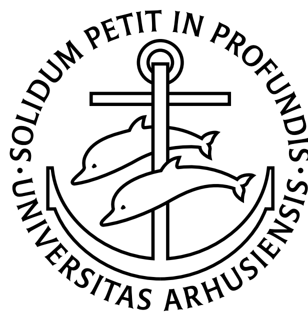
Patterns in Software Development

Aino Cornils

PhD Dissertation



Department of Computer Science
University of Aarhus
Denmark

Patterns in Software Development

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Aino Cornils
November 19, 2001

Forord

If you don't look after knowledge, it goes away
— Terry Pratchett, The Carpet People, p. 36.

Denne afhandling bygger på et ønske om at gøre verden til et bedre sted at leve i. Man kunne argumentere, at jeg med det ønske skulle have valgt et andet studie, som for eksempel marinøkologi, medicin eller bioteknologi. Men det gjorde jeg ikke og nu må jeg tage konsekvensen og med begge fødder solidt plantet i datalogi og matematik gøre verden bedre herfra.

Man skal jo starte et sted og da min kollega Ellen Agerbo og jeg af vores daværende specialevejleder, Ole Lehrmann Madsen, blev gjort opmærksom på eksistensen af det nye begreb, *mønstre*, satte vi os for at undersøge om de virkelig kunne lette systemudviklingens forskellige faser.

Mønstre er beskrivelser af abstrakte løsninger på ofte forekommende problemer og de fordele der påstås brugen af dem har, er: fælles ordforråd, forbedret dokumentation og videreførelse af erfaring. I løbet af vores specialeskrivning blev vi opmærksomme på at ting kunne være bedre, og at man burde gøre noget for faktisk at få fordelene ud af at anvende mønstre. Da vi startede på ph.d. uddannelsen med Jørgen Lindskov Knudsen som vejleder arbejdede vi det første år sammen på at bedre situationen og foreslog som løsning en mængde retningslinjer til klassifikation af design mønstre og et bibliotek af design mønstre. Derefter afbrød Ellen sin uddannelse for at tage job i det private erhvervsliv og jeg gik selv i samarbejde med det private erhvervsliv via et projekt indenfor Center for Objektteknologi (COT). Her var fokus for mit vedkommende på formidling af mønstre til industrien på en letfordøjelig måde, der kunne udbrede brugen af mønstre og dermed indføre det fælles ordforråd, udvekslingen af erfaringer og, som en konsekvens heraf, forbedre dokumentationen.

I det obligatoriske udenlandsophold valgte jeg at besøge Görel Hedin, som i sin tid havde introduceret mig til mønstermiljøet. Det blev et halvt år, hvor jeg videreudviklede ideen om et bibliotek af design mønstre til brug i systemudvikling. Resultatet blev et værktøj til hjælp ved anvendelse af, og dokumentation med, design mønstre.

Alt i alt har forskningen i denne afhandling potentiale til at hjælpe udviklere i deres systemudvikling og vedligehold af systemer og dermed i sidste ende gøre verden til et bedre sted at være for systemudviklere og indirekte for dem, der bruger deres system.

Foreword

If you don't look after knowledge, it goes away
— Terry Pratchett, The Carpet People, p. 36.

This thesis springs from a wish to make the world a better place to live in. It could be argued that other studies, like marine ecology, biotechnology or medicine might be more suited for that wish. But the subjects I studied were computer science and mathematics and I had to take the consequences and make the world better from there.

My interest in the concept *patterns* was initiated by Ole Lehrmann Madsen, our then supervisor, who brought our attention to this new concept. We decided to investigate the extent to which patterns could ease the various phases of system development.

Patterns are descriptions of abstract solutions to often occurring problems, and the benefits they are claimed to provide, are: common vocabulary, enhanced documentation and shared experience. In the writing of our masters thesis, it occurred to us that things could be better, and that something had to be done to actually achieve these claimed benefits. When we began our Ph.D. with Jørgen Lindskov Knudsen as supervisor, Ellen and I worked together the first year to improve this situation and we proposed a set of guidelines for a classification of design patterns and a class library of design patterns.

After this, Ellen chose to leave the university and I entered a project connected to industry within Centre for Object Technology (COT). I focused on promoting the pattern concept to the Danish software industry in order to help them take advantage of the benefits of applying patterns in system development.

In the obligatory stay abroad, I chose to visit Görel Hedin at Lund Institute of Technology. During my months there we developed the idea of a class library for design patterns further. The result was a CASE tool that uses reference attribute grammars to ease the application of, and documentation with, design patterns.

All in all the research described in this thesis has the potential to help developers in their system development and thereby, in turn, make the world a better place to live in for developers.

Main Contributions

This section gives a brief account of the four main contributions of this dissertation.

The role of patterns in system development
Since the emergence of the concept *patterns* in software it has been discussed thoroughly what patterns are. There is an endeavour to place patterns in the extension of an already existing concept, like methods, techniques or such. This dissertation seeks to show that patterns are orthogonal to these existing concepts.

Classification of patterns
The rapid evolution of design patterns has limited the benefits gained from using design patterns. The increase in the number of design patterns makes a common vocabulary unmanageable. This dissertation presents an analysis of design patterns that will strongly reduce the number of fundamental design patterns.

Library of design patterns
Over the past years, along with the increase in popularity of design patterns, some problems with the use of design patterns have been identified. The so-called *tracing problem* describes the difficulty in documenting software systems using design patterns. The tracing problem obscures the documentation that should be enhanced by using Design Patterns. This dissertation shows how strong language abstractions can solve the tracing problem and thereby enhance the documentation.

Tool support for patterns: the $\mathcal{DPDOC}$ tool
The solution to the tracing problem based on a library of design patterns was followed by a design pattern tool. This dissertation presents a flexible and extensible tool which enables designers to use design patterns in a safe and efficient way, which checks the design pattern rules, and which semi-automatically documents, and maintains the documentation of, a software system.

Acknowledgements

I would like to thank Centre for Object Technology for believing in my ideas and supporting me throughout my Ph.D.

Also thanks to:
Ellen Agerbo for working with me all the way through our masters thesis and guiding me away from all my most unrealistic ideas.
Ole Lehrmann Madsen for supervising the masters thesis I wrote with Ellen Agerbo and guiding me through the Ph.d. application.
Jørgen Lindskov Knudsen for supervising me through my Ph.D. and with a gentle hand removing all the jokes from my papers and my dissertation.
Görel Hedin for teaching me about design patterns, making me believe I could do a Ph.D. and finally supervising me for several months in Lund.
Erik Corry for giving me a massage when I needed it and proofreading my Danglish :-)
Kasper Østerby for opening my eyes to literate programming.
Mads Torgersen and Kresten Krab Thorup for many coffee-breaks and discussions about almost anything.
Ulrik Pagh Shultz for giving me ideas.
My daughter Maja for always reminding me what is important in my life.

Aino Cornils,
Århus, November 19, 2001.

Contents

Forord v

Foreword vii

Main Contributions ix

Acknowledgements xi

1 Introduction 1

1.1 A Definition of Patterns . . . 1
1.2 Motivation . . . 2
1.2.1 Communication of experience . . . 3
1.2.2 Common vocabulary . . . 3
1.2.3 Enhanced documentation . . . 3
1.3 Experiences with patterns . . . 4
1.3.1 My Approach to patterns . . . 4
1.4 Centre for Object Technology . . . 5
1.5 Chronology and Dissertation Outline . . . 6

2 History of Patterns 9

3 Philosophy of Patterns 13

3.1 Swarm Intelligence . . . 14
3.2 Alexanders' Paradigms . . . 14
3.3 Patterns and Eastern Philosophy . . . 15
3.4 The Paradigm in Software . . . 16
3.5 Alexanders' Reply . . . 17
3.6 Discussion . . . 18

4 Object-oriented System Development 19

4.1 Processes . . . 19
4.1.1 RUP . . . 19
4.1.2 eXtreme Programming . . . 21
4.2 Methods . . . 23
4.2.1 The OOA and D method . . . 23
4.2.2 OODA . . . 25
4.3 Techniques . . . 27

4.3.1 Class-Responsibility-Collaboration card (CRC card) . . . 27
4.3.2 UML . . . 28
4.4 Technologies . . . 30
4.4.1 Programming Languages . . . 30
4.4.2 Tools . . . 31

5 Patterns 35
5.1 Comparing patterns with existing concepts . . . 35
5.1.1 Synthesis . . . 37
5.2 Patterns in the Development Process . . . 37
5.2.1 UML and Patterns . . . 37
5.2.2 The COT Experience . . . 38
5.3 The Role of Patterns in System Development . . . 39

6 Overview of Pattern Categories 43
6.1 Pattern Templates . . . 43
6.2 Analysis Patterns . . . 44
6.3 Architectural Patterns . . . 46
6.4 Antipatterns . . . 46
6.4.1 Analysis Paralysis . . . 47
6.4.2 Death by Planning . . . 49
6.5 Organisational Patterns . . . 49
6.6 Database Patterns . . . 50
6.7 Design Patterns . . . 51
6.8 Case: Developing a planning system . . . 54

7 Classification of Design Patterns 59
7.1 An Analysis of Design Patterns . . . 59
7.1.1 The Analysis . . . 61
7.1.2 Applying the Analysis . . . 65
7.2 Related Work . . . 76
7.3 Conclusion . . . 77

8 Library Design Patterns - LDPs 79
8.1 Solving the Tracing Problem by certain language features . . . 79
8.1.1 Simulating Multiple Inheritance by Composition . . . 81
8.1.2 Implementing the LDPs . . . 82
8.2 Related Work . . . 86
8.3 Conclusion . . . 86

9 The Design Pattern Tool $\mathcal{DPDOC}$ 89
9.1 Providing Better Documentation . . . 89
9.2 The Tool . . . 90
9.3 Implementation . . . 93
9.3.1 Application- and $\mathcal{DPDOC}$-programmer interface . . . 93
9.3.2 The interface between the programming language and the design patterns . . . 96

9.3.3 Pattern specification framework 97
9.4 Extension of the tool 97
9.5 The Names Facility 98
9.6 Exploiting APPLAB 99
9.7 Future work on $\mathcal{DPDOC}$ 100
9.8 Related Work 101
9.9 User evaluation of the tool 104
9.10 Conclusion 104

10 System Development Revisited 107
10.1 Literate Programming 107
10.1.1 Example - Elucidative Programming 108
10.2 Aspect-Oriented Programming 110

11 Conclusion 113
11.1 Summary 113
11.2 Future Work 114
11.3 Conclusion 115
11.3.1 Patterns in Development 115
11.3.2 The Classification 116
11.3.3 The Library of Design Patterns 116
11.3.4 The Design Pattern Tool $\mathcal{DPDOC}$ 116
11.4 Summa summarum 116

A Extensions of Beta and UML 117
A.1 Proposed extensions of BETA 117
A.1.1 Virtual Nonterminals in BETA 117
A.1.2 Simulating Virtual Patterns Using Virtual Nonterminals . 119
A.1.3 Virtual Lists 121
A.2 Extended UML notation 123
A.2.1 Notation for Nested Classes 123
A.2.2 Notation for Singular Objects 124
A.2.3 Notation for Virtual Classes 124

B The Specifications of Decorator in DPDOC 127

C Publications, teaching and presentations 133
C.1 Publications 133
C.1.1 Conferences 133
C.1.2 Workshops 133
C.1.3 Technical reports 133
C.2 Presentations 134
C.3 Teaching 134
C.4 Other activities 135

Bibliography 137

List of Figures

4.1 The use case is the common thread between the different models. 20
4.2 The object system shows a future user's perception of the software systems problem domain. 24
4.3 The Main Activities 25
4.4 The four steps in Booch's method 26
4.5 Class-Responsibility-Collaboration card 28
4.6 A taste of UML 33

5.1 The focus of the different patterns in the development process. . 39

6.1 Observations and category observations 45
6.2 The phenomenon formerly known as category 45
6.3 **Broker** - An architectural pattern 47
6.4 Design pattern and Antipattern concepts 48
6.5 Structure of the database pattern **Physical views**. 52
6.6 The class diagram - **Composite**. 53
6.7 The class diagram for **MVC** 54
6.8 The collaboration between **Observer** and **MVC** 55
6.9 The analysis model for planning, **Resource allocation** 56
6.10 The overall system design 57

7.1 Two families of design patterns 64
7.2 The **Factory Method** Design Pattern 65
7.3 **Factory Method** modelled using virtual classes 66
7.4 The **Template Method** Design Pattern 66
7.5 *single hook* **Template Method** 67
7.6 The **Observer** Design Pattern 68
7.7 The **Mediator** Design Pattern 69
7.8 The **Visitor** design pattern 70
7.9 The collaboration in **Visitor** 71
7.10 The **Strategy** Design Pattern 71
7.11 The **Object Adapter** Design Pattern 73
7.12 The **Class Adapter** Design Pattern 73
7.13 **Adapter** using nested classes 74
7.14 Analysis of design patterns from [GHJV95] 76

8.1 Documenting the application of design patterns with UML . . . 80
8.2 The immediate way of using a LDP 81

8.3 **Decorator**-LDP 83
8.4 **Flyweight**-LDP 85

9.1 The **Decorator** design pattern structure 91
9.2 The user interface 93
9.3 Excerpts from the grammar modules 95
9.4 Reference attributes connect the design pattern applications to the program 96
9.5 The pattern specification framework 98

10.1 Dual usage of a WEB file 108
10.2 The navigational proximity between documentation and program. 109
10.3 The box named MoveTracking shows an aspect that cross-cuts methods in the Point and Line classes. 111

A.1 Nested classes 124
A.2 Inheritance in nested classes 124
A.3 Singular Objects 125
A.4 Further virtual bindings in subclasses 125
A.5 Virtual classes; anonymous extension 126
A.6 Anonymous virtual classes 126

Chapter 1

Introduction

Within software development it is possible to save time via reuse, not only on code level, but also on a more abstract level with ideas. Experienced developers reuse their own development ideas by recognising many problems as similar to problems they have solved before and apply the best solution they used when they last encountered the problem. Because of this reuse of ideas, some solutions appear as recurring patterns in the good designs. *Patterns*, introduced as concept in software in the late 1980's, provided a way to preserve this experience.

1.1 A Definition of Patterns

In general, a pattern description has four essential elements:

- The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names is one of the biggest challenges in describing patterns.

- The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

- The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements solves it.

- The **consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.

The above is taken from the introduction of the GoF book, and present the most important aspects of a pattern description. Every catalogue of patterns extend the description of their patterns with more descriptive elements. In the GoF catalogue of design patterns the pattern descriptions each consist of 13 descriptive elements. Some of these can be projected to the four essential elements described above, the rest are extra information, not necessary to describe the pattern.

The **Name** and **Also known as** give the *name*(s) of the pattern. The **Motivation** and the **Applicability** together constitutes the description of the *problem*, that the pattern solves. The **Structure**, **Participants** and **Collaborations** comprise the *solution* and the *consequences* can be found in the **Consequences** section. The descriptions also contain the **Intent**, which is a few lines describing the intent behind the pattern solution, making it possible to scan through a lot of pattern descriptions efficiently, **Sample code** (in C++ or Smalltalk), **Known uses** (the solution has to have been used at least twice to constitute a pattern) and finally **Related Patterns** giving references to other patterns, that are variations on the same theme or maybe solves a partial problem, that can not be solved by the pattern at hand.

As can be seen from the pattern descriptions in Chapter 6, the different categories of patterns contain different types of descriptions. All pattern authors choose their own style in pattern descriptions, but the descriptions all contain the four essential elements; name, problem, solution and consequences.

1.2 Motivation

The application of patterns has shown to have three main benefits; Communication of experience, because the experienced developers' experience can be written down in an easily accessible way via patters; a common vocabulary, because patterns allow people to communicate on a higher level of abstraction as can be seen with data structures and algorithms; and enhanced documentation because the higher level of abstraction allows for a more concise documentation than traditional documentation.

Another benefit is that the burden of maintenance is reduced with the use of patterns. Patterns are aimed at flexible designs and the application of patterns will induce maintainable systems.

When patterns are used in a development process, the basic idea in the design can be described using patterns. The knowledge, that would otherwise be implicit knowledge between the developers, can now be made explicit and thereby easier to take advantage of. The developers will follow the same idea, which makes the design process more effective.

Naturally, where there are benefits, there are also liabilities, and this goes for patterns as well. The biggest disadvantage is that developers new to patterns have a tendency to overuse them. They are so impressed by the concept, that they seek to use patterns everywhere. But there do not exist pattern solutions for all problems and it is not a good idea to change the problems to fit a pattern. Patterns should fit the problems.

1.2.1 Communication of experience

As in other system development disciplines there are recurring problems in object-oriented design. Many of these problems have an immediate solution with unwanted consequences. It can be hard to foresee the consequences of a solution, and without sufficient design experience, the immediate solutions will often be chosen. When a developer has gained experience, he will be able to recognise patterns in the problems he sees and use the solutions with the best consequences. If these solutions are written down as patterns, other developers will be able to benefit from this experience.

1.2.2 Common vocabulary

In addition to getting the ability to communicate experience with patterns, the developers will be able to discuss development on a more abstract level. Each pattern covers a small sub-design, and can be identified by its name. When the developers, with knowledge of patterns, discuss potential solutions to a problem, they use the names of the patterns to communicate complex ideas in an exact and concise way. If patterns were universal and used both in education and industry, they could be used as algorithms and datastructures are used now.

1.2.3 Enhanced documentation

As the communication within development discussions can rise to a more abstract level using patterns, so can the documentation. The task of documenting systems can be made more efficient and thus, in effect, may facilitate the creation of documentation in situations where it might otherwise have been omitted. The developer can document a whole subsystem by indicating the pattern used and the roles played by the classes. This brief information will give the reader knowledge of how the classes are related to each other, what responsibilities they have, and what interfaces they respond to.

For every pattern, there is a description of the context. By stating which pattern is chosen, the problem that led the developers to chose the pattern is thereby, roughly, also stated. In traditional documentation there is almost never a description of what led the developers to do what they did. If the reader of the documentation also has knowledge of patterns, he will be able to see the implicit information that lies in the selection of a pattern solution. Thus, documentation with use of patterns can give more information than traditional documentation, thereby enhancing the value of the documentation. Rumbaugh's book on the UML notation [RBWPL91], contains descriptions of how applications of patterns can be shown directly in the documentation.

1.3 Experiences with patterns

A number of researchers from the pattern community have collected their experiences with patterns in the paper "Industrial Experience with Design Patterns" [BCC+96], that can be found via the patterns homepage [uiu].

One of the authors of the paper, John Vlissides, is also co-author of the GoF-book. In the paper he describes how the use of patterns affected his way of working as a consultant. He had been working as a consultant for about 4 years, when he in 1993 started using raw patterns descriptions in his work. These raw descriptions should later become a part of the GoF book. Two things had been annoying him over the years, which was what had initiated the use of patterns. Firstly, it was impossible to make the developers describe their design. This was a result, both of a lack of communication tools, and an inability to see the relations in their own design. Secondly the developers had designed the system without attention to why they had chosen the specific design. The reasons for the design choices was never remembered let alone written down. The result was that changes in the user requirements could not immediately be transferred to the design, because it was unintuitive where the changes should be made in the design.

When John Vlissides started using patterns, he did it by comparing the design made by his clients with the patterns he had encountered in his own design experience. If their design fitted the patterns he knew, he was able to associate parts of their design with the parts in the patterns and see, whether they could be used. In that way he created a frame of reference, when they discussed design. It became easier to communicate designs and easier for him to pose the right questions.

In addition to six such stories of personal experience with patterns, the paper contains an accumulation of common experience. The most important points are:

Firstly, the concept of patterns is a good means of communication, because it arms the developers with a common vocabulary with precise and concise design descriptions. Secondly, the descriptions of design with patterns eases the developer's task of following their common initial design decisions. These decisions are often implicit and by making it explicit, the development process is made more effective. Thirdly, patterns are useful for software design education. Experience and good design ideas are described in an accessible way that suits the teaching situation.

1.3.1 My Approach to patterns

As can be seen from the topics I have chosen to work with concerning patterns, my main interest has always been on the usefulness of patterns. I have never doubted that the idea of saving knowledge in pattern descriptions for later use is beneficial for software development. All the benefits we get from applying patterns, both in communication, learning, development and maintenance of

systems are invaluable. Therefore the easier it is to find the right pattern and use it, the better for the software community as a whole.

1.4 Centre for Object Technology

My Ph.D was sponsored by Centre for Object Technology, which in turn is sponsored by the Danish National Centre for IT-Research (CIT). Centre for Object Technology, abbreviated COT, is a Danish collaboration project with participants from Danish industry (A. P. Møller, Bang & Olufsen, Danfoss, Maersk Line, Maersk Tranining Center, Odense Steel Shipyard, RAMBØLL, Systematic Software Engineering, and WM-data), research institutes (Computer Science Department, University of Aarhus, Department of Computer Science at University of Copenhagen, The Maersk Mc-Kinney Møller Institute for Production Technology, University of Southern Denmark), and technological service institutes (Danish Maritime Institute, Technological Institute) The overall goal of Centre for Object Technology (COT) is to carry out research, development and technology transfer in the area of object-oriented software construction. The activities in COT are based on actual needs from the participating industrial partners.

One of the major obstacles to the utilisation of object technology is the various many existing systems that are in use within organisations. It is economically and organisationally impossible to rewrite all these systems to be based on object-oriented technology. Therefore one of the projects in COT had as it's purpose to study the integration of OO technology with legacy systems. The participants in this project were: Technological Institute, Computer Science Department, University of Aarhus, WM-data and RAMBØLL.

This project focused on the issues involved in the integration of object technology into existing systems, and the focus was in particular on the integration of object technology into administrative systems. Partly due to the nature of administrative systems, one of the major focus points of this project was the database technologies involved in these systems (mostly relational database systems), and the relations to the newer object database technologies (object-relational and object-oriented database technologies).

Moreover, due to the more and more decentralised organisations and the tighter cooperation between companies within these sectors, this project was also focused on system architectures for decentralised systems (distributed systems), and the role of object technology within these areas.

Finally, it was an important aspect of this project to experiment with ways to introduce object technology into these organisations.

Within this area, my main focus was on the architecture of object-oriented systems. The goal here was to enhance reuse of design and code. This was done by working with architectural elements such as (application-) frameworks, design rules and design patterns, component-based architectures, distributed objects, open implementations, documentation, etc.

1.5 Chronology and Dissertation Outline

The concept of patterns was first introduced to the world of (building) architecture by the architect Christopher Alexander. Some 15 years after, it emerged into software. Patterns are seen by many to be *the* way of looking after knowledge within software development. With that common goal, people still disagree as to what extent patterns can help us. Some, like James Coplien, believe that "patterns are truth". He, among others, operates in one extreme of the patterns movement. They are religious about patterns and tend to have strong opinions about them that few others can relate to, let alone understand. Another extreme is the engineer-minded people, who acknowledges that they can use patterns as a so-called literary form and thus describe their experience for the benefit of others. This includes people like Frank Buschmann and Erich Gamma.

To understand these disagreements one has to know the history of patterns. In this dissertation Chapter 2 is dedicated to this history because it has such a great influence on the field of patterns and the pattern movement of today. Derived from the philosophy of Christopher Alexanders patterns is the philosophy of software patterns. Some say that it is the same philosophy, others disagree. A discussion on this, included in Chapter 3, enlights the different ways of using patterns in software. Based on that discussion, the implication is that patterns are, if nothing else, useful in software development, in that they can be used as both a literary form and a philosophy to enhance software development in all its facets. The research has over the last years been dedicated to making patterns accessible and easy to use. One way of making patterns easily accessible is to classify them in order to get a better overview of the field of patterns.

