of the minimalist approach helps readers with:

- using online documentation, printable with quality, eventually integrated with the development environment (readers are also developers);

- adapting the information presented according to their needs in terms of kind of audience, level of detail, kind of information (prescriptive vs. descriptive), layers, etc;

- navigating the documentation using different traversal paths, top-down, bottom-up, or combinations, due to the extensive hyperlinking of the information;

- searching and querying the information using powerful XML standards and tools;

- understanding how contents are intrinsically organized, because the structure is uniform and consistent (also in Section 8.2);

- understanding the implicit relationships between different kinds of contents (if configured during production) (also in Section 8.2);

- sharing and exchanging documentation, because the contents don’t need to be physically tied together, but only logically integrated (also in Section 8.2).
8.5 Consolidation

The preceding sections have presented and discussed the properties of each of the three component of the minimalist approach (model, process, tools) that help with producing and using framework documentation, and examined their effect on the corresponding sub-theses defined in Figure 8.1.

8.5.1 Summary of arguments

Several arguments were presented in order to validate each sub-thesis. Below is presented a summary of the strongest arguments used:

- **Model**
  - meta-model to support heterogeneous kinds of contents;
  - powerful structuring features (layers, aspects, filters);
  - flexible relationships mechanisms (static, dynamic);
  - templates for typical framework documents.

- **Process**
  - explains how-to document frameworks;
  - helps with controlling the documentation process (define-write-assess-listen-refine);
  - simple to follow.

- **Tools**
  - flexible integration of contents;
  - attractive and easy-to-use Wiki-based documentation environment;
  - interoperable set of tools;
  - helps getting started fast (templates, guidelines);
  - based on widely available technology (Wiki, XML).

From a practical point of view, these strongest arguments are associated with the tools, mainly because it is up to the tools to provide concrete, effective, and easy-to-use solutions to the issues addressed, what is considered to have been achieved.
8.5.2 Assessing the maturity of the minimalist framework documentation

As a complement to the validation based on arguments above presented, it is interesting to assess the quality of “minimalist framework documentation” based on documentation maturity models targeted to software documentation [Huang and Tilley, 2003; Visconti and Cookand, 2002], considering both the process and the product of the minimalist approach. Although these maturity models can’t be considered sufficiently “mature” to be used as a final word, they can however provide an useful indication.

**Process maturity**

Based on the process maturity model of [Huang and Tilley, 2003], which defines 5 levels (1-5), the minimalist approach ensures a level 2, named “repeatable”, and provides features that enable to reach a level 5, named “optimizing”.

The other process maturity model of [Visconti and Cookand, 2002] is targeted to assess an organization, not only the technical processes. Based only on levels of the key practices, which are 4 levels (1-4) the minimalist approach ensures a level 3, named “defined”, and provides features that enable to reach a level 4, named “controlled”.

Depending on how the inconsistencies in documentation are handled, in traditional documentation approaches the level ensured may be as low as level 1 in both models.

**Product maturity**

Based on the product maturity model of [Huang and Tilley, 2003], which also defines 5 levels (1-5), the minimalist framework documentation ensures a level 4, named “contextual”, or level 5, named “personalized”, in the text format and efficiency key product attributes, depending on the mode used to browse the documentation, static pages or dynamic pages.

Another way of assessing the quality of documentation as a product is proposed by [Arthur and Stevens, 1992; Stevens et al., 1988] as “documentation adequacy”, which includes attributes such as completeness, accuracy, and usability. Usability can be further divided in understandability, and logical traceability:

- “completeness”, defined as the likelihood of containing all the information required to satisfy the needs of the audiences for the purpose it was created;
- “accuracy”, defined as the correctness, exactness of the information provided, without any inconsistencies between any of its contents: textual descriptions, source code, models, cross-references, auxiliary information, and writing style;
- “understandability”, defined as the ease of comprehension of the
documentation; to be comprehensive requires a text structure that is modular, well-structured, concise, and include appropriate redundancy; to be comprehensive requires also the use of appropriate writing style, consistency of terms, and physical readability (format, print, etc.);

- "logical traceability", defined as the ability to follow the logical train of thought in the documentation through all of the relevant parts, independently of the parts being contiguous or not [Stevens et al., 1988]; this implies that the reader is capable of finding everything he needs using search facilities and references.

Therefore, adequate documentation means that it is complete, its contents are consistent, modular, well-structured, concise, has appropriate redundancy, is well-written, uses consistent terms, has a readable presentation, is searchable, easy to query, and easy to navigate (top-down and bottom-up).

It is important to mention that only a subset of these factors can be ensured by a documentation approach, which are: contents consistency, well-structured, readability, easy to search, easy to query, and easy to navigate.

In addition, the approach enforces and provides incentives for writers to consider some of the other factors, namely: modularity and consistency of terms.

Finally, because good documentation requires good writers, the factors that remain depend only on the writer's ability; they are: concise, appropriate redundancy, and well-written.