The pattern hype was at its peak, when Görel Hedin encouraged my colleague, Ellen Agerbo, and me to write a paper on implementing the design patterns from the most popular book on design patterns [GHJV95] (the GoF-book) in BETA to ascertain whether the patterns were easier to implement in BETA than in C++ or SMALLTALK (as used by the authors). The paper on this work, 'Implementing Design Patterns in BETA ' [AC97a], was accepted on a workshop at the ECOOP'97 conference, and contained first attempts to change the religious view[1] on patterns. Our presentation triggered a lively discussion at the workshop.

Since our ideas obviously were interesting to elaborate on, it became the beginning of our master's thesis "Theory of Language Support for Design Patterns." [AC97b]. The thesis contains a thorough discussion of all the patterns in GoF, based on their implementation in BETA and our work with a new idea; making a class library of design patterns. We concluded in the thesis that it was beneficial to have a critical view on design patterns, that patterns can show which language constructs are useful and that it would be a good idea to make a library of design patterns. For some of the more theoretical discussions on

[1] "Don't question the patterns in the bible" [GHJV95].

patterns versus language constructs, we looked into proposed extensions to the BETA language and extensions of the UML notation. Because these are applied in later chapters, this chapter from the thesis is included in Appendix A in a partially rewritten form.

After our masters degree, Ellen Agerbo and I started our Ph.D. Our thesis was accepted as a part of our Ph.D.

We wrote a paper "How to Preserve the Benefits of Design Patterns." [AC98] which was a further development of material from our master's thesis. It contained a proposal of a classification of the design patterns from the GoF-book. The classification divides the design patterns into related families and eases the task of finding just the right pattern. The classification discussion from the paper, extended with discussions from the masters thesis can be found in Chapter 7. In the paper was also a proposal for a library of design patterns, based on the masters thesis. Patterns are abstract descriptions of solutions and can therefore be applied in a number of ways. In our research we tried to identify the kernel of each design pattern to see what could be reused for every application of the design pattern. These kernels are placed in a library and are called *library design patterns* (LDPs) and the proposal is described in Chapter 8.

This paper was accepted as a technical paper at OOPSLA'98. At the conference our ideas were presented to a larger audience, and we found that there was great interest in our work. It turned out that we had touched a common nerve, since confusion with the large number of patterns was a widespread feeling.

Some researchers had done work parallel to ours, trying to find a way to classify the patterns, and we had many fruitful discussions with researchers at the conference. There was also an event called "A bowl of patterns" at the conference, in which widespread interest was expressed in following the idea of easing the choice of the right pattern for the task, for example by classification of patterns in the way proposed by Ellen Agerbo and me. Linda Rising is one of the pioneers of the application of patterns and she wrote "The Patterns Handbook" [Ris98], a catalogue of pattern references the same year for the same reasons. A few years later, she wrote "The Pattern Almanac" [Ris00] with more references. It is indeed necessary to have some point of reference to the field of patterns, preferably with an indexing as ours but a pure handbook of references is also very useful when using patterns.

After we had given our paper, I turned my attention to the Danish industry through Centre for Object Technology. As already mentioned, the activities in COT are based on actual needs from the participating industrial partners, and thus it was possible for me to get in touch with different companies and learn how they could benefit from patterns. The concept was new to most of them, although they had the GoF-book on their shelves. Together with people from the industry I wrote an introduction to patterns [COV99] in the shape of a Danish-language technical report to the Danish audience. We had a lot of

fruitful discussions of the concept, the way to introduce it to Danes, and its role in system development. In Chapter 5 a part of this report is included and an extension of the discussions we had then.

For my six months abroad I was able to visit my former teacher Görel Hedin in Lund. During that period we implemented a design pattern tool called *DPDOC* described in Chapter 9. The chapter is based on the two papers we wrote to describe the practical [CH00a] and the theoretical [CH00b] aspect of the tool. *DPDOC* is a tool, that can help programmers apply design patterns in the right way, and semi-automatically document the code throughout the development and the maintenance of the system. It builds partly on the idea of a reusable kernel, that was discussed in Chapter 8, but is in some ways more elegant. An example of a specification of a design pattern is given in Appendix B.

The chapters mentioned below are supporting chapters to the ones already mentioned.

Chapter 4 describes my view on system development from which the discussions in the dissertation stem. Chapter 6 describes different types of patterns and gives examples of them. Chapter 10 revisits system development in the light of the contributions of the dissertation and compares *DPDOC* with literate programming and aspect-oriented programming. Chapter 11 concludes the dissertation, summarising its contributions, and discusses the possibilities and need for further research.

Chapter 2

History of Patterns

The history of software patterns starts in a branch of architecture developed by an Austrian architect called Christopher Alexander. Christopher Alexander graduated from Cambridge University with a degree in Mathematics and Architecture. Later he obtained a Ph. D. in Architecture at Harvard University.

Christopher Alexander observed in his work with urban planning and building architecture that "most of the wonderful places of the world were made by people, not architects". This observation led him to write his huge experience with good building patterns down for the future users of the houses to apply in their own homes. People should design for themselves their own houses, streets and communities, using his patterns. "A Pattern Language" [Ale77] was the book which contained the actual patterns, while other books [Ale64], [Ale75], [Ale79] described the theory behind the pattern language and applications of the patterns. He presents a pattern language of architecture, in which he gave advice on how to design human environments so that these would encompass the *quality without a name*[1]. The Pattern Language was formed as a collection of patterns with each pattern an atomic entity, describing how to design a doorknob, a room with a window, a house or a town; his idea was to make the entire world a better place to live in, filled with the *quality*, by describing all entities scaling from the smallest detail to the largest pattern.

In 1987 after studying Alexander's patterns, Ward Cunningham and Kent Beck decided to use some of Alexander's ideas to develop a small 'pattern language' for user interfaces. Their intention was to let representatives of the users finish the design. Since these people naturally were novice Smalltalk programmers, the patterns were meant to help them take advantage of Smalltalk's strengths and avoid its weaknesses. The users succeeded in developing an elegant user interface based on the patterns and Beck and Cunningham wrote up the results and presented them at OOPSLA'87 in Orlando in the paper "Using Pattern Languages for Object-Oriented Programs" [BC87]. At that point in time, they expected to cover the field of object-oriented programming with 100-150 patterns.

Other computer scientists followed their trail and described their experience by patterns. It was evident that the field of object-orientation needed something

[1] [Ale79, pp. 19 ff]

to make the learning curve less steep, and it was possible that patterns were the answer. Only very few developers mastered the skill of creating strong, flexible systems and it was often the most experienced ones, that were the best. Patterns promised to enable the developers to capture and communicate their experience.

Most famous of the collections of patterns is the book ‘Design Patterns: Elements of Reusable Object-Oriented Software” [GHJV95] written by Gamma, Helm, Johnson and Vlissides. These four authors are often referred to as *The Gang of Four*, which is why the book is known as the ‘GoF-book’. Obviously the authors didn’t just get together and write this literary monument. Many events had occurred beforehand. The *Workshop on Reflection and Meta-Level Architecture* started by Bruce Andersen in 1991 was repeated in 1992 where Frank Buschmann’s first publication on patterns [BKPS92] was presented. Jim Coplien had been cataloguing language-specific C++ patterns he called idioms, and Addison-Wesley published the book [Cop92] in September 1991. Peter Coad had been exploring patterns in parallel as well; he published an article [Coa92] in CACM in 1992. In August of 1993, Kent Beck and Grady Booch initiated the emergence of a group of people converged on foundations for software patterns. The group was called *The Hillside Group* and it planned the first PLoP conference in early April 1994. The conference took place on August 4, at the Allerton Park estate near Monticello, Illinois. The PLoP proceedings came out in May 1995 as “Pattern Languages of Program Design.” [Cop95b]. In the meantime, the *Gang of Four* had finished their work and the first major compendium of patterns made it out in time for OOPSLA ’94.

The Siemens group, with Frank Buschmann as the head figure, wrote the POSA books: “Pattern-Oriented Software Architecture - A System of Patterns” [BMR+97] and three years later “Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects” [SSRB00]. Their first book contains general patterns on three different levels, idioms - coding patterns, design patterns - like those in GoF, and architectural patterns - on system level. Their second book contains patterns for more specialised purposes, namely concurrency, synchronisation, networks and related issues. This development between the books is similar to that of the pattern community. The focus on general patterns has been set aside by the focus on specialised patterns. The specialisation can be on different programming languages, as in: [ABW98], [Bec97], [Gra98], or on different domains, as in: [Lea99], [MM97], [NWB00]. The vertical extension of the field of patterns with the specialised patterns is augmented with a horizontal extension of patterns with different pattern types, e.g. the architectural patterns in [SSRB00], the analysis patterns in [Fow97] and the anti-patterns in [BMB+98].

Over the years the pattern concept became increasingly popular and a large number of papers and books containing patterns had been published. At the OOPSLA conference in 1998, the hot pattern subject was how to classify or organise patterns to make the field less confusing, as described e.g. in [AC98]. This was the main subject in the discussion seminar called “A bowl of patterns”, where it became known that Linda Rising was working on a handbook for patterns [Ris98], which could make it easier to find the right pattern for the

task. A few years later, she followed it up with a pattern almanac [Ris00]. These books have proved to be a great help in finding the right pattern for the task, and thereby extending developer's knowledge of patterns beyond the GoF-book.

Chapter 3

Philosophy of Patterns

It is a fact that some developers had discovered and described recurring patterns of good solutions in their code before the work of Christopher Alexander became known. Based on this it could be argued that patterns in software were not initiated by Alexander's ideas after all.

It remains true though, that the philosophy behind Alexander's pattern language is the basis for many software patterns. Some authors relate their work to Alexander's ideas more than others, but Alexanders philosophy is important in order to understand how the concept of patterns and pattern languages can revolutionise software engineering.

> *Let us start by seeing how the great cathedrals, Chartres and Notre Dame, were made within a pattern language... There were hundreds of people, each making his part within the whole, often for generations. At any given moment there was usually one master builder, who directed the overall layout... but each person in the whole had, in his mind, the same overall language. Each person executed each detail in the same general way, but with minor differences. The master builder did not need to force the design of the details down the builders' throats, because the builders themselves knew enough of the shared pattern language to make the details correctly, with their own individual flair... We imagine, because of the distorted view of architecture we have learnt, that some great architect created these buildings, with a few marks of the pencil, worked out laboriously at the drawing board. The fact is that Chartres, no less than the simple farmhouse, was built by a group of men, acting within a common pattern language, deeply steeped in it of course. It was not made by "design" at the drawing board.*
>
> — "The Timeless Way of Building" by Christopher Alexander, p. 216

Alexanders belief is that a mass of people all aware of the same pattern language will together be able to build masterpieces. This is part of the philosophy behind his pattern language, which will be described later in the chapter, and this belief is dealing with a phenomenon known as swarm intelligence.

3.1 Swarm Intelligence

Complex behaviour by a group of individuals need only require that each individual follow simple rules, and the collective behaviour of the group can display facets and filigrees nowhere apparent in those rules. This behaviour is known as swarm intelligence and builds on the synergy found in the swarms of e.g. insects or birds.
Termites are blind, their environmental clues are mainly smell, moisture gradients, temperature and vibration. Some species of termites that live in the tropics build extraordinary nests. By mixing grains of dirt with termite spit, the termites construct huge mounds or towers as much as thirty feet high with many tunnels and rooms occupied by millions of insects. Looking inside them, the arches and tunnels look as if they were drawn by an architect. How is this when we know, that the insects are not only unintelligent, but also blind? The answer is to be found in the accumulated work produced by multitudes of termites following simple patterns. Each termite builds according to a very simple set of principles:

- Find dirt.
- Mix with saliva.
- Place where the smell is strongest.

In the beginning the building will look like (and actually be) pieces of saliva-soaked dirt distributed randomly over a surface. After a while though, by chance, some pieces of dirt will be placed right next to each other. With two pieces so close to each other, the smell will be stronger and the termites will place their pieces next to or on top of these two. After a while a tower of dirt will be raised and there is a high probability of another tower in the vicinity. When the two towers have reached a certain height, they will touch each other, since the termites will be confused by the two smelly points and place their piece on top of one, but offset slightly in the direction of the other.

The termites are not intelligent in their own right, but the colony is intelligent. There is synergy, the amount of intelligence of the society of termites is larger than the sum of the individuals' intelligence.

It is a peculiar thought that man in himself is intelligent (at least to some extent). Yet the "colony" of men living on earth has not created beautiful things in general, has not made the earth as a whole a better place and does not move and work together with a common target. It could lead to the belief that swarm intelligence only works within societies of unintelligent beings, and that the presence of intelligence destroy the possibility.

3.2 Alexanders' Paradigms

The reason for looking at swarm intelligence in relation to patterns comes from Christopher Alexander, who was convinced that there are certain rules that

everything must follow in order to have the *quality without a name*. His book "The Nature of Order" elaborates on this idea giving examples and descriptions of the patterns that can be found in everything. The fact that "the right solutions" recur gives rise to the thought of patterns.

Christopher Alexander proposed a paradigm for architecture based on three concepts: the quality, the gate, and the way:

- The Quality

  (a.k.a. "the Quality Without a Name") This is the essence of all things living and useful that imparts to them qualities such as: freedom, wholeness, completeness, comfort, harmony, habitability, durability, openness, resilience, variability, and adaptability. It is what makes us feel "alive" and "sated", gives us satisfaction, and ultimately improves the human condition. It is very difficult to explain *the quality* with the concepts at hand. It has been described as an objective beauty, as it can be found in oriental carpets.

- The Gate

  This is the mechanism that allows us to reach the quality. It is manifested as a living common pattern language that permits us to create multiform designs which fulfil multifaceted needs. It is the universal "ether" of patterns and their relationships that permeate a given domain. The gate is the conduit to the quality.

- The Way

  (a.k.a. "the Timeless Way") Using the way, patterns from the gate are applied using a technique of differentiating space in an ordered sequence of piecemeal growth, progressively evolving an initial architecture, which then flourishes into a "live" design possessing the quality. Alexander likens it to "a process of unfolding, like the evolution of an embryo, in which the whole precedes its parts, and actually gives birth to them, by splitting." ([TTWoB] p. 365).

By following the way, one may pass through the gate to reach the quality.

3.3 Patterns and Eastern Philosophy

After the first hype around the software patterns, an increasing number of people from the pattern community have started studying patterns in depth, they have started looking at Alexanders ideas in relation to the patterns they use and how they use them. As can be seen from various mail discussions and news groups, many relate pattern philosophy to eastern philosophy, from which one can quote expressions like: "Alexander is Zen-like from day one." and "Why should pattern study "ending up in Taoism and Zen" even remotely come as a surprise?".

The discussions focus on the definition of patterns in a broad sense, i.e. not only in architecture and software.

Most participants in the discussions take "I Ching" [Hua98] as the starting point and relate what Alexander chooses to call *forces* versus *solution* to what I Ching authors call *receptive* versus *creative*. According to Alexander, patterns are something we accept, not create. They form the underlying universal patterns the Chinese philosophers called "Tao".

It is argued that pattern study is a logically based, holistic science new in the West, but with a long history of refined application in the East: Pattern study, as holistic science, adds descriptive logical synthesis to a traditionally generative mathematical analysis, that has been our exclusive scientific philosophy for about 400 years.

Where traditional mechanical science studies mathematical sequences as quantifiable transformations, pattern study adds a new category to science by using descriptive logic to refine the quality of the whole, such as one's life. Quantitative math versus qualitative logic is *yin* versus *yang* of Eastern reasoning. This fire and water dualism appears in every philosophy, East or West.

Even though the philosophy behind Alexanders patterns is said to be Eastern in flavour, there are plenty of Western philosophers that can lead one to the same levels of understanding wholeness. Kant calls the universal pattern "Transcendental Reasoning": reasoning that transcends experience our intellect uses patterns prior to learning how or why to use them. In other words, and according to Kant, "transcendental reasoning" is an inherent universal aspect, he calls it "a priori", of a reasoning being. This puts Kant in complete agreement with Chinese philosophy that long argued that 'Nature' versus 'Tao' finds in Tao a source of universal patterns that intellectually guide nature into systematic living structures.

The above describes the essence of the ongoing mail- and news-discussions on pattern philosophy. I find the thoughts interesting and important as an aspect of patterns, but they are not the focus of my interest. The reason that they appear in this thesis is that the philosophy behind patterns is an important aspect of patterns, especially in the understanding of the concept of patterns.

3.4 The Paradigm in Software

Alexander's paradigm is the basis of the philosophy behind his patterns and must be translated to software in order to evaluate to what extent his philosophy has been captured by the software patterns.

- The Quality

  What kinds of code, design, or systems improve the human condition? What quality do we find in design and code, that is worth striving for? Current software patterns focus on flexibility and maintainability. These qualities make it easier to change systems in order to reflect changing (interpretations of) requirements, but it is difficult to see how the patterns improve human conditions. If the humans in focus are the users of the system it is a far fetched claim to say, that the flexibility and

maintainability of their software systems improve their lives. The only change the patterns make is that they make it easier and in turn cheaper to change the systems to live up to the users needs. Some pattern people describe patterns for how to communicate with customers [Ris97] and how to build an organisation [Cop95a] and these patterns can improve human conditions, but are they software patterns? In my point of view, they are periphial software patterns. They improve the human relations and conditions of the software developers, without actually touching the software. The actual software patterns, most likely, improve only the lives of the developers.

- The Gate

  The living common pattern language for software development has only just started to form. If we choose to see the software patterns of today as posessing the quality, then we are well on our way. There exists a huge number of pattern already, that together comprise at least part of the pattern language for software development. Christopher designed a pattern language, that was descriptive enough to communicate the exact idea, yet simple enough for everybody to understand. Wolfgang Pree proposed his meta patterns as the counterpart for this in software. In the paper: "Meta Patterns: A Means For Capturing the Essentials of Reusable Object-Oriented Design" [Pre94], he describes the seven meta patterns. It cannot be denied, that meta patterns are simple, and that most software design can be described by them, but they are not descriptive enough to be candidate patterns in a pattern language for software. For further discussion see [AC97b].

- The Way

  The way to apply patterns in order to reach the quality. A global awareness of the patterns is needed to obtain the synergy within the collected software development. According to Alexander, patterns are to be generative to be able to give the wanted result. With generative patterns a whole pattern language is meant. Pattern languages deal with rich interactions between patterns to indirectly generate solutions for more fundamental problems, much larger than any one pattern can address by itself. Even though some pattern categories provide a set of related patterns for each pattern, these related patterns are mere variations of the pattern or solve a problem arisen by use of the pattern, and are not a form of generativity between the patterns.

3.5 Alexanders' Reply

Christopher Alexander held a keynote at the OOPSLA conference in 1996. He had been invited to give a talk by the pattern community, led in this respect by Jim Coplien. Luckily the talk was recorded. He took the opportunity to evaluate his own success with patterns and the pattern work done by software

developers. In short, Alexander said that we, as far as he could see, had misunderstood his pattern movement. The argument was that our patterns did not improve human conditions in any way, but instead were mere hacks in our design. He described his own pattern movement as being a failure, because so many buildings and towns were already built and it would be too expensive to change it all. The other reason for the failure was that only very few architects accept Alexander's patterns as the true way of building, while the rest wanted to show off and make money. This is sad, he said, but there is hope for us in the software business. He believes that there is a possibility for us to use pattern languages to reach the quality, if we start thinking now. As he said, we have the world in front of us, and since software development is in its infancy, it is actually possible for us to change things for the better, before it is to late, as it is with architecture.

Joe Bergin, a professor of Computer Science at Pace University, is an eager member of the pattern community and one of those who take the time to stop and think. He claims that there are two extreme ways of looking at patterns; as a holy script and as a mere literary form. In between these two lies his point of view: That the patterns should follow Alexander's principles to make life better for people. Bergin does not believe that the current use of patterns makes the software architects' life any better, neither that of the end users of the systems, since they cannot use them to decide for themselves how to build the systems.

In building architecture only one pattern language is needed since this pattern language is built on human nature. The architects design not only the walls, but also, and in a sense more so, the space between the walls. It is still an open question whether this applies to computer science in Joe Bergins opinion.

3.6 Discussion

My claim is that the software patterns actually do make life better for developers or software architects. Good design solutions can also be said to induce elegant code, that is easily readable and comfortable to work with. If code is elegant it can be argued that it improves the human condition of the developers and thereby possess the quality without a name. The developers are the ones who step into the code like people step into a house. When you are designing or coding you walk into the design model or the program and you can feel comfortable or not. If the design or code has the quality without a name you feel better than if it does not. When you design your own house you design for change, because you want a house that can grow with you, in the sense that you might need more rooms for children along the way or you might want to turn it into something that old people can live in (e.g. fewer rooms, no stairs etc.) Similarly if you are developing code, since you are able to foresee other changes while some changes are unforeseeable, you want to make code which is comfortable to be in right now but which can also be easily adapted to other requirements.

Chapter 4

Object-oriented System Development

This chapter contains an overview of the processes, methods, techniques and technologies that constitutes my view of object-oriented system development. The descriptions are of necessity not exhaustive. In Chapter 5 we elucidate the role of patterns in systems development and the discussion there is built on the definitions described in this chapter.

4.1 Processes

Processes are the **what** and **when** of system development. To quote from [Boe88]:

> ”... a process model addresses the following software project questions: 1. What shall we do next? 2. How long shall we continue to do it?.”

This section contains descriptions of two processes; The Unified Process and the extreme programming process.

4.1.1 RUP

The “Unified Process” also known as the “Rational Unified Process” or simply RUP, is described in [Kru00] and gives a full-fledged description of a development process. The process has four phases:

- Inception. The preparation of the project, where the business issues are dealt with and the extent of the project is defined.
- Elaboration. The establishment of the project where the problem domain is analysed, and the architecture is chosen, a project plan is developed and the greatest risks are eliminated.
- Construction. The building phase, where the project takes shape and a version is finished.
- Transition. The final phase, where the system is tested and delivered to the end customer.

Within each phase there can be a number of iterations, depending on the size of the project. As a rule of thumb the inception phase can contain at most one iteration and none if the project is sufficiently small. The number of iterations in the elaboration phase depends on whether there is an architecture or not, because it affects how much there is to be determined in that phase. During the construction phase at least one iteration is needed and the number of iterations grows with the size of the project. In the transition phase the number of iterations depends on whether errors are detected in the release.

For each iteration a result is expected. In some phases it is easy to know when to stop an iteration, because what is needed is a prototype with a well-defined functionality, but for the inception phase and the elaboration phase, there is the danger of prolonging the preparation of the project, see also a description of the anti patterns **Death by Planning** and **Analysis Paralysis** in Section 6.4.

As notation, UML is always used in the Unified Process. Throughout the project UML is used to communicate analysis, design and implementation issues with customers, managers and other developers.

As illustrated in Figure 4.1 the Unified Process is use-case driven. The use-case model is the common denominator of the different development models. The use-cases describe the functionality of the system and the knowledge they contain can be used in the planning of the iterations, the analysis, the design, the implementation, the verification and the user manuals.

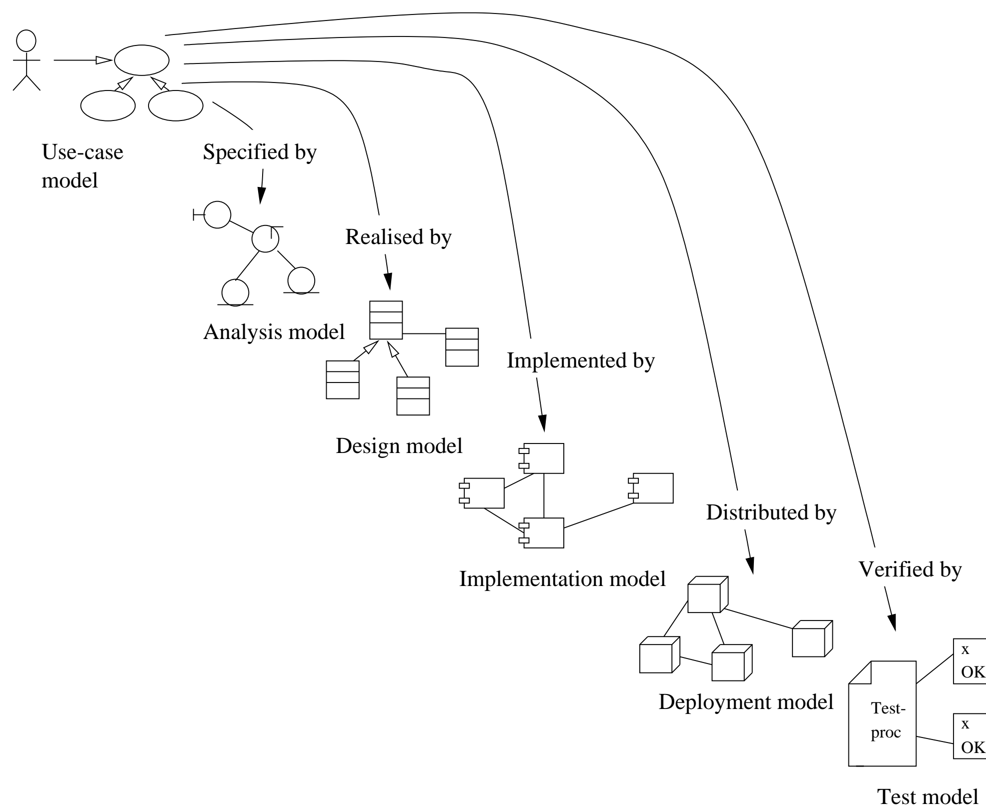


Figure 4.1: The use case is the common thread between the different models.

To minimise the risks, the prototypes are an important part of RUP. The prototypes are not part of the resulting system. They are concrete visual aspects of the system, that are easier for the customers to relate to.

The architect is a central figure in RUP. The role can be played by one or more, but they must work as a unit to supervise and coordinate the work. The architect creates the structure of the system as a whole and has an overview of the width of the system, without knowledge of the depth.

4.1.2 eXtreme Programming

As mentioned in Chapter 2, Kent Beck is one of the pioneers of the application of the pattern concept to software. His latest public idea is that of eXtreme Programming [Bec99].

Listening, Testing, Coding, Designing. That's all there is to software. Anyone who tells you different is selling something.
— Kent Beck

1. Listening. It is easy to write code that does something. It is a bit harder to make it do exactly what the clients, users, and managers want, unless you understand their needs perfectly. To this you have to listen, listen and listen to them.

2. Testing. If your program does not work, or it works in a way the customer did not expect, the system is useless. If you're smart, you'll write tests of your code before you write the code. In that way you will know the instant you're done, and do not have to waste time trying to reach a goal you have not clearly defined.

3. Coding. This is important, since the matter we are delivering to the customer in the end is code in some sort of wrapping. We can have many good intentions, but if we do not have code, we do not have anything.

4. Designing. You have to take what your program tells you about how it wants to be structured and feed it back into the program. This item is a bit tricky and cannot be understood until one learns a bit more of XP.

eXtreme Programming is a so-called lightweight process, meaning that the description of the process consists of guidelines and best practices, not a thorough description of all details in the process as it is often seen. Let us just go over a few of the most interesting items.

Coding

The coding is done as pair programming. In pair programming two developers are sitting next to each other sharing table and computer. The programmers alternate between two roles, the one actually using the keyboard (also called the driver) and the one observing or giving suggestions. This approach to programming makes it easier for novices to be integrated in a project, it gives

the experienced programmers a chance to spread their knowledge and it makes errors less frequent.

Testing

Another important part of the process is that of unit testing. All code ideally has a test program to live up to, when it is made. The tests are made first, and after that, the code is implemented to live up to it. One of the reasons for this is that it is the only way to maintain a system of common code. In XP all code is everybody's responsibility, but instead of getting code that is nobody's responsibility, it is a rule that all developers have access to the code and are able to make changes in the code, as long as it still lives up to the unit tests created for it. Constant unit testing is also a condition for small releases. Unlike traditional systems development processes, where a system is presented to the customers when it is “finished” according to the requirements or prototypes, XP takes the approach of small releases, which means that they always have a running system. The system, in the beginning being very slim and rough, is presented to the users at a very early point in the development process. Every two weeks or so the customers are presented with its current functionality and are at any point in time able to stop the project. If they choose to buy the system as is, it will be sold at the current price and with the current functionality. This way the users will not feel abandoned by the developers and the developers always know that they are working in the right direction.

In a paper describing experience obtained through a joint project between a research group at the University of Aarhus and a shipping company in developing a prototype for a global customer service system, the following quote is found:

> ”It should also be said that a major development requirement from our point of view is to have a running prototype even of minimal design to confront practice with and, in reaction, thereby elaborate modelling issues.”

Their paper is called “The M.A.D. experience” [CCD+98] and refers to the very evolutionary way the development was executed.

Designing

During the development of the system, there is no “master plan” of the system. The developers must always code the simplest possible to live up to a test, they must not try to guess in what way the system requirements might change to try and make it flexible in the right parts. Instead the code will grow and when it reaches a certain point, it will be beneficial to re-factor the code to place common properties in a common superclass and such. (See refactoring [Fow99] by Martin Fowler) This approach to coding can be compared to swarm theory, since it is the developers who together create the whole picture, step by step, not a superior architect who dictates what to do. With the common knowledge of

how to code and design well, the developers intuitively make the right decisions. The common knowledge consists of naming conventions,

In the MAD experience refactoring played a great part too, since an early success with refactoring the system led the development team to explicitly plan for refactoring of the system constantly throughout the project. Based on their experience the developers conclude in their paper: "Software Architectural Evolution in the Dragon Project" [CDH+99] that it rarely is possible to design the final architecture of the system during the initial phase of the development. Instead, they argue that architectural evolution is necessary. One could argue that it is an example of an XP project before XP was on the shelves.

4.2 Methods

When defining what a method is in relation to a process, one has to look at details. In *Object-Oriented Analysis and Design* [MMMNS97] the role of methods in system development is described: *System development is more a craft than a science. A good developer is good because of his personal properties gained through practical experience more than because of his methodical knowledge from theoretical studies. Methods play an increasing role in the development of the subject. They work both as a means of flattening the learning curve for beginners and for communication between skilled developers.* The authors describe a method as being a collection of general guidelines for the execution of an activity. It is possible that the current word for method simply is 'process'. There are some minor differences, though. Firstly processes are very concerned with how much time is spent on which activity, in what order things are done and with how many iterations. Those are considerations most method authors leave to the reader. Secondly the methods describe in detail how to deal with the different phases, how to assign responsibilities, how the communication, which are aspects left to the readers of process descriptions. While processes are the **what** and **when** of system development, methods are the **how**.

Different methods of system development coexist in modern system development. Even though time has changed the development researchers view, developers are still using the method they find most useful in their domain, independent on what the state-of-the-art between researchers is.

4.2.1 The OOA and D method

In their book "Object-oriented analysis and design" [MMMNS97], Lars Mathiassen et al. give a representation/exposition of the central principles of object-oriented software development. The book provides a method for system development which, in their own words, is a collected set of principles for the entire development process. The method search to inspire people to reuse code through the object and component concepts. Additionally it describes ways to reuse design ideas through the application of role models and patterns. See more details on this in Section 5.2.

To describe the object oriented analysis and design method, the authors need to define the problem domain.

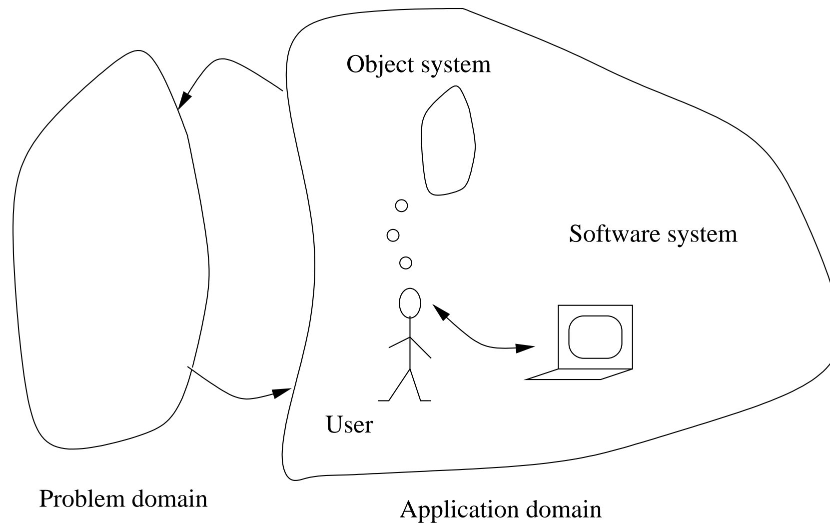


Figure 4.2: The object system shows a future user's perception of the software systems problem domain.

Figure 4.2 shows the two sides of a software systems environment: The problem domain and the application domain. The problem domain can be defined to be that part of the environment, that is to be administrated, surveyed or controlled via a software system. The future users perception of the problem domain is called the object system. It is important that the object system that is described respects the future user's current perception of the problem domain. The developer's task is to understand the business model of the users, but also to create a future system with enhancements of the working process.

The application domain is an organisation, that administrates, surveys or controls a problem domain. This domain is part of the user organisation, it is the bridge between the problem domain and the application domain that makes the software system.

The main activities of the OOA and D method describes four central perspectives on a system. The perspectives focus on the information contained in the system, the use of the system, the system as a whole and the parts of the system. These perspectives are bound to the four main activities: Analysis of the problem domain, analysis of the application domain, design of architecture and design of components. These four main activities are shown in Figure 4.3.

Analysis of the problem domain.

In the analysis of the problem domain lies three part activities, describing classes of objects, finding their internal relations and describing the dynamic properties of the objects. The emphasis is on demarcating the object system and describing its dynamic properties.

Analysis of the application domain.

Since the application domain is how the system is to be used, the customers and users are more interested in the analysis of the application domain than that of the problem domain, but it is important to start the analysis of the problem domain first, since it is easier to analyse the application domain using

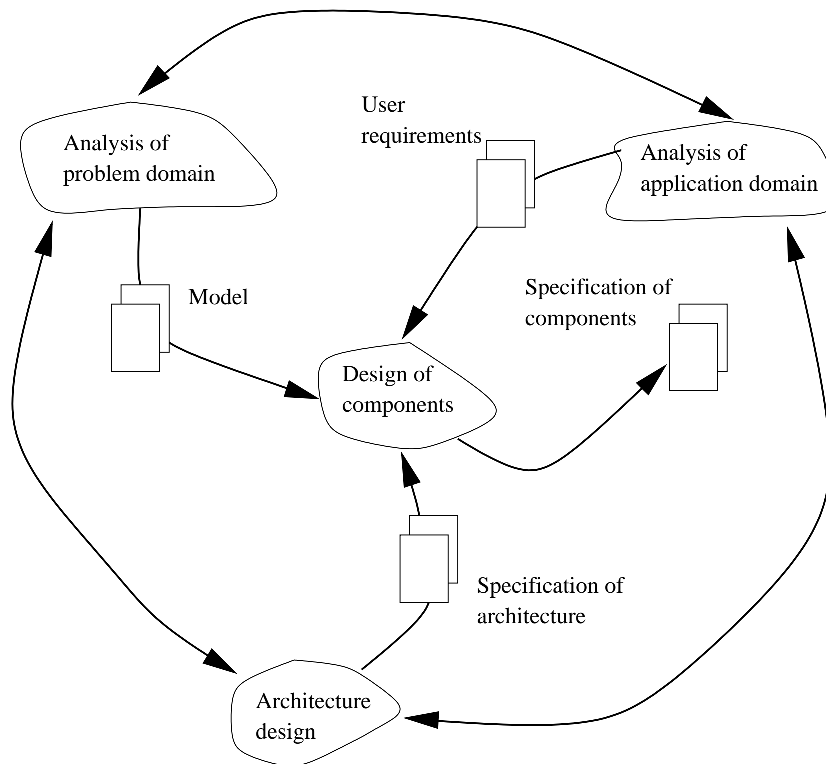


Figure 4.3: The Main Activities

concepts from the object system shown in Figure 4.2 . The analysis of the problem domain can also give reason to evaluate and revise the object system.

Design of architecture.

This activity holds the requirements against the possibilities and finds the best match. The three part activities contained in this are: Generating a prioritised list of requirements, developing the structure of the classes of the system into components and defining the dynamic relationships between the objects in the system.

Design of components.

The part activities of this activity build on the architecture of the system. The architecture of the system will describe the different subsystems that will constitute the system. These will be designed as components communicating via interfaces and so this activity is about designing these different interfaces.

4.2.2 OODA

The book: "Object Oriented Design with Applications" by Grady Booch offers another object-oriented design method.

The method is iterative and incremental with four primary steps taken serially again and again, until the right level of abstraction is reached. The cycle is illustrated in Figure 4.4.

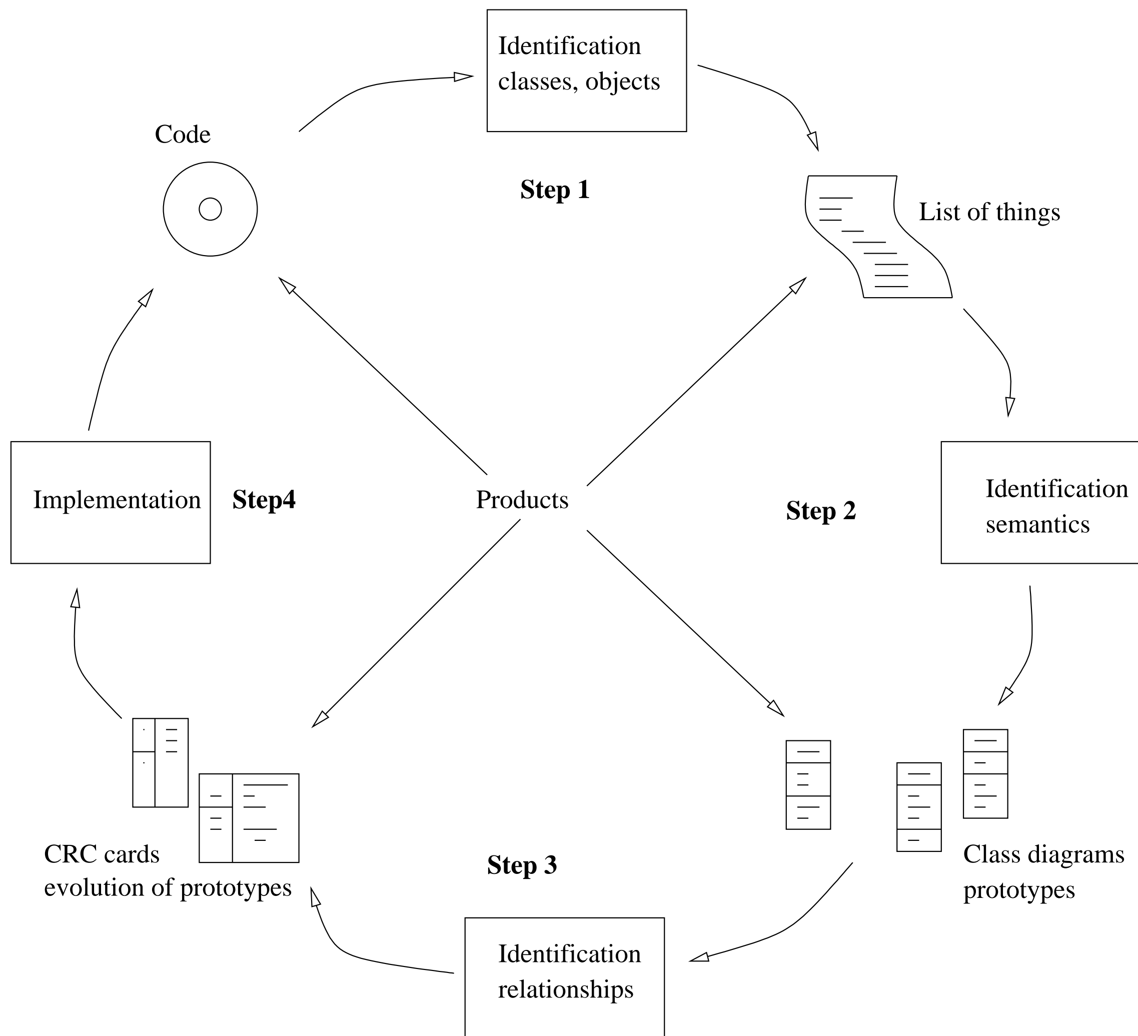


Figure 4.4: The four steps in Booch's method

The first step

Involves the identification of the classes and objects at a given level of abstraction; here, the important activities are the discovery of key abstractions and the invention of important mechanisms.

The products of this step may range from very informal to very formal. It can be an open-ended "list of things" consisting of meaningful names of significant classes and objects. Some of the things on this list may turn out to be classes, some objects, and others simply attributes of objects. It can also be a formal specification of the abstractions and mechanisms discovered. The main goal is to create a common vocabulary and this step should therefore consume a relatively small amount of time.

The second step

Involves the identification of the semantics of these classes and objects; the important activity here is for the developer to act as a detached outsider, viewing each class from the perspective of its interface.

The product of this step is the refinement of the templates drafted in the first step. For each abstraction and mechanism, the static and dynamic semantics should be documented as best it can be done at that point in time. It can be beneficial to prototype parts of the design at this stage to analyse the current design and evaluate alternate approaches to subproblems.

The third step

Involves the identification of the relationships between these classes and objects; here, we establish how things interact within the system, with regard to the static as well as the dynamic semantics of the key abstractions and important mechanisms.

The products of this step include the completion of most of the logical models of the design. The development of the prototypes are continued, older prototypes are refined and new ones created.

The fourth step

Involves the implementation of these classes and objects; the important activities here involve choosing a representation for each class and object, and allocating classes and objects to modules, and programs to processes; this step is not necessarily the last step, for its completion usually requires that we repeat the entire process, this time at a lower level of abstraction.

The products of this step are a refinement of the class structure of the system and the completion of the implementation part of each important class template.

4.3 Techniques

Techniques for describing and communicating ideas, suggestions and designs are important for system development processes. Some developers choose to take the standpoint that code is self-explanatory, but the majority of developers admit the need for supplementary communication of some sort. This section describes an early technique and the (for the moment) generally accepted technique for communication.

4.3.1 Class-Responsibility-Collaboration card (CRC card)

In 1989 Kent Beck and Ward Cunningham introduced CRC cards [BC89] to help identify and specify objects or components[1] in an informal way. They had

[1] the word *class* in class-responsibility-collaboration is there for historical reasons

found the cards useful for teaching novice programmers the concepts of objects, and for introducing experienced programmers to complicated existing designs.

The CRC card can help give an understanding of objects as self-contained autonomous elements, that can otherwise be hard to obtain. As the authors state it: "The most difficult problem in teaching object oriented programming is getting the learner to give up the global knowledge of control that is possible with procedural programs, and rely on the local knowledge of objects to accomplish their tasks. Novice designs are littered with regressions to global thinking: gratuitous global variables, unnecessary pointers, and inappropriate reliance on the implementation of other objects." With the cards shown in Figure 4.5, the responsibilities and collaborations of each object or element can be visualised. Since there is no strict form to use when describing the elements, CRC cards can be used at a very early stage in the design.

Figure 4.5: Class-Responsibility-Collaboration card

In addition to the text on the card the placement of the cards can be informative. If cards overlap, it means that there is a strong collaboration, if they are placed in a hierarchy, this can show the inheritance hierarchy, and if they are placed in small groups, they have strong intercommunication within the group. It is also possible for the developers to pick up the cards and "play the objects". Normally the cards will only be used in the first phase of the development, but they can be used at all phases to illustrate a specific part of the system's behaviour.

4.3.2 UML

In the late eighties, different object-oriented methods and notations emerged. For every aspect of system development, a notation had been developed to let the designers communicate better. In 1995, Grady Booch, Ivar Jacobsen and Jim Rumbaugh, also called the Three Amigos, joined their previous work and created a unified modelling language. They called it UML, and it has grown to be a standard within object oriented software development. The goal was to make a standard modelling language, regardless of the choice of process used in the development. The result is a single, common, and widely usable modelling

language for developers.

> *The UML specifies a modelling language that incorporates the object-oriented community's consensus on core modelling concepts. It allows deviations to be expressed in terms of its extension mechanisms*
> — from UML Summary, Version 1.3

A common goal for UML models is to make them simple, yet informative. Thus the models should have the needed richness of details and nothing more, they need not be exhaustive. If more details are needed another, more detailed model can be drawn in addition to the general model.

Some of the more well-known structures of the UML notation are shown in Figure 4.6. Here follows an explanation of the sub-figures A-H:

A. A *use-case* model consists of related use-cases and actors. A use-case describes a sequence of actions a system executes, resulting in an observable effect for the actor involved. An actor is usually a user of the system, but can also be external devices, such as printers or databases. The use-case structures the functionality of one or more activities in a model, and is often realised with a *collaboration*.

B. An *analysis model* consists of objects and their relations. An analysis object can be one of three kinds; an interface object, a control object and an entity object. These objects and their relations in the model represent the result of the analysis of the problem domain. The analysis can consist of both a description of the existing business model and a description of the model of work. The *use-case* models and the *analysis models* are used together to create a basis for the development of a *design model*.

C. A *collaboration* defines an interaction and a collection of elements, working together to realise a behaviour bigger than the sum of the single elements contained in it. A collaboration can therefore be used to model design patterns and other sub-designs in the system. Since it describes both the interaction and the collection of elements, it has both a structural and a behavioural dimension. The structural dimension is described with class-, and object-diagrams and the behavioural dimension with different kinds of interaction diagrams.

D. A *sequence diagram* is one of the interaction diagrams in UML. It models how instances collaborate to do tasks. The squares at the top contain the type of the object and possibly the name of the object itself, if that is significant to model the given interaction. The dashed lines beneath the objects model the lifetime of the objects and the boxes on top of them are activities in the objects. These activities can be initiated by a call from another object, visualised by an arrow.

E. A *design model* can be either a class diagram showing the compile-time view of the system (or part of the system) or an object diagram showing the run-time view. A class diagram consists of class descriptions and

their relations. The descriptions of the classes can include attributes, level of encapsulation and method descriptions. As with any other kind of model, it should be kept simple. The relations between the classes can be inheritance, implementation of interface, associations, aggregations, etc.

F. An *implementation diagram* consists of components and their cooperation. A component is a physical part of the system, which conforms to and realises a number of interfaces, and can thus be replaced with another component conforming to and realising the same interfaces. In the model the component represents a physical packing of logical elements like classes, interfaces and collaborations.

G. A *deployment model* consists of nodes and connections between them. A node is a physical element existing at runtime, like a server or a laptop. The connections between the nodes illustrate the means of communication between them, like TCP/IP and DecNet.

H. A test model describes the test cases needed to verify that the system lives up to the user requirements. The tests can be unit tests made from the class and object diagrams, integration tests made from the interaction diagrams and system tests made from the use-cases.

4.4 Technologies

This section presents technologies supporting the development of software systems. A number of technologies have emerged to support object orientation. First and foremost of course are the programming languages giving us the raw material with which we can build systems. The next generation of technologies are programming environments making it easier, safer, and sometimes more fun to develop systems.