8.6 Conclusions

Because the thesis dissertation is the conjunction of a set of sub-theses, and the sub-theses were validated based on:

- the fine-grained sub-validations made on each of the issues presented in Section 8.2, Section 8.3, and Section 8.4,
- the strongest arguments presented in Section 8.5.1;
- the lessons learned from running the case studies, presented in Section 7.1, and Section 7.2 (JUnit, mainly),
- and the results of the controlled experiment Section 7.3.6,
- the maturity levels of the product and process possible to ensure with the minimalist approach assessed in Section 8.5.2;
the thesis validation can be concluded by stating that the minimalist approach to framework documentation makes the typical activities of producing and using documentation (much) easier to carry out respectively for documentation authors (and managers), and documentation readers, when compared to traditional documentation approaches based on framework overviews, examples and reference manuals.
Conclusions

The minimalist approach to framework documentation as a whole (model, process and tools) is cohesive and represents a considerable improvement over current practice on addressing the typical problems of framework documentation.

The minimalist approach to framework documentation is systematic, consistent, supported by tools, and combines the strengths of existing approaches, techniques and tools to document frameworks with the design theory of minimalist instruction. It is an evolutionary approach in the sense that it is based on existing knowledge, practice and tools, thus enabling a smooth adoption by framework developers, and easy usage by framework learners not aware of the approach.

This approach is the first comprehensive and integrated approach addressing the overall problem of documenting frameworks, being at the same time easy to use and expected to be easy to adapt to the needs of a majority of framework development projects and environments.

From the resulting work, it can be concluded that the approach helps reduce the severity of many issues of framework documentation. In addition, the research contributes with a workable approach, based on a conceptual
model, a pragmatic process, and an extremely easy to use documentation environment that promises to be useful not only to framework documentation, but for software documentation in general.

This chapter presents the main contributions made by this dissertation and outlines future research work to be done following this work.

9.1 Summary of Contributions

From the perspective of framework documentation, the main result is the contribution with a integrated and workable approach to document frameworks, which guides and supports writers on the overall process of producing framework documentation, primarily targeted for framework users, but not exclusively.

Another important contribution is the conceptual model that organizes all the typical contents of minimalist framework documentation in a very flexible and powerful way, which is easy to understand and easy to implement.

The last contribution is the tool that was built to support the approach and make it convenient for adoption in a wide range of environments. XSDoc steps forward on the direction of documentation-enabled development environments, which promise to change the attitude of developers in relation with the usefulness of documenting.

From the perspective of minimalism, the contribution was simply as another application of the theory to large and complex systems (frameworks).

In the end, the minimalism is not very tangible, because only a few characteristics are immediately visible, for who knows minimalism, while the majority are dissolved in design decisions made all over the work developed, spread out in many aspects of the approach. For example, the choice of a Wiki as the main documentation environment was made when looking for editors satisfying the requirements of being brief, and enabling writing and reading in any order.

9.2 Future Research

Many issues were addressed by this dissertation but it has also uncovered many other interesting areas for further research based on this work.

In particular, we would like to assess the effectiveness of the approach in
industrial settings and real projects.

Another area for possible future research is to explore the concept of Wiki-based software documentation, taking advantage of being largely used and infiltrated in development environments, and having good capabilities of integrating external tools that help automate the analysis, and production of documents, specially when based on XML technology.

Another area of research is on visualization techniques to help navigate in the documentation using the browsing controls proposed, namely zoom, resolution, aspects, and layers.

And finally, we expect to evolve XSDoc in order to achieve a tight integration of XSDoc in an open IDE.
References

[Agile Alliance, 2001]

[Agrawal et al., 1997]

[Aguiar, 1999]

[Aguiar, 2003]

[Aguiar and David, 2003]

[Aguiar et al., 2003]
[Aguiar and Vestdam, 2001]

[Aguiar and Vestdam, 2002]

[Aksit, 1999]

[Alexander, 1979]

[Alexander et al., 1977]

[Allen and Garlan, 1992]

[AlphaWorks, 1999]

[Ambler, 2002]

[Ambler, 2003]

[Anderson et al., 2002]

[Andert, 1994]

[Apache, 1999]

[Apple Computer, 1986]
[Arthur and Stevens, 1992]

[Ashman and Simpson, 1999]

[Badros, 2000]

[Bakker, 1987]

[Bäumer et al., 1997]

[Beck, 2000]

[Beck et al., 1996]

[Beck and Gamma, 1997]

[Beck and Gamma, 2003a]

[Beck and Gamma, 2003b]

[Beck and Gamma, 2003c]
[Beck and Johnson, 1994]

[Biggerstaff and Richter, 1987]

[Biggerstaff and Richter, 1989]

[Birtwistle, 1979]

[Bodner and Chignell, 1999]

[Bodner et al., 1999]

[Boehm, 1987]

[Booch, 1994a]

[Booch, 1994b]

[Borland, 2003]

[Bosch et al., 1999]

[Bray et al., 1998]

[Bremer, 1999]
[Briggs, 1993]

[Brockmann, 1990]

[Buschmann et al., 1996]

[Butler, 1997]

[Butler and Dénommée, 1997]

[Butler and Denommée, 1999]

[Butler et al., 2000]

[Campbell et al., 1991]