4.4.1 Programming Languages

Naturally, programming languages are an important means to create software. The object-oriented designs made using the above methods and techniques can be implemented in any kind of programming language, but an object-oriented programming language is the natural choice, due to their direct support for objects, classes, inheritance, etc. The field of object-oriented programming languages is vast and diverging. It all started with Simula [DN66], a language for simulation developed in the sixties by Ole Johan Dahl and Kristen Nygaard. Simula has many descendants. One of these is Smalltalk which became very popular with its use of classes and messages. [2] Another descendant is BETA [MBMP93], a not so well-known programming language with a very strong language construct, called a *pattern*, that unifies classes, types, procedures and

[2] Actually, everything in Smalltalk is an object, even the classes, but this is not something we will enter here...

records. Object-oriented languages became popular with Smalltalk, and have stayed popular for more than 20 years now with a number of new languages being developed as result. C++ is a recent and popular example. C++ is built upon C, a procedural language, and inherited many features from it, among them its efficiency and lack of garbage collection. From C++ a portable garbage collected language emerged of the name Java. Java was born with a silver commercial in its mouth and so, as a consequence, also rose to the heights of fame. Java has included some design patterns in its standard class libraries to support reusable design. It could be argued that several languages have design patterns incoorperated, but since Java is a language young enough to actually refer to the design pattern it uses, it is the most evident case.

4.4.2 Tools

A relevant place to search for object-oriented development tools is the CETUS site [cet], where the motivation for tools is explained by Fred Hebbel as elegantly as this:

"CASE tools offer many benefits for developers building large-scale systems. As spiralling user requirements continue to drive system complexity to new levels, the CASE tools enable us to abstract away from the entanglement of source code, to a level where architecture and design become more apparent and easier to understand and modify. The larger a project, the more important it is to use CASE technology. As developers interact with portions of a system designed by their colleagues, they must quickly seek a subset of classes and methods and assimilate an understanding of how to interface with them. In a similar sense, management must be able, in a timely fashion and from a high level, to look at a representation of a design and understand what's going on. For these reasons, CASE tools coupled with methodologies give us a way of representing systems too complex to comprehend in their underlying source code or schema-based form."

The first tools developed were rough in their support of developing systems, but two lines of development had an effect on them:

Firstly, the research in software development processes gave rise to a number of software design methods. The nature of these methods was ideal for automated support, since they required a number of different development environments with a graphical notation and the need for storage of diagrams, documentation, and such.

Secondly, the hardware support in personal computers, which offered relatively large memory storage capacities, fast processors, and sophisticated bit-mapped graphics displays that were capable of displaying charts, graphical models, and diagrams.

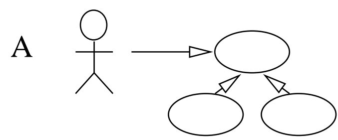
In order to ease the discussion of different types of tools, a classification of tools is useful:

The *general* CASE tools are the tools that are interactive in nature (such as a design method support tool), in contrast to those that are not (like a compiler).

Front-end CASE tools are tools that support activities early in the life cycle of a software project, such as requirements and design support tools. *Back-end* CASE tools are tools that are used later in the life cycle, such as compilers and test support tools. *Vertical* CASE tools are tools that are specific to a particular life-cycle step or domain, such as a requirements tool or a coding tool. *Horizontal* CASE tools are tools that are common across a number of life-cycle steps or domains, such as a documentation tool or a configuration management tool.

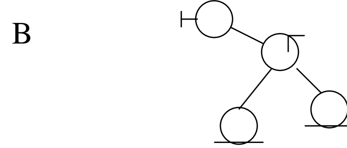
The tool, $\mathcal{DPDOC}$, described in Chapter 9 is an example of a vertical CASE tool, since it works on code-level. It is also a tool for documentation, but it does not, as a horizontal tool would, document all the way from user requirements to code. It could be argued, that it is a back-end tool, since it can be applied to the code just before compilation, but since it can also be applied along the way, it belongs in the group of neither front-end nor back-end tools.

A



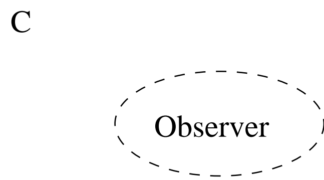
Use-case model

B



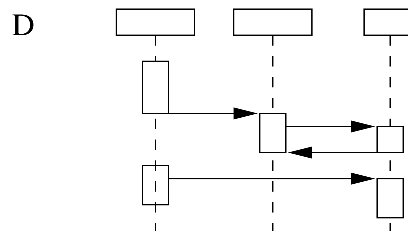
Analysis model

C



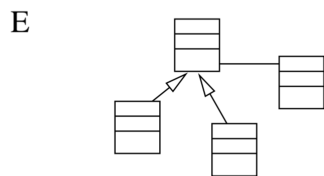
Collaboration

D



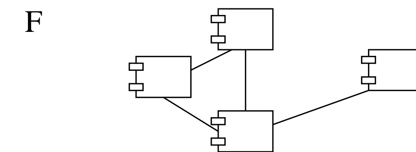
Sequence diagram

E



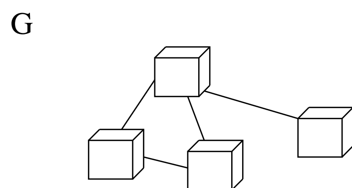
Design model

F



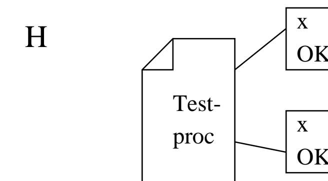
Implementation model

G



Deployment model

H



Test model

Figure 4.6: A taste of UML

Chapter 5

Patterns

An important question is how patterns, or the application of patterns, fit into object-oriented system development. This chapter contains a discussion and a conclusion on this.

5.1 Comparing patterns with existing concepts

The question is whether patterns belong to one of the four concepts; processes, methods, techniques or technologies.

Patterns as a process

As discussed in Chapter 3, one philosophy behind patterns is that they should be known by all developers, and the that the "swarm" of developers thereby becomes more intelligent. The logical consequence of this would be that the philosophy behind the application of patterns does in fact resemble a process: A very light weight process only giving the one guideline, that every developer should know the patterns. This viewpoint is neither supported, nor contradicted in pattern literature.

There are similarities and differences between the two earlier mentioned processes and the philosophy behind patterns, but not enough homogenity between the level of detail to prove that patterns are a process. Pattern philosophy can be compared, though, to a more loose concept, such as participatory design:

A major part of the justification for computer science is its ability to give results which help people in one way or the other. Some parts of computer science put more emphasis on this subject than others. As the philosophy behind patterns is to make people's environment complete and sound, it will be of interest to look a the other parts of computer science that take the users into account. One philosophy of how to do that can be found in the field of participatory design.

Participatory design is rooted in the idea that understanding the use of technology is essential for informing design. The people who use technology

should actively participate in the design and development of the products and services they use.

Participatory design does not go to the lengths of letting the end users themselves design their own product, but it is on the way to Alexanders ideas, since the users are to be directly and continuously involved in the development.

Patterns as methods

Patterns are seen by some as a descendents of methods since they cover whole solutions, not just small pieces of procedures on how to solve problems.

> *We have realized that the typical methodology books, though valuable, only present the first step in the learning process that must also capture the actual things that are built. This realization has flowered into the patterns movement*
> — Martin Fowler, from the preface in [Fow97].

It could be argued that the concept of patterns brings with it a methodology, but patterns are abstract solutions to often recurring problems. They are not guidelines covering the entire field of problems found within object-oriented system development, they do not describe procedures for how to use them and they can be used within every methodology as more abstract solutions.

Patterns as techniques

When applying patterns in the simplest possible way, it is possible to claim them to be a simple technique. That is, patterns are seen by some developers to be a mere notation for how to describe experience. This is not a widely accepted position and there is a huge chasm between it and the more religious view of patterns.

There is an number of developers standing between them, though. When a developer's point of view is permeated with the importance of getting the most out of patterns, he can apply patterns in the same way as he would use an algorithm or a datastructure, and still believe that the code as a whole wiil be better, i.e. more flexible, more maintainable.

Patterns as technologies

Patterns are an abstract concept and therefore not a technology, but many technologies have built on, or use, patterns. Since design patterns and architectural patterns are the ones most easily compiled to code, they are the only types of patterns, that are in question in a discussion of patterns as technologies. Language support for design patterns is a field of research explored by [Sou95], [TC98] [HrLK00] and others. The basis for this lies in the fact that the pattern solution are well-tested good solutions to often recurring problems and the thought of having direct support for them in the programming language is captivating. Actually implementing design patterns as language constructs has also been discussed and since some design patterns seem to be cover-ups for missing language constructs found in other languages it could be sensible to

incorporate some design patterns in some languages[1]. If the claim in the book "Anti Patterns" [BMB+98] is correct, then today's design patterns could be tomorrow's antipatterns, and in that case we are much better off with strong simple generic languages.

Pattern tools relate to this discussion too, since they incorporate the patterns, not in the programming language but in the programming environment. A thorough description of pattern tools is to be found in Section9.8.

5.1.1 Synthesis

From the above discussions it can be concluded that patterns as a concept are unable to fit into one of the existing development concepts, which leads me to conclude that patterns are a concept of its own. My claim is that patterns, on account of their strong independence of other concepts, are orthogonal to traditional system development and can therefore be used everywhere in almost every way. In the following, I will give an argumentation for this understanding of the role of patterns in system development.

5.2 Patterns in the Development Process

Since the emergence of software patterns, many members of the object-oriented software development community have described how to apply patterns based on their development experience with and without patterns. In the following we will characterise some of these approaches.

5.2.1 UML and Patterns

To give a brief description of how experienced developers propose to use patterns in system development two books are chosen.

Lars Mathiassen et al.

The book "Object-oriented analysis and design" [MMMNS97] describes ways to reuse design ideas through application of *role models* and patterns. Role models are good examples than can be compared to the existing problem and semi copied to provide the solution needed. Patterns are more abstract descriptions of solutions, thereby more generally applicable, but also harder to use.

In developing software systems, the authors pay special attention to three criteria, formulated in the principle: "A useful, flexible and understandable whole". In the description of understandability, patterns play an important role. Simple patterns like certain uses of the programming language and the design language mechanisms are useful to create a homogeneous system. These patterns are also called idioms. They are very small building blocks, but can help create a better understanding and overview. Another kind of patterns, which can be viewed as library classes, provide extensions to the programming language. An example could be a special kind of collection that is to be used

[1]See also a discussion in [AC97b]

a number of times. Giving it a name and placing it where it can be used by different parts of the system makes it a piece of code, that will stand out as a pattern, thereby requiring only one documentation. To describe more general solutions the authors introduce their readers to design patterns, which can provide design solutions for a specific type of problems. Reusing solutions can enhance the understandability, because it is possible to refer to a certain pattern instead of describing the specific solution in detail. The last type of patterns described is the architectural patterns, which give solutions for creating the skeleton of whole systems or subsystems.

Criag Larman

In the book "Applying UML and patterns" [Lar97], the author, Craig Larman, describes what role he gives to patterns in software development. The book leads the reader through all phases of object oriented system development via Craig Larmans process. First one creates interaction diagrams, which illustrate how objects will communicate in order to fulfil the requirements. After this, design class diagrams can be drawn which summarise the definitions of the classes and interfaces that are to be implemented in the software. The creation of interaction diagrams requires the application of principles for assigning responsibilities and the use of design patterns. The book's treatment of patterns bypasses the issue of what can be appropriately labelled a pattern, and focuses on the pragmatic value of using the pattern style as a vehicle for presenting and remembering useful software engineering principles.

The author explains how to apply design and architectural patterns to system development, and emphasises how important they are within framework development. In addition to this he gives the reader nine fundamental responsibility assignment principles, captured in patterns. He finds the assignment of responsibilities to be one of the most important aspects of analysis and design. The patterns as a whole are called the General Responsibility Assignment Software Patterns (GRASP).

5.2.2 The COT Experience

Working with people from the industry through COT, I got to experience to what extent patterns were used in Denmark. Considering the good experiences[2] a number of developers from other countries had had with using patterns [BCC+96] it seemed logical that developers in Denmark could benefit from using the patterns too. Unfortunately, as predicted by Ellen Agerbo and myself in our paper [AC98], they were overwhelmed by the large number of patterns and often discarded them without ever trying to apply them. A reduction of the number of patterns could make things better, but such a classification was not likely to take place overnight. They needed motivation for applying patterns and some guidelines for how and where to apply them. Triggered by

[2]Experience with the application of patterns shows that in addition to making systems, that are more reusable it also makes the development easier and more effective. See also Section 1.3

this, a group from the legacy project of COT wrote an introduction to patterns [COV99] aimed at the Danish industry.

The report is focused on the application of patterns, describing what the concept covers and giving an explanation of how and when patterns can be applied in a development process. The readers of such a report are expected to be software developers who know something about object-oriented software development. The section of that report discussing the role of the patterns in system development is included in the next section in an extended form.

5.3 The Role of Patterns in System Development

The various part activities involved in the system development process and where the different types of patterns can be applied is shown in Figure 5.1.

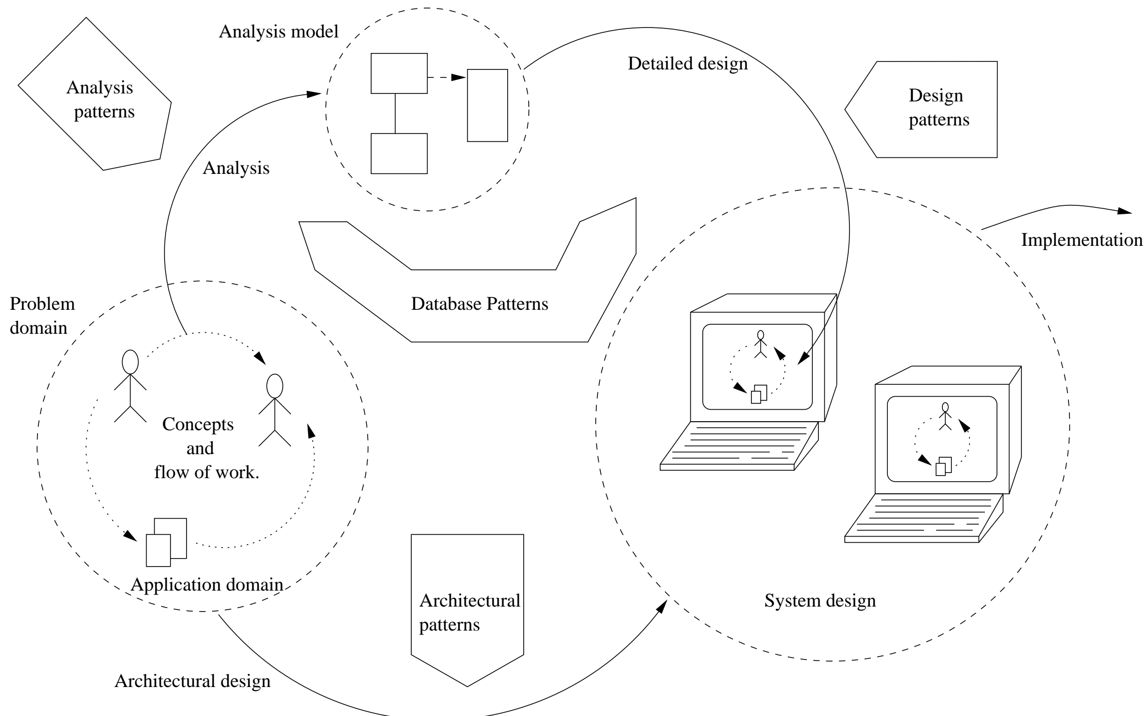


Figure 5.1: The focus of the different patterns in the development process.

The different types of patterns, and some examples of them are included in Chapter 6. The process is partitioned into part activities, each with a certain problem and solution domain. The partitioning into part activities is done merely to show that different types of patterns can be used in different part activities, due to their different focus. We do not want to dictate the chronological order in which people develop their systems.

There are three circles in the figure, each representing a subgoal. The leftmost (Problem domain) represents that part of reality, the system is developed

for. This contains both the concepts and flow of work in the problem domain, and the the external forces in the application domain. The topmost circle shows the role of the analysis model, which describes the identified concepts and the flow of work in the problem domain. The rightmost circle (System design) represents the total design of the system based on the problem domain and the analysis model, in a way that will be described later.

The arrows show the part activities, in which patterns can help development. It is important to remember that it can be necessary to reanalyse the problem domain during design and that the architecture of the system is not necessarily built directly upon the flow of work and the responsibilities found in the problem domain. Thus the direction of the arrows relates solution types to problem types.

The arrow representing the architecture design points from the problem domain to the system design to show that architecture patterns are used to design the overall structure of the system from the problem domain. This analysis is on a macroscopic level, during which the system is partitioned into subsystems and their responsibilities are delegated. The resulting structure is the skeleton of the system. The arrow representing analysis also points from the problem domain, but the analysis is on a more detailed level, where the individual elements are identified and responsibility and communication between them is determined to create the analysis model. This model is an abstract and formal description of the analysis of the problem domain. Based on this, the system can be described by a design model using design patterns as represented by the detailed design arrow. The last arrow in the figure, the arrow representing the implementation is beyond the scope of the report. How a system is implemented depends heavily on the choice of implementation language and system environment.

We use the sharp division of the analysis, design and architectural patterns to be able to describe their different focus in the development process. Some analysis patterns and design patterns are very alike, but their solutions work on two different levels of abstraction, as shown in Figure 5.1. Design patterns and architecture patterns can also be hard to separate. Both kinds of solutions lie within the final system design. The most obvious difference is that, due to the larger size of the elements that comprise the architectural patterns, a change in choice of architecture pattern after implementation is more painful than a change in choice of design pattern. Since database patterns can be applied in all phases of system development (analysis, design, and later optimisation), they are orthogonal to the three types of patterns already described.

The antipatterns are divided into different categories, the programmers', the architects' and the managers' patterns. Naturally the two first categories can be placed between the design, and architecture patterns in the figure, but what about the last category? We did not have the managers view of the system development process in the Danish report and it is hard to imagine a graphical 2D description of these patterns together with the different phases already described in the figure.

The organisational patterns have the same problem as the manager antipatterns. They are encountered in management decisions, and are therefore not directly applicable to the somewhat narrow code-oriented perspective taken in Figure 5.1.

The conclusion must be that the three categories, analysis, desgn, and architectural patterns together comprise the relevant patterns to describe system development on a technical level.

Chapter 6

Overview of Pattern Categories

Since the emergence of software patterns a large number of different categories of patterns have emerged. Some of the categories are subcategories of others or simply a renaming of an existing category.

This chapter gives a definition and an example of six different categories of patterns. These categories of patterns offer a sufficient basis for applying patterns in system development. In the end of the chapter is found a description of a case study where several different categories of patterns are applied in order to design a system.

6.1 Pattern Templates

For every pattern author there is a style of presenting patterns. The styles are all siblings of Christopher Alexanders style for describing patterns for building architecture, and have more or less the same ingredients.

Alexander defines a pattern to be a three-part rule, which expresses a relation between a certain context, a problem and a solution.

"As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves."[1] For Alexander, who is an architect, the forces working in his field are often the physical forces, and since the physical forces are of less interest in computer science than in building architecture, most pattern descriptions do not use the term forces, but simply describe the problem found in the context. One author who uses the concept forces in pattern descriptions is Buschmann, who describe parts of the problem with different forces and the solution as a balance between these forces. In my opinion, it makes the pattern solution easier to grasp because it shows how different aspects of the problem operate in relation to and against each other. It makes it easier to understand, that a pattern cannot solve the whole problem. There will always be different forces to consider.

[1] From "The Timeless Way of Building" by Christopher Alexander p. 247

6.2 Analysis Patterns

Analysis patterns as described in [Fow97] reflect conceptual structures in the problem domain more than they describe actual software implementations. An analysis pattern describes a typical problem, a part of the reality that could have been modelled in a difficult and inefficient way if the analyst had chosen to use the immediate solution. It then gives a description of how a possible model could look and describes any problems inherent in the model. Using patterns in the analysis is different to the traditional way of doing analysis as described in [MMMNS97] and [RBWPL91]. The descriptions found in these books give tools, like techniques and notations, to create models based on the problem domain and application domain. With analysis patterns the solutions are whole miniature analysis models. Martin Fowler states that analysis models are very important for the development process, since software development should reflect human thinking, not the other way around. The systems must be made to fit us, we should not strive to fit the systems as is sometimes the case when the immediate solution is chosen.

The analysis patterns are divided into 9 categories, each focusing on their own problem domain. Though the patterns are divided into the problem domains in which they were identified, it is not the intention to constrain the use of them to these domains. Within each domain, several descriptions of patterns are given, the first being the simplest. Martin Fowler uses the idea in this pattern to evolve more and more complex patterns. In this process, he specialises both the problem description and the solution. This implies a simple learning process for the reader, while he is introduced to the more complex patterns.

As an example of analysis patterns we look at the pattern collection for observations. The context is an environment where the results of measurements and observations are important to save in a unique and safe way. The setting is a hospital where a patient is being diagnosed or treated. It is necessary to be able to make quantitative statements, such as height, weight, etc. but it is as important to be able to make qualitative statements, such as gender, blood group and whether or not they have diabetes. If we are recording a persons gender, the two possible values are (in the normal case): male and female. We can think of gender as being what we are measuring, and male and female as two values for it, just as any positive number is a value for height. As can be seen from Figure 6.1, an analysis model of this scenario could let an observation be either a measurement, or a category observation, and let these be annotated with quantity and category. This makes gender an instance of phenomenon type, while male, and female instances of category.

We now have to consider how we can record that certain categories can be used only for certain phenomenon types. A blood group can have 'A' as one category, but so can a grading of liver functions on the Child's-Pugh scale. There are two possibilities: we can chose to make the mapping from category to phenomenon type multivalued, so that the category 'A' consists exclusively of the letter 'A', or we could make it singlevalued, which would allow us to record useful information about the categories, such as 'A' is better than 'B' when it is of the phenomenon type 'liver function'. There is no better solution,

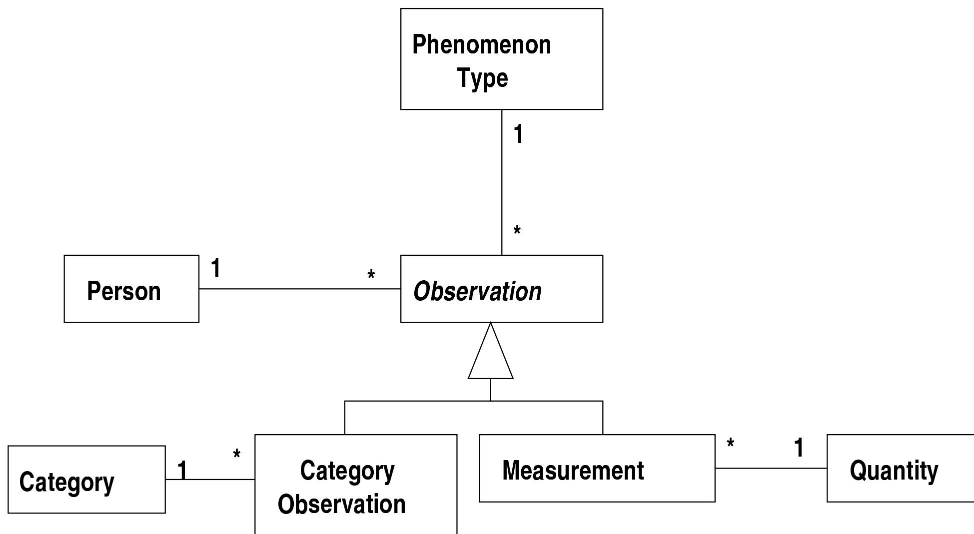


Figure 6.1: Observations and category observations

it depends on the actual case.

If we chose to make the mapping single-valued, the analysis model would be be as in Figure 6.2, where the "category" concept' is renamed to "phenomenon" and linked singlevalued to phenomenon type.

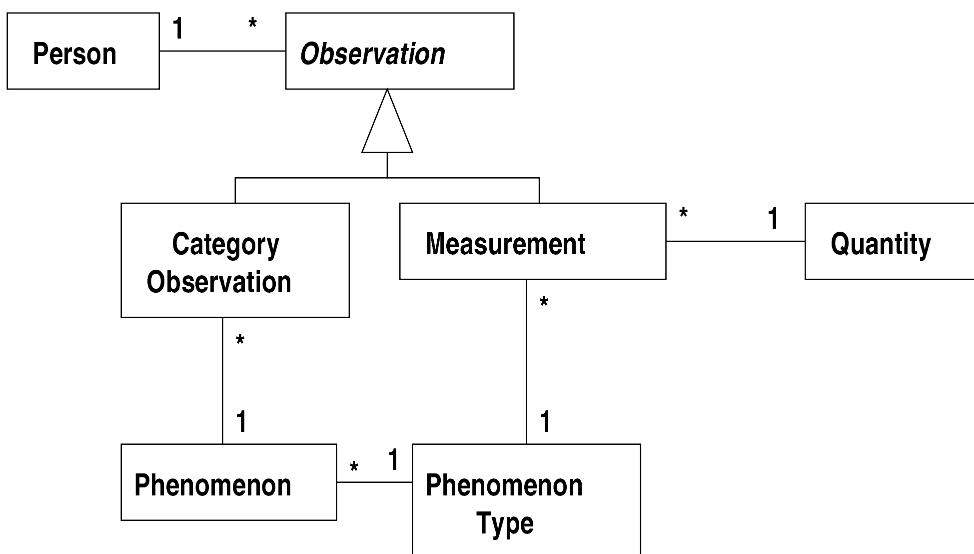


Figure 6.2: The phenomenon formerly known as category

Fowler describes how we can incorporate more requirements in our analysis model, by moving on to the next analysis pattern described within that problem domain. For the sake of brevity the reader is referred to [Fow97] for details.

6.3 Architectural Patterns

Architectural patterns are, as Frank Buschmann describes them in [BMR+97], a fundamental structural organisation schema for software systems. An architectural pattern provides a set of predefined subsystems, specifies their responsibilities and include rules and guidelines for the organisation of the interaction between them. Architectural patterns specify the structural properties of a software system and have an effect on the architecture of the subsystems. The choice of architectural pattern is thus a fundamental decision. Buschmann states that architecture is important because the only way to make surviving software systems is by following superior principles for structure.

An example of an architectural pattern is **Broker**. The **Broker** architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

The context is the development of a distributed and possibly heterogeneous system with independent cooperating components.

The problem is how to make a complex software system as a set of decoupled and interoperating components without a complicated and limiting inter-process communication.

The forces that Broker balances are:

- Components should be able to access services provided by others through remote, location-transparent service invocations.
- You need to exchange, add, or remove components at run-time.
- The architecture should hide system- and implementation-specific details from the users of components and services.

The solution is to introduce a broker component to achieve better decoupling of clients and servers. In Figure 6.3 the class diagram of the solution is shown. Servers register themselves with the broker, and make their services available to clients through method interfaces. Clients access the functionality of servers by sending requests via the broker. A broker's tasks include locating the appropriate server, forwarding the request to the server and transmitting results and exceptions back to the client.

The resulting architecture describes a system, where distributed services can be accessed simply by sending message calls to the appropriate object. Compared to a system where focus is on low-level inter-process communication, the complexity involved in developing distributed applications is reduced. Thus, application of the **Broker** pattern ensures a flexible and extensible system.

6.4 Antipatterns

The concept *antipatterns* has evolved from the concept patterns, to describe often occurring solutions with negative consequences. The authors of the book:

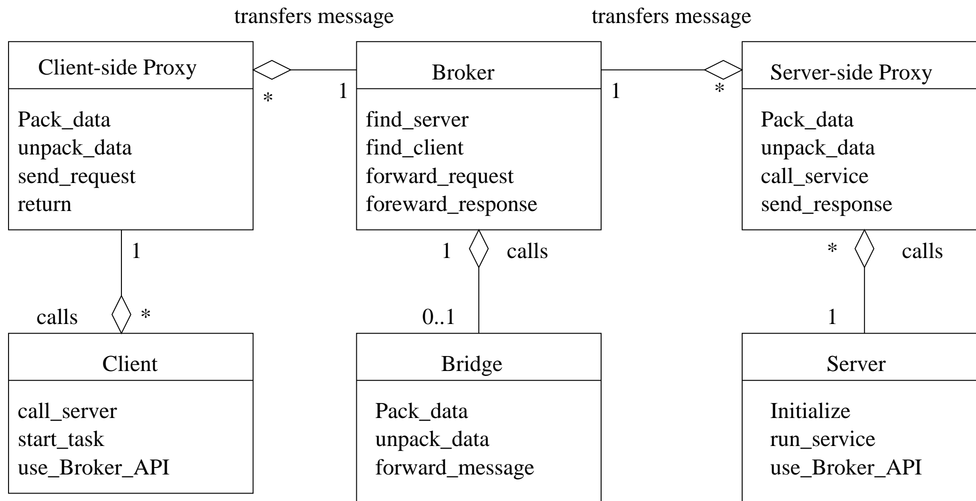


Figure 6.3: **Broker** - An architectural pattern

"AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis" Brown98 define anti patterns as " ... a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences."

Without neglecting the usefulness of patterns, the authors believe their antipatterns to be much more powerful than patterns. This is due to the fact that anti patterns operate on legacy and existing problems, which are more commonplace in practice than the green-field problems often described in pattern descriptions. Moreover, the patterns of yesterday may become the antipatterns of today: "A popular pattern, such as procedural programming, can be the popular paradigm of one era, and fall out of favour in the next as its consequences are better understood."

This shift is indicated by the punctuated line in Figure 6.4, where a design pattern solution over time evolves into the first of the two solutions found in an antipattern description. The first is the problematic solution with more negative than positive consequences, the second is the refactored solution. The refactored solution is a commonly occurring method in which the antipatterns can be resolved and reengineered into a more beneficial form. It is worth noticing that some of the antipatterns describe problems so serious and with so severe consequences that the refactored solution is a description of how to do it right the *next* time. In such cases the authors suggests that the developer who is aware of the antipattern could start preparing his or her resume.

As examples of antipatterns we shall describe **Analysis Paralysis** and **Death by Planning**.

6.4.1 Analysis Paralysis

Where a pattern describes the problem concisely and the solution thoroughly, an antipattern focuses on what lead to the problematic solution, that causes the

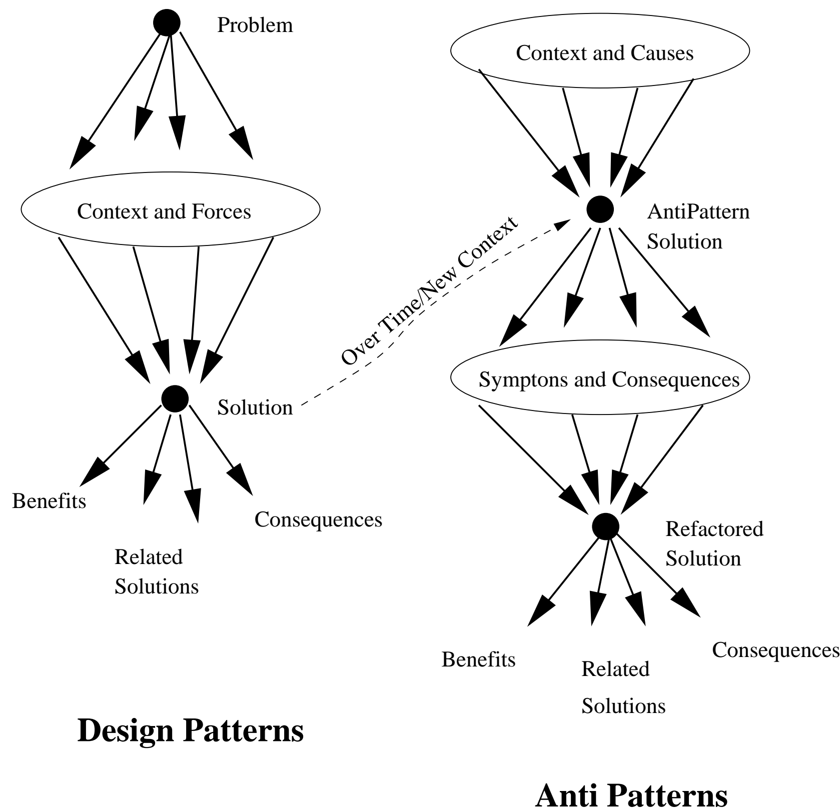


Figure 6.4: Design pattern and Antipattern concepts

grievance. The *root causes* for **Analysis Paralysis** are: “ Pride and Narrow-Mindedness”, the *unbalanced force* is: “Management of complexity”, and one of the *anecdotal evidences* is: “ Well, what if the user wants to create the employee list based on the fourth and fifth letters of their first name combined with the project they charged the most hours to between Thanksgiving and memorial Day of the four preceding years?”.

The problem is that the analysis modelling has gone too far. **Analysis Paralysis** is a classic anti pattern in object-oriented software development due to the way object-oriented analysis is defined. Object-oriented analysis is focused on decomposing a problem into its constituent parts, but gives little advice on when to stop decomposing. This leads to a quest for the mythical “completeness”, that paralyses the developers.

The use of the waterfall model for developing systems is also a factor that may cause paralysis, since the assumption is that the analysis can be successfully completed prior to coding. A thumb rule is that when the domain experts no longer understand the analysis documents, it is time to stop the analysis phase.

The refactored solution to **Analysis Paralysis** is to use incremental development instead of waterfall development.

6.4.2 Death by Planning

The *root causes* for **Death by Planning** are: "Avarice, Ignorance, Haste", the *unbalanced force* is: "Management of Complexity" and the *anecdotal evidence* is: "The plan is the only thing that will ensure our success" and "As long as we follow the plan and don't diverge from it, we will be successful."

Software projects contain by nature many unknowns and chaotic activities, and if detailed plans for a project are taken too seriously the project can end in **Death by Planning**.

The symptoms are for example an inability to plan at a pragmatic level, more time spent on planning, detailing progress, and replanning than on delivery of software and that developers break down their own activities into tasks. The consequences are for example that endless planning and replanning causes further planning and replanning, object shift from delivery of projects to delivery of plans, and in the end failure to deliver a critical deliverable.

The refactored solution describes how to make more general, flexible plans. The project plan should show primarily deliverables and be supplemented with validation milestones. The deliverable plans should be updated weekly to ensure appropriate planning and controls that reduce project risks. When estimating manhours it is advisable to allow a contingency period for all those inevitable "unknowns", such as requirements increase, third party software workaround etc. It is also important to establish a minimum time frame in which to accomplish any activity. This prevents such constraints as two days to code and test a "simple" program.

6.5 Organisational Patterns

Organisational patterns can be used to shape new organisations and development processes.

Jim Coplien has with his paper: "A Development Process Generative Pattern Language" [Cop95a], shown how to make a generative pattern language for organisation. By a generative pattern language, a set of patterns is meant that indirectly generates the right process. This indirectness, as Jim Coplien explains, is the core of Alexander's patterns. These patterns do not make architecture, but generate it indirectly by the ordinary actions of the people. A set of simple patterns can cause complex emergent behaviour. Though the patterns are simple, as Copliens comment, it is not an easy task to apply them.

As patterns as design and architectural patterns are discovered in good designs, Jim Copliens patterns are drawn from singular organisations with unusually high productivity and do not describe the state of the art.

Two patterns ; **Size the Organisation** and **Gatekeeper**, are chosen to give an impression of the pattern language.

Size the Organisation answers the question of how large the organisation should be. The context is an organisation, that creates software with more than twenty five thousand lines of code in the first release of the end product. The forces, that are at work in the context are as follows:

Large organisations are ships that are hard to steer. Small organisations do not have a critical mass. Size affects the deliverable in a nonlinear manner, because communication overhead increases as the square of the size, while the horsepower goes up linearly.

The solution given is to let the organisation comprise ten people, which gives as a result that there is a critical mass yet it is still possible to manage global communication.

Gatekeeper answers the question of how to foster communication with typically introvert engineers. The context is an organisation of developers that has formed in a corporate or social context scrutinised by peers, funders, customers, and other outsiders. The forces are: Information flow is important, Communication overhead goes up non-linearly with the number of external collaborators.

The solution is to let *the* extrovert project member rise to the role of the gatekeeper. This person disseminates leading edge and fringe information from outside the project to project members, translating it into terms relevant to the project.

6.6 Database Patterns

Due to the lack of maturity in object-oriented databases and the large number of existing databases, relational databases are still a part of most software systems. As a consequence of this, many system developers have experienced how hard it is to access these in an elegant and maintainable way from object-oriented programs. What one typically encounters when studying their attempts is SQL-code in the applications or endless duplication of trivial database-access-code in order to access many tables.

Patterns describing solutions for access from the object-oriented design to relational databases plus the actual modelling of the database have emerged because of this, and we have chosen to call these database patterns.

Design of, and access to, databases takes place in all parts of the development process and therefore we claim that the use of these patterns is orthogonal to the use of analysis, design and architectural patterns. Therefore they are included in the middle of Figure 5.1.

The patterns presented are taken from two different papers: "Crossing Chasms" [BW95] and "Relational Database Access Layers" [KC].

In "Crossing Chasms" the chasms to be crossed are the ones between object-oriented programs and relational databases. The patterns in the paper are divided into three groups with nine patterns in each.

The first group is what they call the static patterns. These patterns solve problems related to the difference in type between the objects in the object model and the value attributes in the relations, known as the impedance mismatch problem. Seven of the nine patterns in this group describe how to represent elements of an object-oriented model in a relational database. E.g. identities vs. primary keys, objects, object relations, class hierarchies and collections.

The last two gives advice on how to make the object-oriented program, that should cooperate with the database.

The patterns in the second group are called the dynamic patterns. Unlike the static patterns, that focused on the definition of the database, these patterns deal with the interaction between the information in the database and the object-oriented program, that should retrieve the information. E.g. fetch, save and transactions.

The last group are the client-server patterns. They describe the cooperation between an application program and a database. E.g. distribution of code, data synchronising, locking strategies and coaching.

From "Crossing Chasms" the most central pattern is probably **Broker**, not to be confused with the architectural pattern of the same name. **Broker** is a dynamic pattern, that shows how to encapsulate tables of the database. Every table is encapsulated by a database broker, which deals, not only with the access to the table, but also with constructing objects of the data collected from the table. In addition it accepts objects, whose data contents should be saved in or deleted from the database. A broker class can be parameterised with the object type, such that the common operations are only specified in one place. The broker could for instance contain and control a cache; if it is likely that the broker will be asked to fetch the same data more than once, it will often be necessary that it does not just generate several objects with the same data, but that it returns a reference to the same object every time.

The paper "Relational Database Access Layers" [KC] describes patterns for change of database design after the application program is made. The reasons for changing the database design can be a need for fast access to the database a maintainability or reusability requirement etc. The paper presents a framework pattern called **Relational Database Access Layer**, some structural patterns and a number of tricks for database optimisation.

As an example we have chosen the pattern **Physical Views**. This pattern shows a low-level encapsulation of the database. The encapsulation enables the developer to use different tricks for database optimisation without polluting the application program. In the physical views denomalization and controlled redundancy is used, but this is transparent from the logical view.

The technique used in the pattern is to encapsulate every table and every view in the database with an object called physical view, as shown in Figure 6.5.

The physical views have a common superclass, declaring the shared operations. One can define more physical views as required. The pattern can also be applied within non-relational databases, flat files etc.

6.7 Design Patterns

Design patterns are patterns on an object/class level. They are descriptions of communicating objects and classes that are customised to solve a general design problem. Design patterns are not small designs as linked lists and hash tables, that can be encoded in classes and reused as is. Nor are they complex, domain-specific designs for an entire application or subsystem.

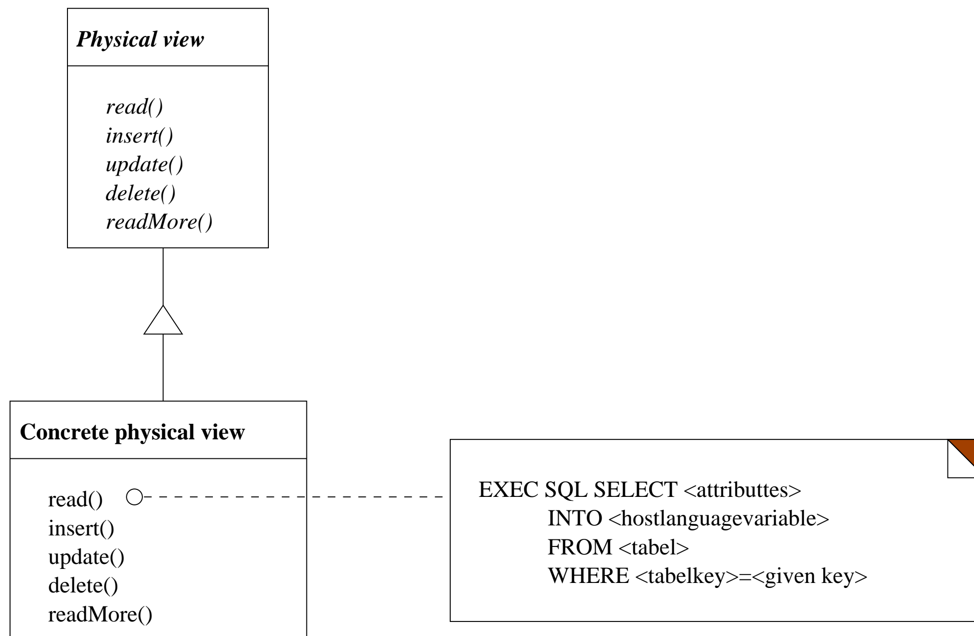


Figure 6.5: Structure of the database pattern **Physical views**.

The solutions found in design patterns are very similar to those of analysis patterns, the difference lies in their focus. Analysis patterns help create analysis models, that can be designed and implemented in several different ways, while design patterns are closer to the code and the solutions found here are thus to be implemented following the guidelines of the pattern.

Some design patterns are described in a sufficiently vague way to make them usable as architectural patterns as well, where the roles that would have been played by objects now are played by subsystems. This has induced an ongoing discussion on what the difference between design and architectural patterns really is. Sometimes the only way to see the difference between a design pattern and an architectural pattern is by changing the choice of pattern, after implementation and see which one is most painful to change. That would be the architectural pattern.

There are database patterns on a design pattern level and on a architectural pattern level, but as mentioned above they have an altogether different focus and thus have their own category.

To get a taste of the strength of design patterns let us use the example of a word processing system, which should be able to create documents containing both graphics and text. The system should be effective, user-friendly and with the possibility for later extensions. Since the system should be a WYSIWYG system, the user must be able to continuously see the resulting document and change it in a simple way. For instance he should be able to create a graphical document by including a picture, drawing a frame around it and placing text beneath it. Once he has created it he would like to be able to change its size, by changing the size of the picture while the frame grows proportionally. He would also like to be able to move it around while the text follows the frame.

In other words, he would like to treat the three elements as a whole.

The above provides us with the following requirements specification:

- Text and graphics must be able to be used in the same way and have the same priority.
- Composite elements and simple elements must be treated as one. In order to be able to work on them without knowledge of their type, they should have the same interface.
- Different types of elements should have the possibility of being analysed in different ways. Eg. spelling control can be applied to a text, but not a picture.

One way to live up to these requirements is to make a recursive composition of the simple components of a document. As a first step letters and graphics can be placed from left to right to from a line in the document. Lines can be grouped to columns and columns can form a page of the document etc. This physical structure can be represented by attaching an object to every element in the document. Not just the physical elements as letters and graphics, but also columns, lines, and pages. The class diagram for this is shown in Figure 6.6

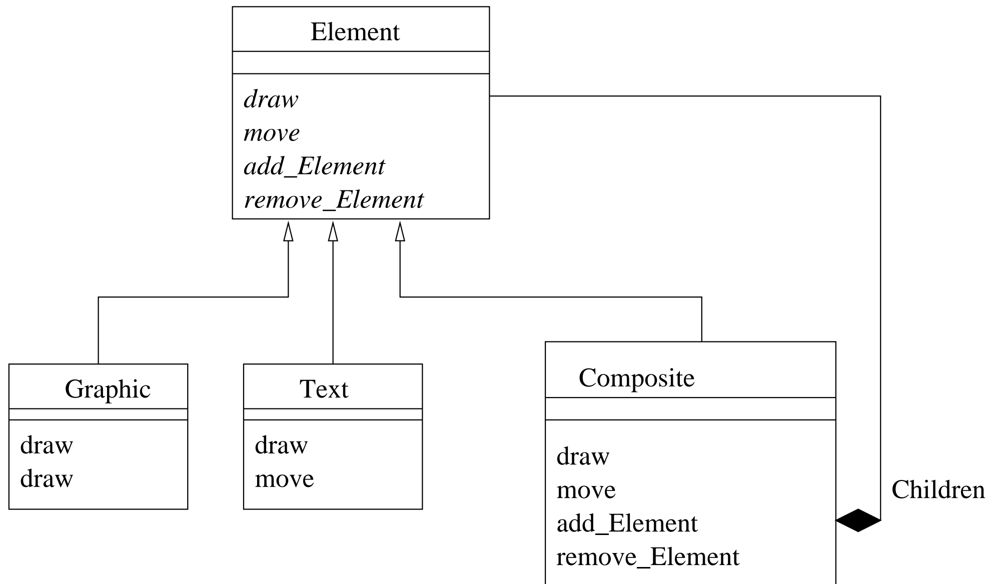


Figure 6.6: The class diagram - **Composite**.

The simple elements, like text and graphic, have their own class, while the composite elements, like columns, pages and composite figures, have a common superclass, *Composite*. All these classes are specialisations of the abstract superclass *Element*, which provides them with a common interface to all kinds of elements. Composed elements have three basic responsibilities. They know how to draw themselves, they know which elements comprise them and when a composed element is asked to deal with a request, it will forward the message

to all its children for them to follow the request. With this structure the system lives up to the user requirements. Text and graphic is on the same level in the class hierarchy, the simple and the composed elements share a common interface and it is easy to extend the system with more types of elements.

This solution of a design problem is found in the design pattern **Composite**, which is described in the GoF-book [GHJV95]. Apart from the benefits of the solution, which we have already described, the pattern also describes the liabilities of using the pattern. One of the liabilities is that it is hard to constrain the types of elements composed at runtime, due to the generic nature of the solution.

6.8 Case: Developing a planning system

To get a feeling of how to apply analysis patterns, design patterns and architectural patterns to the same development process, we shall use the example of software implementing a planning system for a software organisation.

The system is to be able to keep track of the meetings in the organisation. One could imagine a number of things of interest when planning for meetings. Which developers are going to which meetings, which rooms are occupied by which meetings. Different types of employees need to use the system in different ways. The secretaries want to be able to see everything that goes on. The developers want to see when and where they are going to what meetings. The financial controllers want to see how many hours the developers have spent at meetings, but they do not care where the meetings were held. he manager wants to have a nice user interface.

This description of the system would make a developer with knowledge of patterns to think of the architectural pattern **Model-View-Controller**, also known as **MVC** from [BMR+97]. This pattern splits an interactive system up into three parts as shown in Figure 6.7.

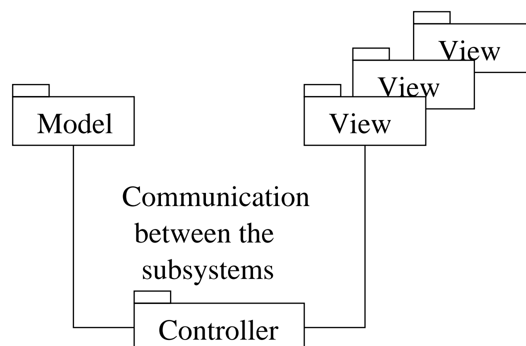


Figure 6.7: The class diagram for **MVC**

The Model part contains data and the functionality, that is not a part of the user communication. The Views show the user that particular part of the system he is interested in, in his preferred way. The Controllers (there is one for every view) take care of the user requests. The views and the controllers together

comprise the user interface. With this partitioning, a separation between the data and the user interface is achieved. This is preferable, since the part systems can be specified and developed independently of each other. The partitioning makes the system more flexible with respect to changes. The users views can be changed without influencing the Controller part or the Model part and vise versa.