[Carroll, 1990]

[Carroll, 1998]

[Chai, 1999]

[Clark, 2003]
[Clements et al., 2002]

[Coates and Rendon, 2002]

[Collard et al., 2002]

[Conklin, 1987]

[Cotter and Potel, 1995]

[Cronin and Barnett, 2003]

[Cunningham, 1999]

[Dahl et al., 1970]

[Dean, 2002]

[Demeyer et al., 2000]

[Deutsch, 1989]

[Draper, 1998]

[Drye and Wake, 1999]
[D’Souza and Wills, 1999]

[Eclipse, 2003]

[edéis, 2003]

[Farkas, 1998]

[Fayad and Schmidt, 1997a]

[Fayad, 1999]

[Fayad and Johnson, 2000]

[Fayad and Schmidt, 1997b]

[Fayad et al., 1999a]

[Fayad et al., 1999b]

[Feiler and Meadow, 1996]

[Ferraiolo et al., 2003]


Also available online at URL http://java.sun.com/docs/books/jls/.
[Gossain, 1990]

[Grice, 2002]

[Gupta et al., 2002]

[Hansen, 1997]

[Haramundanis, 1999]

[Hargis, 1997]

[Hargis, 2004]

[Hartmann et al., 2001]

[Hedin and Knudsen, 1999]

[Helm et al., 1990]

[Highsmith, 2000]

[Hoydalsvik and Sindre, 1993]
[Huang and Tilley, 2003]

[Hueni et al., 1995]

[IBM, 2003]

[IBM Corporation, 1983]

[IEEE, 2001]

[INESC Porto, 2000]

[Jacobson, 1994]

[Johnson, 1992]

[Johnson, 1997]

[Johnson and Foote, 1988]

[Johnson and Russo, 1991]

[Jugel and Schmidt, 2003]


[Lajoie and Keller, 1995]

[Lea, 1994]

[Lewis et al., 1995]

[Linton et al., 1989]

[Luckham et al., 1995]

[Madsen et al., 1993]

[Mamas and Kontogiannis, 2000]

[Mattsson and Bosch, 1999]

[Medvidovic and Taylor, 2000]

[Meusel et al., 1997]

[Meyer, 1988]
[Meyer, 1992]

[Meyrowitz, 1986]

[Mili et al., 1995]

[Mili and Sahraoui, 1999]

[Minto, 1991]

[Monday et al., 2000]

[Moriconi et al., 1995]

[Moser and Nierstrasz, 1996]

[Naur and Randell, 1968]

[Nawrocki and Wojciechowski, 2001]

[Nørmark, 2000a]

[Nørmark, 2000b]
[Nørmak, 2000c]

[Nørmak et al., 2000]

[Oatey and Cawood, 1997]

[Odenthal and Quibeldey-Cirkel, 1997]

[OMG, 2001]

[OMG, 2003a]

[OMG, 2003b]

[Opdyke, 1992]

[Østerbye, 1995]

[ParadigmaXis, 2001]
ParadigmaXis (2001). simatware application framework. Internal technical report, ParadigmaXis SA.

[Parikh and Zvegintzov, 1983]

[Parnas and Clements, 1986]

Prechelt and Unger, 1998


Pree, 1991


Pree, 1995


Pree, 1999


Prosise, 1999


PT Inovação, 2003


http://www.ptinovacao.pt/.

Quatrani, 1998


Quintino, 2003


Ramsey, 1994


Rapicault, 2002


[Schappert et al., 1995]

[Schmid, 1997]

[Schmidt, 1995]

[Schmucker, 1986]

[SEI, 2003]
SEI (2003). Software Engineering Institute web page on Architecture Description Languages (ADLs).

[Shaw and Garlan, 1996]

[Shull et al., 2000]

[Silva et al., 2000]

[Silveira et al., 2003]

[Smart, 2002]

[Soloway et al., 1988]

[Sommerville, 2000]
[Sousa, 1999]  

[Spencer, 1988]  

[Stevens et al., 1988]  

[Storey et al., 1999]  

[Sun Microsystems, 2003]  

[Szyperski, 1998]  

[Taligent Inc., 1993]  

[Taligent Press, 1994]  

[Taligent Press, 1995]  

[Tichy et al., 1993]  

[Tilley, 1993]  

[Tracz, 1995]  
[van der Meij and Carroll, 1998] 

[van Heesch, 2002] 

[Vestdam, 2003] 

[Visconti and Cookand, 2002] 

[Vitali and Bieber, 1999] 

[VKI, 2003] 

[Vlissides and Lint, 1990] 

[von Mayrhauser and Lang, 1999] 

[von Mayrhauser and Vans, 1994] 

[Waite, 2002] 

[Walsh, 2002] 
[Wegner et al., 1992]

[Weinand et al., 1989]

[WID, 2003]

[ Wilde et al., 1993]

[Williams and Kessler, 2000]

[Williams, 1992]

[Wirfs-Brock et al., 1990]

[Wirfs-Brock and Johnson, 1990]

[World Wide Web Consortium, 1999]


[Wright, 1994]

[Wunderling and Zockler, 2002]