With the choice of architecture the skeleton of the system is settled, but a large amount of design work is still needed. The detailed separation of objects and their communication is still to be found. A large part of this will follow from the analysis, but even after the analysis an amount of design work is to be done. This could be because the system needs more functionality than was evident from the analysis. A pattern like MVC describes the underlying structure, but the detailed must be added by use of the more detailed patterns; the design patterns. An open question in MVC is when and where data is changed in the model in accordance to the change of data from the users views. In an object-oriented implementation of MVC at least one class would be made for every component in the pattern, and the Observer pattern[2] could be used to define the communication between the subsystems as can be seen from Figure 6.8.

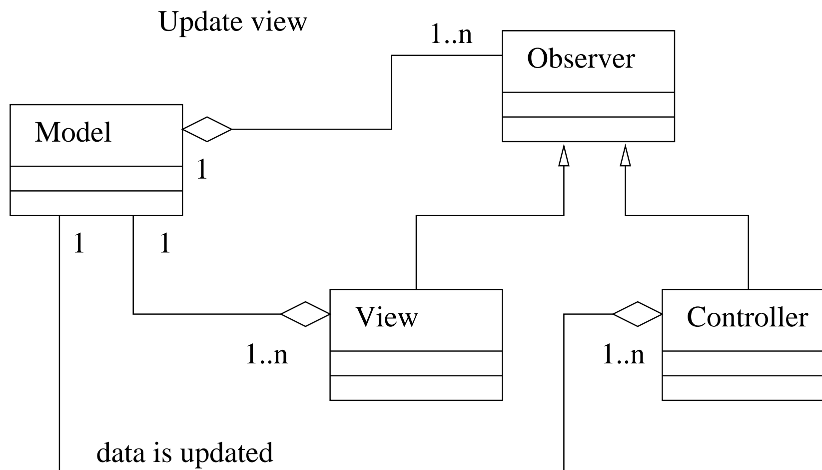


Figure 6.8: The collaboration between **Observer** and **MVC**

The structure of this pattern is almost the same as the structure of MVC. Since Observer is a design pattern, its roles are played by classes, not subsystems. The Control part of Observer lies in the data and view objects instead of being separate from them. The communication between the data and the views is triggered by the user changing the data in his view. This view sends a message to the data object, which automatically propagates the change on to all the other views attached to the data object. When this pattern is used to define the communication between the part systems described by the architectural pattern, the system will inherit its form of communication and automatically propagate all changes to all user views.

[2]described in more detail in Section 7.1.2

During the analysis of the problem domain an analysis model will be created. This model must be made with respect to banal things, such as the fact that a person cannot be at more than one place at a time, and more complex things, such as space constraints in rooms. In this analysis we use an analysis pattern called **Resource allocation**.

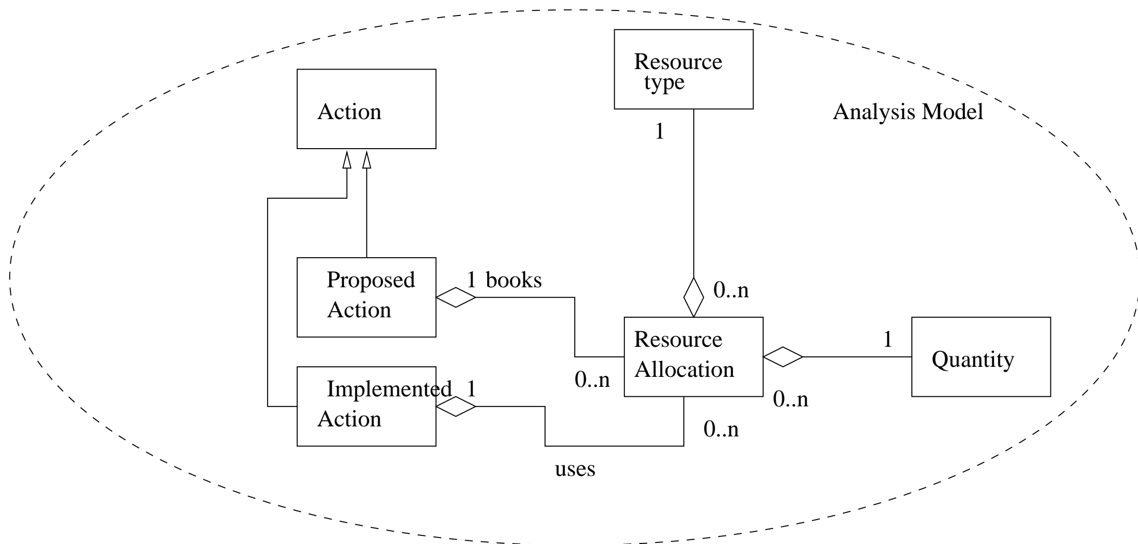


Figure 6.9: The analysis model for planning, **Resource allocation**

As can be seen in Figure 6.9 an action can be either proposed or implemented. The difference lies in whether the resources it is dependent on are still just booked or already used. The analysis model will be the source of design that is to be placed in the Model part of the system (as illustrated in Figure 6.10), since it describes an underlying functionality for the system. The Control part of the system deals with user input, and if necessary propagate information on to the Model part, which can then deal with it according to its functionality.

Since the resource model is an analysis model it does not in itself imply a specific design. It could for instance be sensible to merge resource type and quantity into one class in the design to simplify the design, if it is not relevant in the given case to have two separate classes. If the only resource is paper, it is sufficient to have quantity, because the type is insignificant. In a case like that the design would be unnecessarily detailed if the analysis model was followed literally.

The design pattern **State** from the GoF-book can be applied directly to the analysis model. The intention behind **State** is to give a solution to the problem that arises when it must be possible for an object to change its state, and in turn its functionality at run-time, in accordance to user input. In the model in Figure 6.9 an action is an object that changes its state from proposed to implemented and thus changes its functionality. This functionality can be modelled by the design solution from **State**. The full system is shown in Figure 6.10.

The skeleton of the system is designed using the solution from **MVC** with

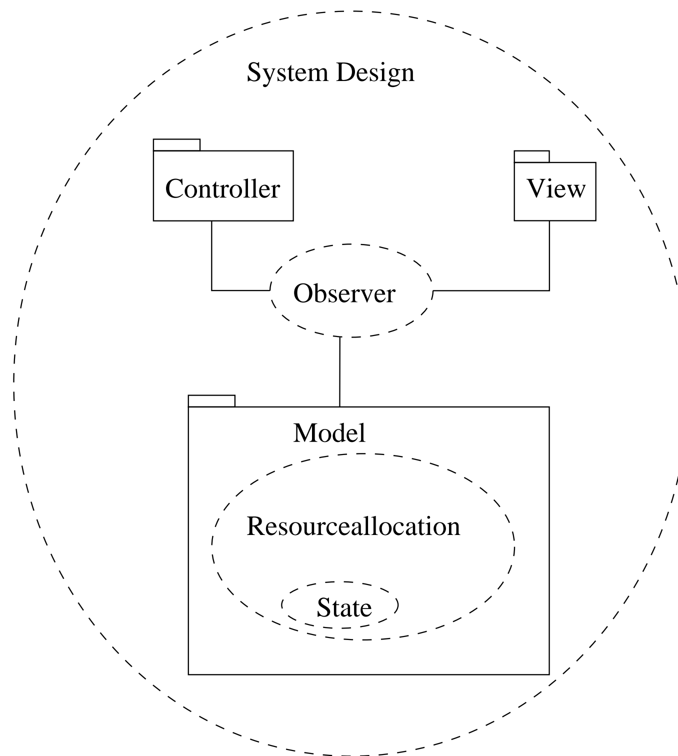


Figure 6.10: The overall system design

an intern communication described by **Observer**. A part of the Model part is described with the analysis model **Resource allocation**, and this model is the basis for a design in which **State** is used.

Chapter 7

Classification of Design Patterns

Design patterns are presented as a means of encapsulating the experience of programmers in a form that is easily communicated to other programmers in all domains regardless of their expertise within computer science.

The benefits that they claim to provide are the following:

A. They encapsulate experience.

B. They provide a common vocabulary for computer scientists across domain barriers.

C. They enhance the documentation of software designs.

Unfortunately design patterns became more popular than was good for them and the rapid evolution of design patterns has hampered the benefits gained from using design patterns.

The objective of this chapter is to promote the point of view that the forming of design patterns should be restrictive, and to suggest a way of evaluating existing design patterns which leads to a reduction of the number of design patterns.

We propose a set of guidelines to follow when evaluating a design pattern, and we present the results of these guidelines applied to the design patterns of [GHJV95].

7.1 An Analysis of Design Patterns

The "schoolbook´´ definition of a design pattern is that it is a description of a well tested solution to a recurring problem within the field of software designs in object oriented languages.

This definition clearly accentuates what the principal idea behind design patterns is; namely to distribute the knowledge of good design, so that designers of software applications can benefit from work previously done in similar areas. However, the definition also leaves it up to the individual designer to decide what constitutes a design pattern since terms like "well tested" and "recurring" are not objective terms that can be evaluated "true" or "false" in an unambiguous way. The consequence of this is that new design patterns appear

in a seemingly endless stream; each of the new design patterns being presented with the best of intents, since they represent some experience to be distributed to the entire society of framework designers. One has but to look at the Patterns Home Page[1] to be convinced that there exist numerous patterns and that the number is continuously increased by PLoP conferences and discussion groups.

The obvious consequence of this is that the number of design patterns will grow to a level, where it becomes impossible to maintain an impression of what design patterns exist, let alone to know which problems these design patterns actually solve. This will in turn destroy the possibility of using the design patterns as a common vocabulary, which otherwise holds the potential of becoming one of the primary benefits of using design patterns. It will also obscure the entire field of design patterns, so that it becomes too hard to find the design pattern to help with a given problem, which may dissuade designers from using design patterns as a helping tool in the design phase. In short, an overdose of design patterns will eliminate two of the three benefits that design patterns offer; they will make it too laborious to find and use the encapsulated experience, and they will make the common vocabulary too large to be easily comprehended.

There are two possible solutions to this problem: One is to restrict the submittance of new design patterns, by inventing restrictions that prospective design patterns must abide to in order for being accepted. The problem with this approach is that too much control in the innovative phase of discovering new design patterns will invariably exclude new design patterns unjustly, since it is next to impossible to find proper restrictions without knowing all potential design patterns beforehand. Another solution is to evaluate the existing design patterns and for each design pattern decide whether it qualifies or not. The problem is again to find the guidelines by which to decide whether or not the prospective design pattern is accepted, but the advantage is that each design pattern will be evaluated in its own right, which should minimise the probability of rejecting a design pattern unjustly.

We will in this chapter present an analysis in the form of a set of criteria, that we have used for an evaluation of the design patterns that are presented in [GHJV95]. Our analysis does not go so far as to identify the *true* design patterns and throw away the rest; instead it focuses on assembling a core of *fundamental design patterns* which should capture good object oriented design on a high enough level so that they can be used in various kinds of applications. The design patterns that are not judged to be Fundamental are then either classified differently or rejected completely.

It is important to note that we do not believe our analysis to be *the* analysis of design patterns. It has evolved from our work with the design patterns from [GHJV95], which means that the criteria are based on a rather narrow set of design patterns. If the analysis was tested on a larger number of design patterns, it might be revealed that the criteria are not sufficient or that some of the criteria are too restrictive in that they unjustly rule out some valid design patterns. We do believe, however, that the criteria form a sound starting point

[1] http://hillside.net/patterns/patterns.html

in a much needed discussion on the quality of the design patterns.

In [AC97a] we have shown that by using the guidelines of this analysis, we can remove half of the design patterns from the core of Fundamental design patterns, so that out of the original 23 design patterns in [GHJV95] only 12 remain. We give some examples of how the guidelines of the analysis are applied to a few of the design patterns — for the complete analysis we refer to [AC97a].

7.1.1 The Analysis

We present an analysis whose purpose it is to restrict the number of Fundamental design patterns. As mentioned above, we believe it is better to have a conservative analysis, that will accept too many design patterns rather than unfairly reject some design patterns. Our analysis is therefore based on three guidelines on when *not* to accept a prospective design pattern. It will be possible to make a stricter analysis by adding further guidelines without changing the original guidelines.

Design Patterns vs. Language Constructs

In [GHJV95] the authors state that one person's design pattern can be another person's primitive building block, because the point of view affects one's interpretation of what is and what is not a design pattern. And the point of view is influenced by the choice of programming language.

In [GHJV95, p. 4] it is said:

"The choice of programming language is important, because it influences one's point of view. Our patterns assume SMALLTALK/C++ level language features, and that choice determines what can and cannot be implemented easily. If we assumed procedural languages, we might have included design patterns called "Inheritance", "Encapsulation", and "Polymorphism". Similarly, some of our patterns are supported directly by less common object-oriented languages."

Thus, they believe that design patterns do not need to be language independent.

We agree with [GHJV95] so far that the design patterns extracted from various applications will always be dictated by the programming language used in the application; things that are easy to do will not be worth forming into a design pattern. But where [GHJV95] seem to believe that design patterns should emerge from each programming language, we are of the conviction that the fundamental design patterns should not be covered by any generally accepted language construct. This point of view is rooted in our belief that a Fundamental design pattern must be independent of any implementation language. There should not be "Design Patterns for C++ programmers´´ or "Design Patterns for Delphi programmers´´, since a such partition would have the following consequences:

- Programmers using one programming language will be able to understand and exchange design patterns with other programmers using the same programming language, but not with programmers using some other programming language. This will either create barriers between programmers

who have essentially the same background, namely the object oriented line of thought, or it will mean that the design patterns will not be used to the full of their potential even within the different societies of programmers. In either case the design patterns will have lost their ability to provide a common vocabulary between object oriented designers *regardless* of their background.

An example of this can be found in [ABW98, p. 3] where the authors justify the need for gathering the design patterns from [GHJV95] in a SMALLTALK version with the following:

"The Gang of Four's design patterns presents design issues and solutions from a C++ perspective. It illustrates patterns for the most part with C++ code and considers issues germane to a C++ implementation. Those issues are important for C++ developers, but they also make the patterns more difficult to understand and apply for developers using other languages.´´

Another example is the book: "Patterns in Java´´ [Gra98], where all 23 GoF design patterns are described in a Java context. It is no surprise, that the implementation part of the description has to change with a change in programming language, but unfortunately some of the more abstract pattern descriptions have also been changed to better suite the taste of the author. This brings us to the next item:

- It will be hard to compare two design patterns coming from different groups of design patterns, since the backgrounds in given programming languages will almost certainly have an impact on the presentation of the design pattern.

- If a programmer who has been used to working in some programming language changes to another programming language, he will have to learn a whole new set of design patterns.

- A collection of language specific design patterns will sooner or later evolve into cover-ups for shortcomings of the programming language, that will explain how things can be done cleverly using some or other language construct.

  An example of this is found in [Cop92], that contains a collection of C++ idioms.

If we then concentrate on building a core of fundamental design patterns, that are not covered by any generally accepted language construct, we can use this core to form the common vocabulary to be used among computer scientists regardless of background.

However, a design pattern which is covered by a language construct in one language might still be a design idea worth preserving in languages which do not have this language construct. Therefore we believe that the design patterns, which are not fundamental because they are language dependent must be kept as *language dependant design patterns* (LDDPs). They should not be partitioned by the languages they are useful in, but rather by which language

construct(s) they are covered by. This way a designer can use the fundamental design patterns (FDPs) plus the part of the LDDPs that is necessary for the programming language he uses for his implementations. In time, we imagine that some of the LDDPs will be removed from the field of design patterns when the covering language constructs are adopted by the majority of the object oriented languages.

These reflections lead to Guideline 1:

design patterns covered by language constructs are not fundamental design patterns.

Design Patterns are Original Ideas

The fields in which the design patterns can be used are numerous. It is an almost certain fact that the various possible applications of some design pattern will not look the same; for each application the roles of the design pattern have been parameterised by roles from the application. There will be restrictions from the applications that were not considered in the design pattern and the design pattern will be forced to adjust accordingly. It might be convenient if these adjustments were recorded in some way, so that programmers who are applying some design pattern in a given field could exploit the experiences from previous applications within the same field. These experiences should in fact be named design patterns in that they clearly fit into the definition of being well-tested solutions to recurring problems, and

- they do encapsulate experience
- they do enhance the documentation of frameworks
- they do provide a common vocabulary within the given field

The obvious problem is that this would cause an explosion of "new´´ design patterns; the disadvantages of which have been discussed in the previous section. These "new´´ design patterns would bring little new of general interest, and they would not be generally understandable for programmers *regardless* of their background. Since these design patterns can be categorized as mere variations or applications of a design pattern, we have chosen to place them as *related design patterns* in design pattern *families*. In each of these families there is a head of the family — the original design pattern — which either is a fundamental or a language dependent design pattern. An example of two families is shown in Figure 7.1, where **Mediator** is head of the family for a group of patterns, including **Observer**, and **Visitor** is head of the family for patterns as **Default Visitor**, **Extrinsic Visitor** and **Acyclic Visitor** (described in the papers [III98] and [cM98]).

When a designer wants to make use of a design pattern, he can get the main idea from the head of the family and investigate the related design patterns for more specific solutions. That these variations will not add to the number of fundamental design patterns will be ensured by Guideline 2:

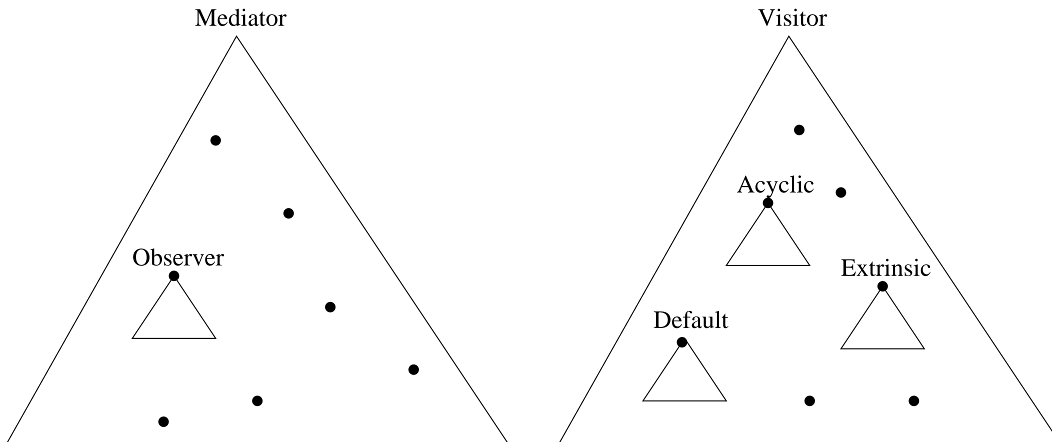


Figure 7.1: Two families of design patterns

Applications and variations of design patterns are not fundamental design patterns.

Design Patterns are Design Ideas

When building an application within object oriented programming, there will be many problems to solve. The size of these problems may naturally differ, as may what appears to be hard problems and what is easily solved. It is therefore difficult to set any limits to the size of problem a design pattern can solve. However since it must be assumed that the programmers who use the design patterns all are schooled in the object oriented line of thought, they posses a common ground of knowledge, that will let them know the answers to certain problems without too much thought. In [GHJV95] the authors have an introductory section containing good advice as to how to apply the object oriented concepts to build flexible, reusable software. It is among other things here explained when to use class inheritance as opposed to when to use composition. These kinds of advice are things that should be common knowledge to programmers in object oriented programming and will therefore not be thought of as problems needing an explicit solution. So even though these advice do represent solutions to recurring problems within the field of object orientation they are not cast out as new design patterns.

New design patterns must represent solutions to actual problems in design that could be of interest to the society of object orientation in general, *regardless* of one's previous experience.

This leads to Guideline 3:

A design pattern may not be an inherent object oriented way of thinking.

7.1.2 Applying the Analysis

We have applied the analysis on the design patterns in [GHJV95]. The design patterns presented in this collection are probably the best known patterns in the area, which should enable the readers of this paper to focus on the analysis and its results instead of on the functionalities of the design patterns. Furthermore they are presented as domain independent patterns, and even though they lay no claims as to being an exhaustive collection of design patterns in the field of object-oriented design, they are fairly widely spread in their proposed uses, so we felt that they would provide a sensible base. For the obvious reasons of space, we will not present the evaluations of all 23 design patterns in this paper, but instead present a few examples of the application of each guideline on a design pattern. For the detailed analysis of all the design patterns we refer to [AC97a].

Factory Method

The purpose of this design pattern is to create objects whose exact classes are unknown until runtime. This is done in [GHJV95] by instantiating the objects in virtual methods that can be bound at runtime as shown in Figure 7.2.

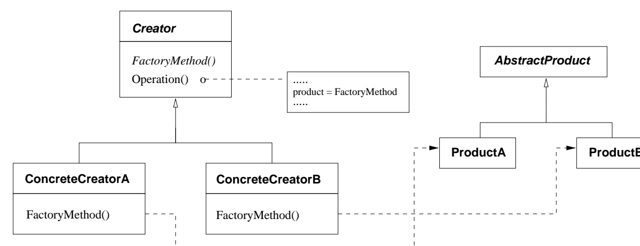


Figure 7.2: The **Factory Method** Design Pattern

In a language with *virtual classes* the goal of this design pattern can be achieved quite differently. Using the extended UML notation from Secion A.2 we can now show how to use virtual classes instead of **FactoryMethod** to guarantee that the productclass can be chosen by the subclasses of the creatorclass. Instead of having a virtual *creator*-method to handle what concrete class to instantiate at runtime, it is now possible to attack the problem more directly by making the product-class virtual. This makes it possible to bind the class to be instantiated at runtime, instead of binding the *creator*-method at runtime.

An advantage in using virtual class patterns is that it is not necessary to rewrite a new FactoryMethod for each concrete product class. Since the pattern solves a problem concerning the change of product at run-time, not functionality, the fact that it is the virtual class binding that is rebound makes the implementation of the pattern in BETA more correct, conceptually speaking. Furthermore it is now possible to extend the interface of the AbstractProduct-class, which is not possible using the original **FactoryMethod** design pattern.

It is clearly demonstrated that **FactoryMethod** is covered by the language construct virtual classes, and according to Guideline 1 it should therefore not

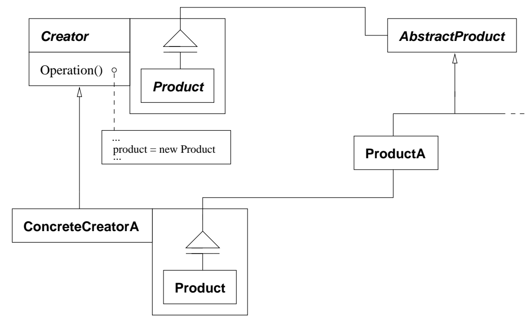


Figure 7.3: **Factory Method** modelled using virtual classes

be accepted as a fundamental design pattern, but should instead be classified as a language dependant design pattern to be used in programming languages without virtual classes.

Template Method

This design pattern is ecpecially useful when a framework has some methods that are meant to be specialised in specific applications, and the idea is then that the variable parts are made as *hook methods* that are called from *template methods*.

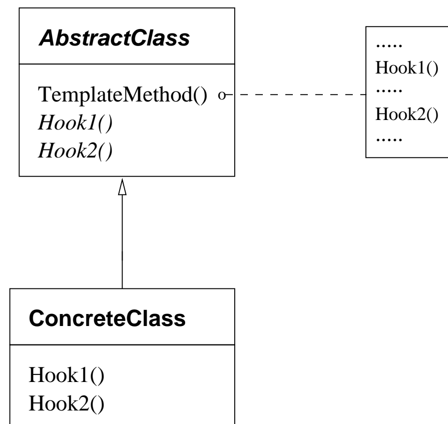


Figure 7.4: The **Template Method** Design Pattern

If the template method only has one hook method, it is obvious that one can use the inner construct of the virtual procedures if the implementation language supports such a construct.

A precondition for extending the functionality in a subpattern is that the inner construct is placed in the virtual procedure of the superpattern where the extended functionality is wanted. So when a part of the functionality of a template method is known only at run-time, one would submethod it to its different functionalities, and place inner where the functionality should differ.

Implementing the *single hook* **Template Method** in this way thus removes the Hook-method from the design pattern, so that the structure now is reduced to the one shown in Figure 7.5

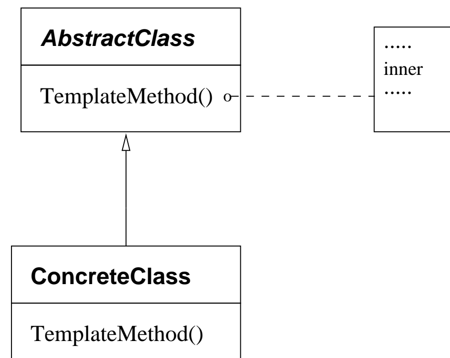


Figure 7.5: *single hook* **Template Method**

As described in Section A.1.1, having virtual nonterminals in the implementation language will cover the situation in which several inners are wanted, and thereby the situation with more than one Hook method.

An ordinary specialisation of a Template method would, when using virtual nonterminals, then be done in the following way:

TemplateMethod: (#
Attribute-part AT: Specification of the attributes
⟨virt_AT: Attributes⟩
Action-part ACB: Specification of actions before extensions
⟨virt_AC^1: Action-Part⟩
Action-part ACA^1: Specification of actions after first extension
⟨virt_AC^2: Action-Part⟩
Action-part ACA^2: Specification of actions after second extension
.
.
.
⟨virt_AC^n: Action-Part⟩
Action-part ACA^n: Specification of actions after n'th extension
#)

HookMethod: TemplateMethod(#
virt_AT:: *Specification of extending attribute-part*
virt_AC^1:: *Specification of first extending action-part*
virt_AC^2:: *Specification of second extending action-part*
.
.
.
virt_AC^n:: *Specification of n'th extending action-part*

#)

In a language with complete block structure, the effect that several inners provide can be alternatively simulated by declaring empty virtual methods inside the TemplateMethod, and then calling these methods where the extensions are wanted. In the concrete subclasses, the TemplateMethod is specialised by extending the virtual methods.

This is a somewhat more elegant solution to the problem than the original Design Pattern.

In conclusion we have the single-hook **Template Method** covered by extensible virtual methods using the inner-construct from BETA, and the multiple-hook **Template Method** covered by virtual nonterminals, which is possible to simulate in any language that supports complete block structure. Thus **Template Method** is an idiom.

Observer

The motivation behind this design pattern is to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. An amount of data (a Subject) can have many representations (Observers) and when one of these representations is changed by the user, the data behind it and all the other representations will be changed. The representations do not know about each other. This enables a user to add or delete new representations as he wishes.

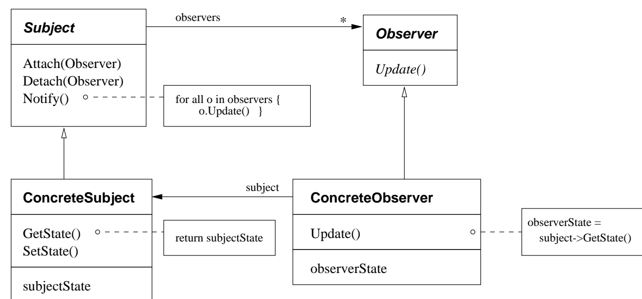


Figure 7.6: The **Observer** Design Pattern

We claim that this design pattern is in fact an application of the **Mediator** design pattern. The **Mediator** design pattern defines an object (a Mediator) that encapsulates how a set of objects (Colleagues) interact. The intent of the design pattern is to promote loose coupling by keeping objects from referring to each other explicitly, and it makes it possible to vary their interaction independently. The structure is shown in Figure 7.7.

When the functionality of an **Observer** is desired, an application of the **Mediator** design pattern can be implemented instead by letting the ConcreteSubject play the role of the ConcreteMediator and the ConcreteObservers play the role of the ConcreteColleagues. Thus the ConcreteSubject will be the mediator between

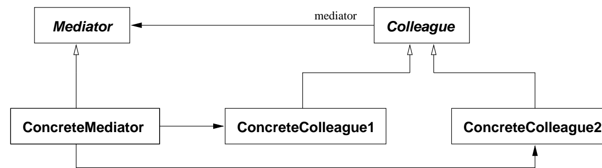


Figure 7.7: The **Mediator** Design Pattern

the ConcreteObservers and the communication it needs to handle will be the notification procedure. That Notify is to be called whenever the state of the ConcreteSubject changes is an application specific feature, that is added in the "observer-part".

There *is* more information in an **Observer** than in a **Mediator** since the communication between the Subject and Observers is fixed, but this is why it is an *application* of **Mediator** and not just a variant.

According to Guideline 2, the **Observer** design pattern should therefore *not* be a fundamental design pattern, but a related design pattern belonging to the family of **Mediator** design patterns.

Visitor

The intent of the **Visitor** design pattern is to be able to define a new operation on the elements of an object-structure, without changing the classes of the elements on which it operates. This is accomplished by representing the operation as Visitor classes.

The example used for motivation is a compiler that represents programs as abstract syntax trees (ASTs), where the nodes represent assignment statements, variable accesses, arithmetic expressions, and so on. The set of node classes depends on the language being compiled, of course, but it doesn't change much for a given language. The set of operations though, might change, since the compiler should traverse the AST many times for many different reasons. The **Visitor** design pattern is a good choice here because the set of operations change a lot while the set of nodes is rigid.

The structure of the **Visitor** design pattern is shown in Figure 7.8.

The figure is to some extent misleading, since the Operation-methods in the elements are irrelevant to the design pattern. They are only made for the ConcreteVisitor classes to access the attributes in the elements. This is an encapsulation- and, thereby, language-specific implementation detail and thus not relevant.

The collaboration between the classes in the design pattern is shown in the collaboration diagram in Figure 7.9 taken from [GHJV95].

As can be seen, the diagram follows the structure from Figure 7.8, and is thus as misleading as that, when it comes to the Operation method. To visualise this, we have chosen to draw parts of this diagram with dotted lines; the parts which from our point of view are not part of the design pattern.

What happens is that an object structure, containing the elements, calls the Accept method on all its members with a ConcreteVisitor as argument. A

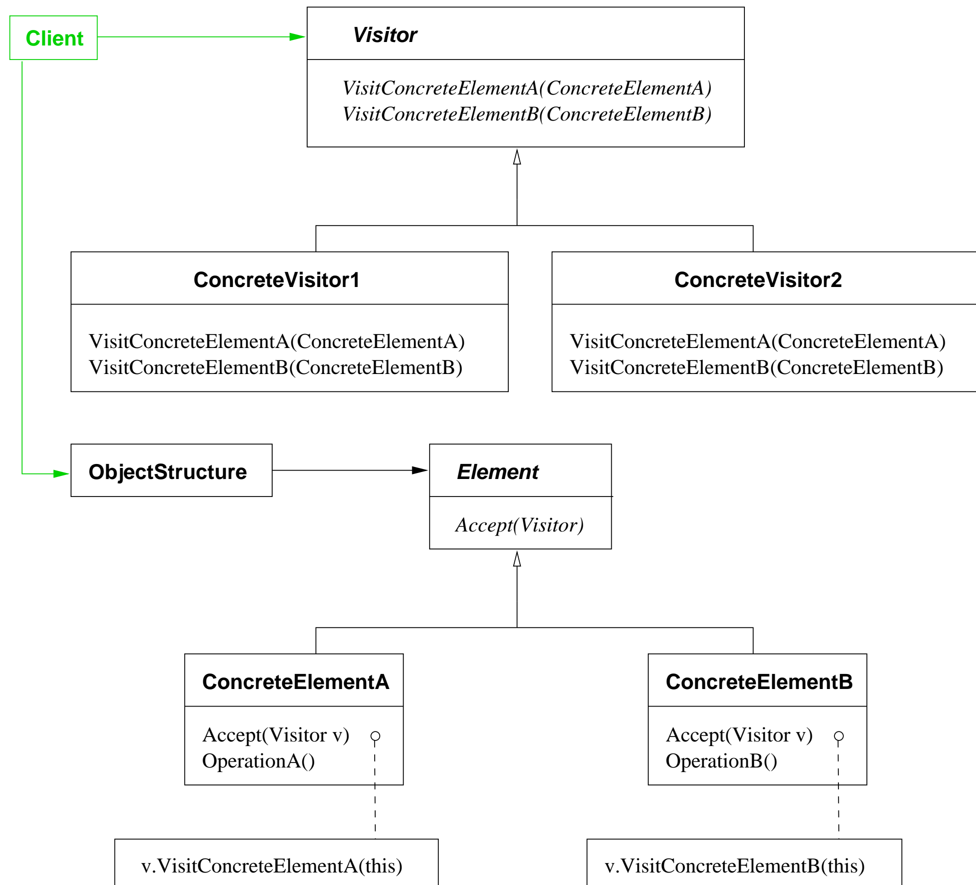


Figure 7.8: The **Visitor** design pattern

call of the Accept method in a ConcreteElement triggers a call to the Visit-ConcreteElement method, in the ConcreteVisitor from the argument, with the ConcreteElement as argument. This enables the ConcreteVisitor to perform the correct operation on the ConcreteElement.

It is crucial to have virtual methods in the implementation language to change the operations without changing the element hierarchy. But this is a feature that is found in almost every object-oriented language. If a language has multiple dispatch, however, the Design Pattern is not needed at all since the clue in the **Visitor** concept is the accept-visitor coordination performing double dispatch.

And since this is a feature in e.g. CLOS, we have chosen to view this design pattern as an LDDP. The **Visitor** design pattern is an example of an LDDP with a family. One of the family members, **Acyclic Visitor**, is found in [cM98]. The intent behind the pattern is to allow new functions to be added to existing hierarchies without affecting those hierarchies, and without creating the troublesome dependency cycles that are inherent to the **Visitor** pattern. This sounds promising, but the catch is that the solution depends on multiple

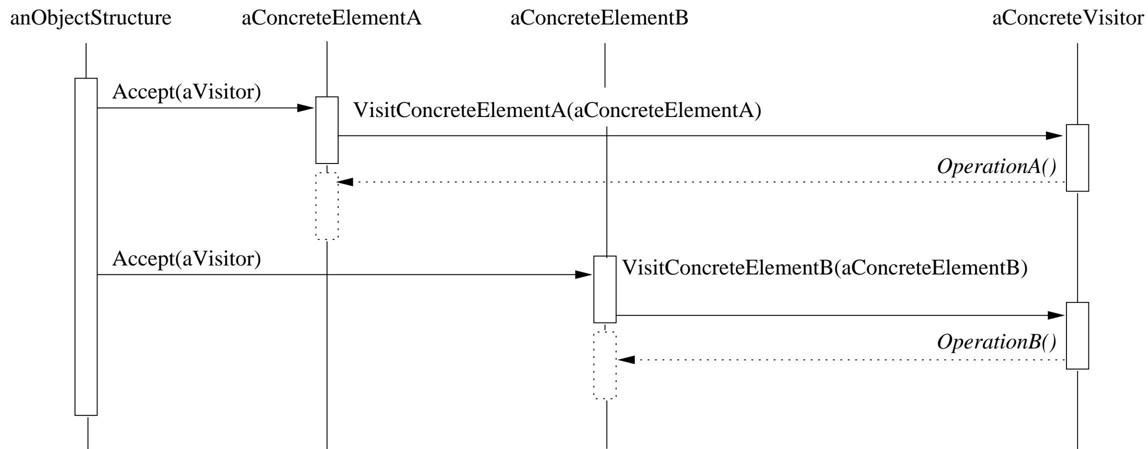


Figure 7.9: The collaboration in **Visitor**

inheritance and dynamic cast[2]. **Default Visitor** and **Extrinsic Visitor** are found in [III98] and give a more precise description of two of the variations mentioned in the GoF description of **Visitor**. **Default Visitor** adds another level of inheritance to the Visitor pattern and thus provides a default implementation that takes advantage of the inheritance relationships in a polymorphic hierarchy of elements. **Extrinsic Visitor** trades the performance overhead of a small number of run-time type tests for reduced complexity and coupling in the visitor and element classes of the pattern. The pattern also provides the ability to easily test the feasibility of a visit operation before actually performing it.

Strategy

The **Strategy** design pattern defines a family of algorithms, encapsulates each one and makes them interchangeable. **Strategy** lets the algorithm vary independently of clients that use it. It is useful when many related classes differ only in behaviour, because it makes it possible to configure a class with one of many behaviours. The design pattern can also be applied when a class has many conditional statements in an operation, to avoid it becoming clumsy and confusing. Each behaviour can be placed in its own class, thus building a simple hierarchy of behaviours. The structure of the **Strategy** design pattern is shown in figure 7.10.

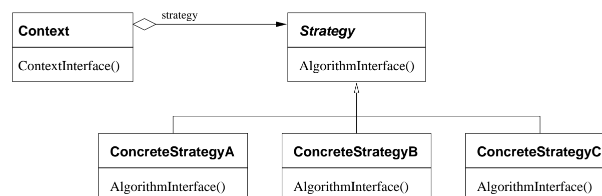


Figure 7.10: The **Strategy** Design Pattern

[2]a cast, that allows you to not only downcast but cast across a hierarchy

When comparing the applicability of the **Strategy** design pattern with the intent of the **State** design pattern in [GHJV95, pp. 305], it will appear that **State** solves the same problem as **Strategy**, thus making **Strategy** redundant. Both aim to encapsulate behaviour in objects, but whereas **State** wants the behaviour to reflect the state of the context and therefore change at runtime, the **Strategy** Design Pattern leaves it up to the client to choose a concrete strategy to work with. In the **State** design pattern it should be possible to change directly from one state to another when some condition is met, which means that the different concrete State classes have to be interdependent so that they can pass whatever data is necessary to one another. In the **Strategy** design pattern, it is the client that decides what ConcreteStrategy to apply, and the data needed by the ConcreteStrategy will be provided by giving the Context object as argument to the Strategy.

It is thus obvious that there is a fundamental difference between the two design patterns, but it is not one that is visible from the structures of the design patterns as presented in [GHJV95]; in fact the close connections in the purposes of the two design patterns is mirrored in almost identical structures of the design patterns.

Evaluating the **Strategy** design pattern we believe that announcing this as a design pattern is stretching the concept of design patterns too far. Having different implementations of some method encapsulated in virtual methods, and using dynamic dispatch for binding them at runtime should represent a fundamental way of thinking when programming in an object-oriented language.

We conclude that the **Strategy** idea should *not* be a Design Pattern according to Guideline 3.

Adapter

The purpose of this design pattern is to convert the interface of a class into another interface which clients expect.

The motivating example for this design pattern is an application with a drawing editor, that has a graphical object, Shape, as its key abstraction. Shape is an abstract class which supplies the interface that all drawable objects should conform to — they must have an editable shape and they must be able to draw themselves.

In the example they bring up the possibility of stealing a class TextView that can display and edit text, so the programmer doesn't have to write his own TextShape-class. The problem is then that the TextView-class doesn't conform to the Shape-interface, so it is not possible to use TextView- and Shape-objects interchangeably.

The solution is to make a TextShape subclass of the Shape class to adapt the Shape objects to the TextView-interface. In [GHJV95] the authors propose two different ways of doing this; the **Object Adapter** (as shown in Figure 7.11) and the **Class Adapter** (as shown in Figure 7.12).

The connection between the names of the motivating example and the general solutions of the **Class Adapter** and the **Object Adapter** is shown in the

following table:

Class name	Original Pattern Role
Shape	Target
TextShape	Adapter
TextView	Adaptee

The **Object Adapter** will have the Adapter hold a reference to the Adaptee and thereby implement its functionality in terms of the Adaptee's interface as shown in Figure 7.11.

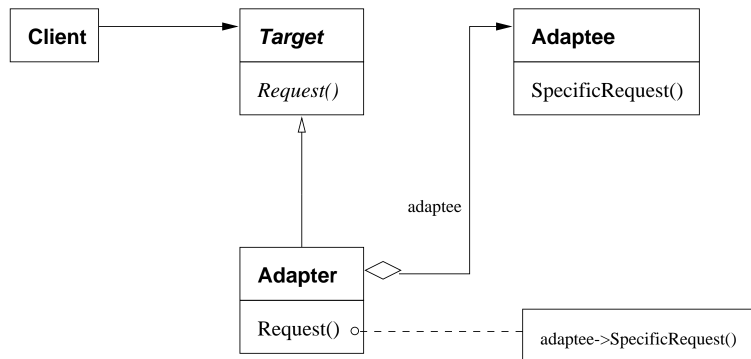


Figure 7.11: The **Object Adapter** Design Pattern

This is quite straightforward to implement in BETA.

The **Class Adapter**, shown in Figure 7.12, will use multiple inheritance to enable the Adapter to inherit the Target's interface alongside with the Adaptee's implementation.

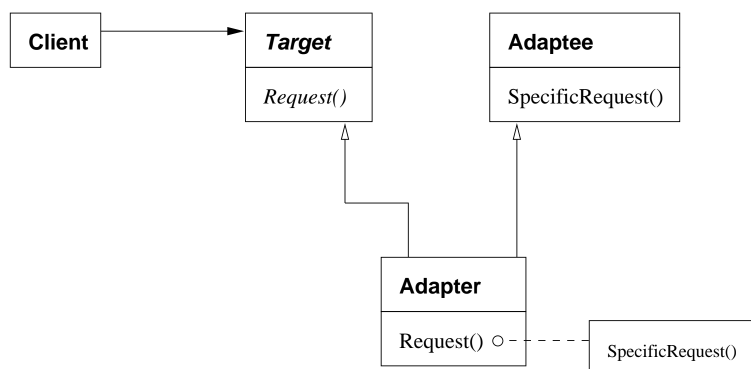


Figure 7.12: The **Class Adapter** Design Pattern

This might seem to be the best solution, but the fact that it relies on multiple inheritance, makes it impossible to implement in languages without this feature. Additionally, **Class** and **Object Adapters** have different trade-offs. A **Class Adapter**

- adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a **Class adapter** won't work when we want to adapt a class *and* all its subclasses.
- lets Adapter override some of Adaptee's behaviour, since Adapter is a subclass of Adaptee.
- introduces only one object, and no additional pointer indirection is needed to get to the Adaptee.

An **Object Adapter**

- lets a single Adapter work with many Adaptees – that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
- makes it harder to override Adaptee behaviour. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

An alternative solution could be to use the nesting of classes that is possible in Beta. This can in some extent simulate multiple inheritance as discussed in Chapter 8.1.1, and thereby behave as the **Class Adapter**.

The Adapter-class should in this version be defined as part of the Target-class, as a (possibly trivial) specialisation of the Adaptee-class.

The structure of the **Nested Adapter** design pattern is shown in Figure 7.13.

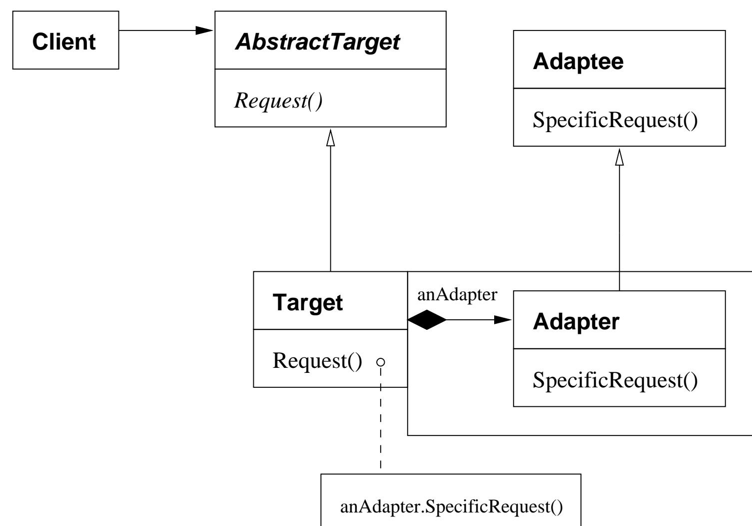


Figure 7.13: **Adapter** using nested classes

It should here be noted that the names of the Target-classes are adjusted a little in comparison with the original **Adapter** Design Pattern as indicated below where the roles of the classes in the motivating example are listed for the two design patterns:

Class name	Original Pattern Role	Nested Pattern Role
Shape	Target	AbstractTarget
TextShape	Adapter	Target
TextView	Adaptee	Adaptee

When comparing the **Nested Adapter** design pattern with the **Object Adapter** and the **Class Adapter** we find that the **Nested Adapter** has almost the same pros and cons as the **Class Adapter** when compared to the **Object Adapter**. It is possible to override Adaptee behaviour in the Adapter using the **Nested Adapter** design pattern, but since the Adapter is nested into Target, an indirect pointer or a renaming scheme is needed to touch the Adaptee behaviour from the Target class. Additionally, the nested Adapter is static and it will be hard to make an Adapter work with many Adaptees.

Thus we conclude that the **Nested Adapter** design pattern is a way of obtaining the advantage (overriding the Adaptee behaviour) of the **Class Adapter** in BETA without using multiple inheritance.

In SELF, data *parents* explicitly implement the representation mechanism found implicit in class-based languages. Objects contain named slots which may store either state or behaviour, and if an object receives a message and it has no matching slot, the search continues via a parent pointer. Thus delegation becomes explicit and no TextShape class is needed.

It could thus be argued that in delegation-based languages with explicit delegation, this design pattern is an idiom.

In [AC97b] and [AC98] we argued that **Adapter** should not have been a design pattern. The reason was, that since this design pattern only deals with reuse of behaviour i.e. reuse of code from one class-hierarchy to another, it is not following the Scandinavian approach where it is the modelling and design that is worth reusing instead of reusing code.

Since then, Java has become very popular and contains a language construct, which supports the idea, the *interface*. This language construct is developed on demand, because the need for adapting code is existing and relevant. Since design patterns have emerged from good design, they also visualise good, and needed solutions. Based on this a revision of the somewhat harsh classification of **Adapter**, is now reversed and the design pattern is classified as a LDDP. Ideally, though, I still believe that it could be made redundant in good design.

Results

For each of the design patterns in [GHJV95], we have in [AC97a] discussed whether it is covered by a known object oriented language construct (and thereby an LDDP), an application of another design pattern (an RDP) or an inherent way of thinking in object-oriented programming. The results of this analysis are shown in table in Figure 7.14.

The **Chain of Responsibility** is evaluated as a fundamental design pattern even though it could be covered by having explicit delegation as a feature in the language. But since this is only found in delegation based languages, the idea

Name	Qualifies as design pattern	Application of Guideline
Abstract Factory	FDP	
Builder	FDP	
Factory Method	LDDP	1: Covered by Virtual classes
Prototype	LDDP	1: Covered by Pattern variables
Singleton	LDDP	1: Covered by Singular objects
Adapter	LDDP	2: explicit delegation.
Bridge	FDP	
Composite	FDP	
Decorator	FDP	
Facade	LDDP	1: Covered by Nested Classes
Flyweight	FDP	
Proxy	FDP	
Chain of Responsibility	FDP*	(1: Covered by Explicit Delegation)
Command	LDDP	1: Covered by Procedure classes
Interpreter	RDP	2: Application of **Composite**
Iterator	FDP	
Mediator	FDP	
Memento	FDP	
Observer	RDP	2: Application of **Mediator**
State	FDP	
Strategy		3: Dynamic dispatch
Template method	LDDP	1: Covered by Complete block structure
Visitor	LDDP	1: Covered by Multiple dispatch

Figure 7.14: Analysis of design patterns from [GHJV95]

will solve an actual problem in all *class based* languages and should therefore remain a design pattern.

Concludingly, the design patterns left as good design ideas seen from a general object oriented view are the twelve marked as an FDP in the table in Figure 7.14.

This leads us to conclude that it is beneficial to have a critical approach to design patterns, because it minimises the amount of fundamental design patterns and thereby makes the area of design patterns easier to get on top of.

7.2 Related Work

Since design patterns are a reasonably new concept, most of the efforts so far have been put into discovering new design patterns and investigating their

usefulness. To the best of our knowledge, little work has been done in evaluating the existing design patterns. The only other critical evaluation of design patterns we have found is the article "Design Patterns vs. Language Design" ([GL97]) where Joseph Gil and David H. Lorenz have offered a taxonomy of the design patterns from [GHJV95] based on how far they are from becoming actual language features. They have partitioned the design patterns as either *clichés*, *idioms* or *cadets*, which correspond to an application of Guideline 1 and 3 from our analysis on the design patterns. The taxonomy was presented as a workshop paper at ECOOP'97, and it needs a more thorough argumentation for its classifications, which we have discussed in depth in [AC97a]. Their resulting taxonomy is difficult to compare to ours directly, since they allow the same design pattern to appear in several categories, and their reasonings are somewhat fuzzy at places. However the fact that the two categorisations are not identical shows that it will be hard to obtain a consensus on any one evaluation of design patterns; especially it will be hard to agree on what design patterns are formalisations over inherent object oriented ways of thinking — [GL97] claims that three of the design patterns fall into this category, none of which we have categorised in the same way. However the fact that two similar set of guidelines have evolved independently indicates that they can be used as valid starting points for a dialogue on the quality of design patterns.

As another way of comparing design patterns, several formalisations of design patterns have been proposed. Amnon Eden has done research in this field, and in his ph.d. thesis, "Precise Specification of Design Patterns and Tool Support in Their Application", [Ede00], he describes how the design patterns can be specified, and how this specification can be useful in many ways. One of the obvious benefits of a formalisation of design patterns from our point of view, is the possibility of a comparison of the design patterns.

7.3 Conclusion

The objective of this chapter is to use classification to help regain the benefits of using design patterns:

A. They encapsulate experience.

B. They provide a common vocabulary for computer scientists across domain barriers.

C. They enhance the documentation of software designs.

We believe that the field of design patterns should be narrowed down to a minimum, to preserve the first two benefits of design patterns. By partitioning the design patterns into fundamental design patterns, language dependant design patterns and related design patterns, we have a core of the design patterns — the fundamental design patterns — which fully provides the benefits of design patterns. Only 12 of the 23 design patterns from [GHJV95] are classified as fundamental design patterns following these criteria. This leads us to conclude

that it is beneficial to have a critical approach to design patterns, because it minimises the amount of fundamental design patterns and thereby makes the area of design patterns easier to get on top of.

Using design patterns in software systems should make it an easier task to document the systems. There is however the problem, that the more design patterns that are applied, the more difficult it will be to recognise the structure of the participating design patterns. This is referred to as the tracing problem, and is adressed in the following two chapters.

Chapter 8

Library Design Patterns - LDPs

In the last chapter we proposed a set of guidelines to follow when evaluating a design pattern, and we presented the results of these guidelines applied to the design patterns of [GHJV95].

For the design patterns which remained original ideas, fundamental design patterns, after this evaluation, we have investigated how they could be placed as a *library design pattern* in a class library and reused by use of inheritance or delegation — any such design pattern will be denoted an LDP. One of the advantages of using such LDPs is that one doesn't have to copy the structure of the design pattern anew each time a design pattern is applied in a new context, thereby reducing the *implementation overhead*; a problem connected to the use of design patterns identified by Jan Bosch in [Bos97]. Another advantage is that by using the LDP it will be possible to *trace* that the design pattern is used in an application, which consequently will promote the documentation benefit.

8.1 Solving the Tracing Problem by certain language features

One of the advantages gained by using design patterns is that large software systems are better documented because a large part of the explanation on how the system works lies in which design patterns that have been used to design the system.

But when the designers have used a large number of design patterns in their applications and some application classes play roles in more than one design pattern it becomes difficult to trace, which design patterns have been used. This problem, known as the *Tracing Problem*, is illustrated in Figure 8.1.

The figure shows how the use of the rough notation, in this case UML, for the use of patterns is insufficient to give a good documentation. The result is simply confusing.

The solution to this problem could be the use of "Library Design Patterns" (in short LDPs). When using LDPs in the application code, it will be possible to *trace* from which design pattern the implementation ideas came.

It is generally recognised that design patterns provide a common vocabulary that makes it possible for designers from widely different application areas to

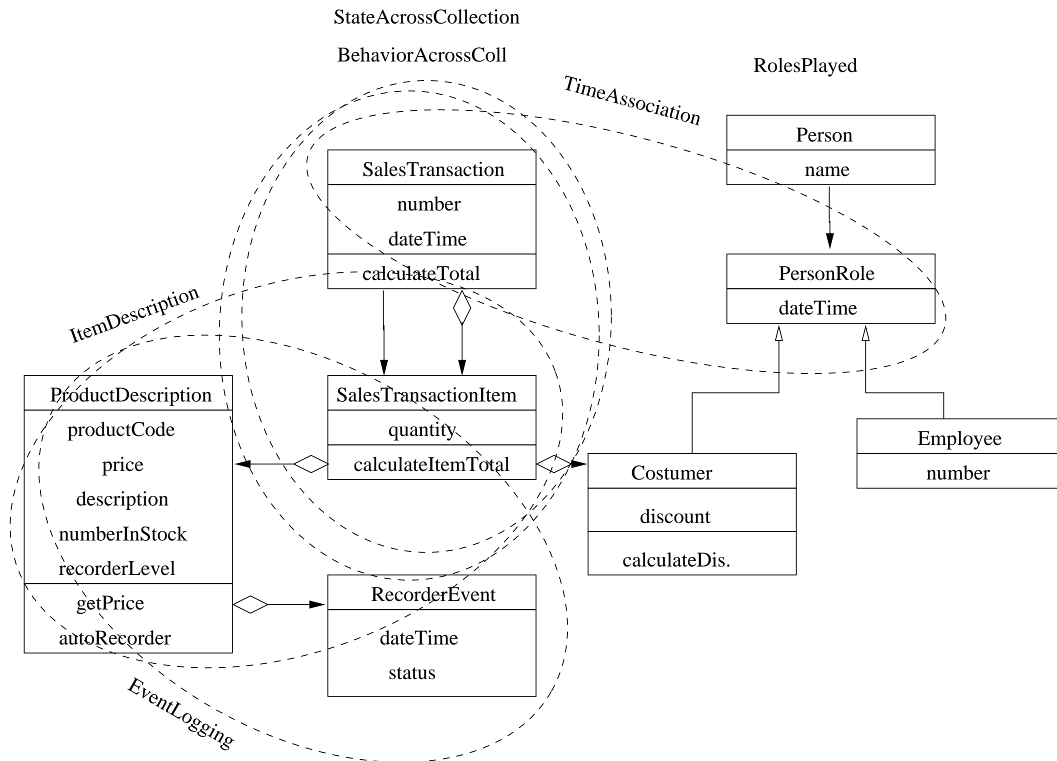


Figure 8.1: Documenting the application of design patterns with UML

communicate with each others. If designers were to make a habit of using commonly known design patterns in their applications, it would make it easier for outsiders to read and understand the programs and thereby making long term maintenance an easier task.

We believe that a way of promoting the habit of using design patterns is to have the design patterns as LDPs in a library where they are easily accessible.

Another advantage of having a design pattern as an LDP is that one doesn't have to copy the design ideas anew each time a design pattern is applied in a new context. However this will only work when the intent of the design pattern is mirrored in the library version, and any application that uses the LDP automatically adheres to the intent of the design pattern. Seen from a modelling point of view it is of course just as good to copy the idea of the design patterns directly from [GHJV95], but this solution places a bigger demand on the designer of the application.

There are naturally also costs to pay when using LDPs. When placing a design pattern in a library as an LDP, this imposes a certain rigidness on any application in which the LDP might be applied. The design pattern will be *fixed*, in the sense that it will not be possible to adapt it in other ways than were foreseen when making the LDP.

Another disadvantage is the use of names in the LDPs. Having an abstract method declared in a class of the LDP with the name `anOperation` will enforce that the application using the LDP has to implement the method under the

name anOperation where the use of another name might have been more informative. This is however a small price to pay to have ready-to-use solutions available in a library, and a common problem for all who use functions from libraries.

The most obvious way of using a library of design patterns is by letting the classes in the application inherit from the classes in the LDP, as can be seen in Figure 8.2.

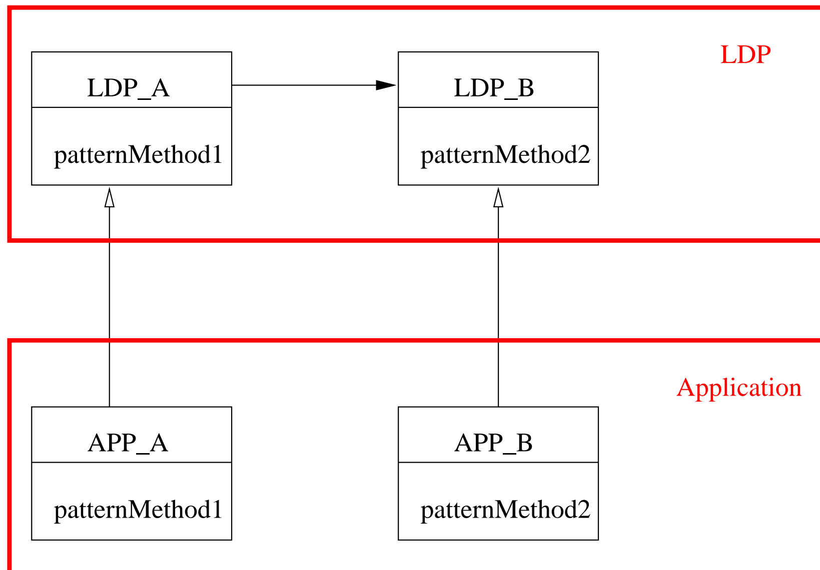


Figure 8.2: The immediate way of using a LDP

In languages without multiple inheritance this will cause problems whenever the classes in the application already inherit from other classes — either because they are part of existing hierarchies in the application or because they play roles from more than one design pattern. In the following subsection we show how the use of composition can solve this problem, provided that certain features are accessible in the programming language.

8.1.1 Simulating Multiple Inheritance by Composition

One of the advantages in using multiple inheritance compared to composition is that with a class inheriting from several other classes, where some of those have virtually declared classes or methods, it is possible to re-bind these.

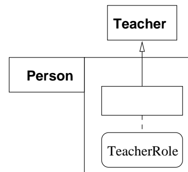
In BETA this advantage could also be achieved with composition by creating a singularly defined part object TeacherRole:

```
TeacherRole: @ Teacher(# ... extension ... #)
```

as an instance of a locally defined anonymous subclass of Teacher. An example of this and the structure in our expanded UML-notation, described in Appendix A is shown below.

```
Teacher:
(#
    Course:@ Text;
    Salary:@ Integer;
    giveGrade:< (# do inner; (*print Student, Course*) #)
#);

Person:
(#
    Name:@ Text;
    TeacherRole:@ Teacher
            (# giveGrade:: (# do ... ; (*print Grade*); #)
            #)
#)
```



Since giveGrade is defined as a virtual method in the class Teacher, and TeacherRole is a singular instance of a locally defined subpattern of Teacher it is possible to further bind giveGrade in TeacherRole. This way, the method giveGrade can be extended to serve the Person class better.

Using this kind of composition, designers can add roles to classes throughout the whole system development by nesting part objects containing roles from the Library design patterns into the application classes and still be able to gain from the virtual classes and methods in the LDPs.

8.1.2 Implementing the LDPs

In [AC97a] we have discussed how and to what extent the fundamental design patterns could be placed in a library of Design Patterns. In this section we show an example of these discussions to illustrate what we believe could be possible and profitable to keep in a library.

The classes in the applications using such a library are sometimes already subclasses of other classes in the application or play roles from one or more design patterns. Therefore, in the descriptions of the LDP's we assume that such a library is used in a language with multiple inheritance or the possibility to simulate multiple inheritance, because the use of LDP's will mean that the classes in the application inherit from the classes in the LDP.

The following discussions are based on the descriptions of the Design Patterns found in [GHJV95], and require the book at hand for full understanding.

Decorator

To make as much as possible of this design patterns behaviour inherent in the LDP, we cheated a little and combined it with the **Template Method** design pattern. By making the Operation in the Decorator class a template-method, we could make it a final binding and then the only thing an actual application has to supply is the extra behaviour and state of the concrete Decorator class in the actual application. If additional behaviour is wanted before the delegation of the Operation, the concrete Decorator class makes a binding of the addedBehaviourBefore method, if not, it is not further bound. The same goes for the addedBehaviourAfter method. These two methods thus holds the functionality of the Hook methods of the **Template Method** design pattern.

Figure 8.3 shows the structure of **Decorator**-LDP and how the motivating example can use this LDP in an application.

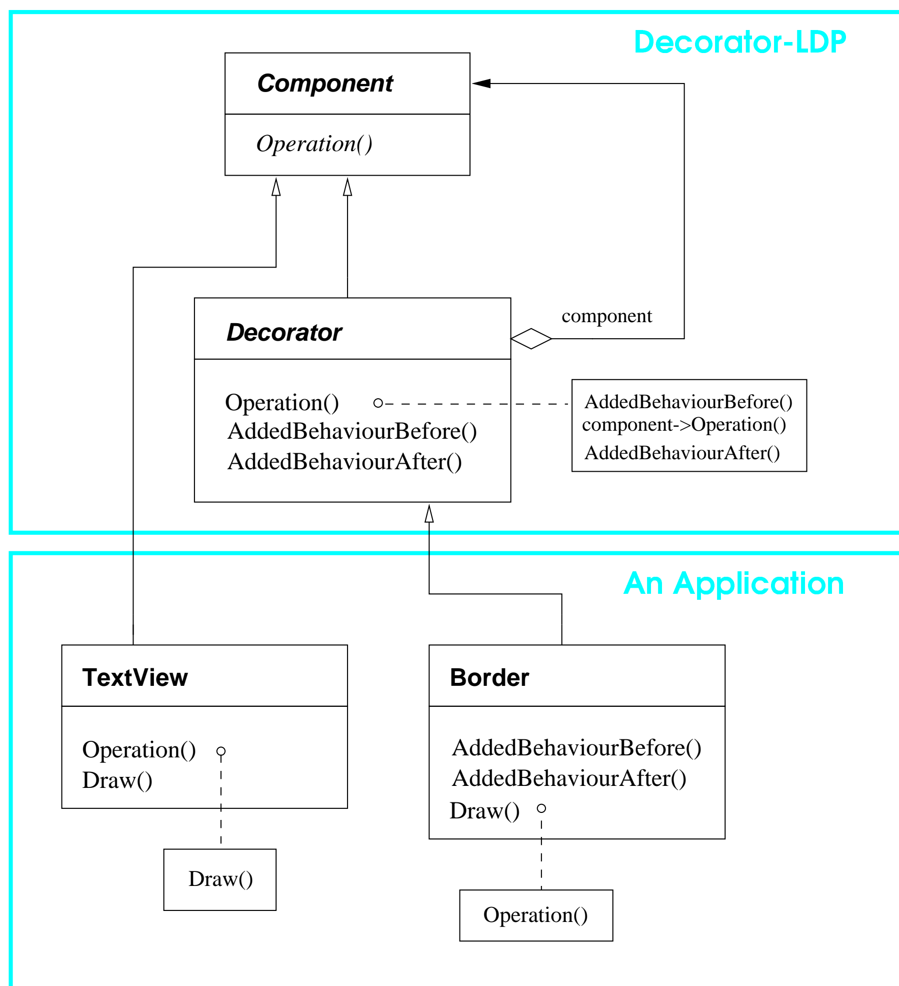


Figure 8.3: **Decorator**-LDP

In this way we believe we have captured most of the intent of the **Decorator** design pattern in the LDP; what is left to the user is now to make sure that

the addedBehavior is provided as well as that the application specific operation is implemented using the Operation from the Decorator as indicated by the diagram above.

Flyweight

The application-dependent issues to consider when making an LDP out of the **Flyweight** design pattern are the following:

- What kind of object is a key?
- How does a key identify a flyweight-object?
- How is the state of an object split into extrinsic state and intrinsic state?
- What operations should the flyweights support?

These considerations have led to a **Flyweight**-LDP as shown in Figure 8.4.

By having the LDPs FlyweightFactory declaring keyType as a virtual class and the procedure getFlyweight a virtual procedure it makes it possible for the concrete application to decide what key to use as well as to specify how that sort of key should identify a flyweight object. It is enough for the abstract FlyweightFactory to know that there is a key and a flyweight determined by the key to be able to maintain the pool of shared flyweights under the invariant that there is only one instance of each flyweight.

In the application of the LDP the flyweightType should be further bound to the class of *shared* flyweights, MyConcreteFlyweight, — it is thus guaranteed that each flyweight in the poolOfFlyweights has this type, which in turn guarantees that the IntrinsicState has been further extended in accordance with the concrete application.

We have chosen to have the abstract class Flyweight declare the classes ExtrinsicState and IntrinsicState since this separation is a fundamental property of a flyweight object. This will however mean that any application using the LDP will have to use the terms ExtrinsicState and IntrinsicState instead of more application-specific names. In the text editor example motivating this design pattern the extrinsic state could typically be the character's font, size and placement. The use of the LDP would here imply that these attributes should be nested into an extension of the virtual pattern ExtrinsicState. The operations possible to perform on the flyweight are not declared in the LDP, since the number of operations, as well as their names and parameters, will be specific to each given application. If the concept of *virtual lists*, as described in Chapter A, was incorperated in BETA, each operation could have been an element of such a virtual list, which would have made it possible to specify all of these characteristic properties in subclasses. However, since the declarations of the operations are not essential for capturing the intent of the design pattern in the LDP, the use of virtual lists is not indespensible in this context.

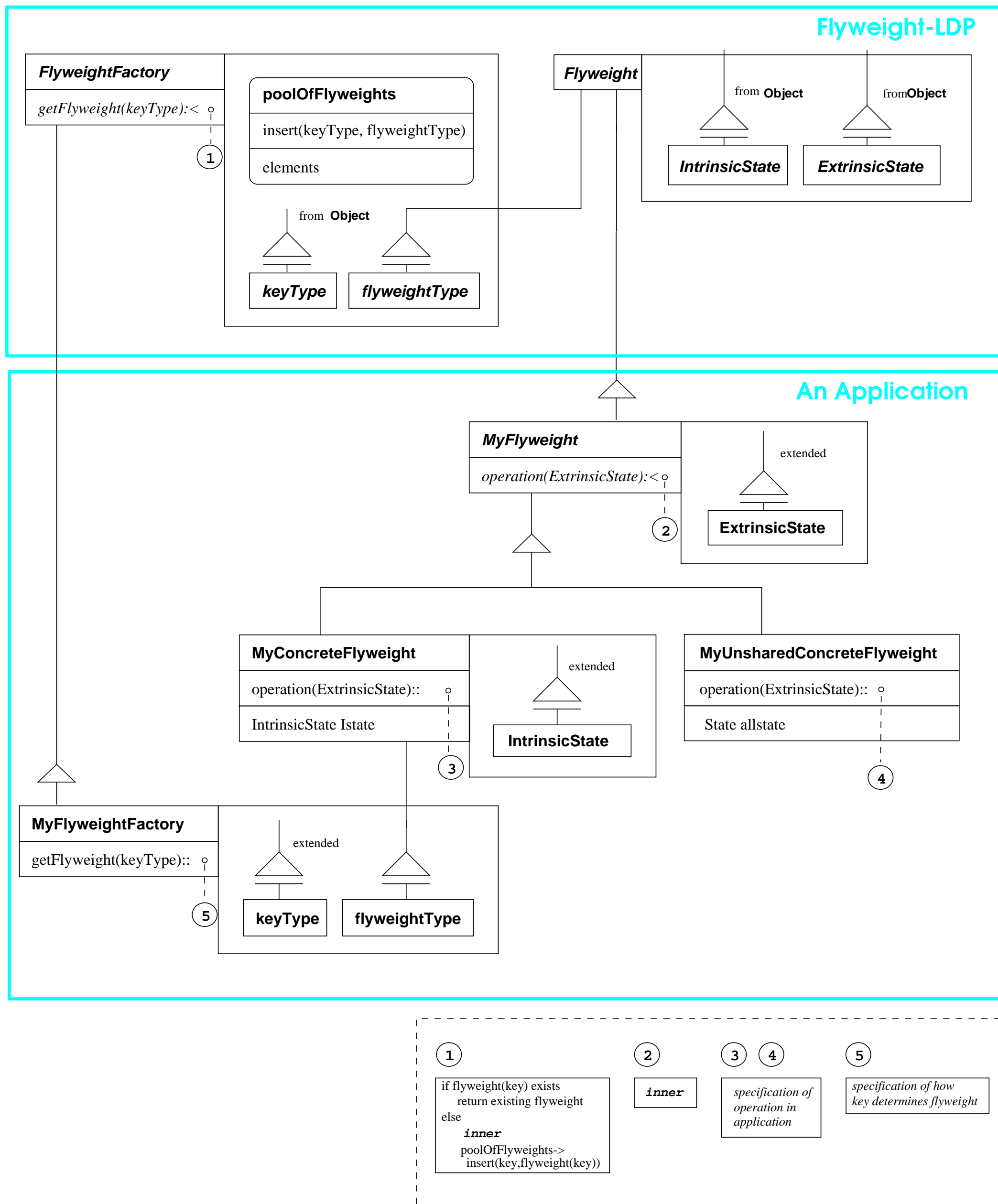


Figure 8.4: **Flyweight**-LDP

The advantage of having **Flyweight** as an LDP lies primarily in the FlyweightFactory class, where the use of virtual classes makes it possible to have an abstract implementation of the poolOfFlyweights even though the keyType and flyweightType is only known in the concrete application. This implementation of the poolOfFlyweights ensures that the intent of the design pattern is fulfilled whenever this LDP is applied in an application.

8.2 Related Work

The tracing problem has become a generally recognised problem within the field of design patterns.

Görel Hedin in [Hed97] proposes a technique for formalising design patterns which allows the design pattern applications to be identified in the source code. The technique is based on attribute grammars, and places a demand on the programmer that he explicitly annotates his program with design pattern roles. This has the benefit, that it will also enable *automatic checking*, i.e. it will be possible to decide whether a design pattern has been applied correctly. One main difference between this approach and ours, is that ours will partly reduce the implementation overhead, which Hedin's does not. Another main difference is that Hedin's solution can work as a debugger for design patterns, while our approach does not guarantee that the design patterns are applied correctly.

Jiri Soukop has also proposed a solution to the tracing problem. In his paper "Implementing Patterns" ([Sou95]) he proposes building a library of design patterns consisting of so-called *pattern classes*. A *pattern class* encapsulates all the behaviour and logic of the design pattern and the classes that form the design pattern in the application thus contain no methods related to the design pattern. What is left in the classes is only pointers and other data required for the design pattern. The problem of this solution is that all the hierarchical structure of the design pattern is lost, since everything is now contained as methods in the *pattern class*.

In their paper, "Programming Design Patterns" [FL00], Peter Forbrig and Ralf Lämmel describe an experimental programming language PaL, which provide language support for Design Patterns. They have implemented a library of design patterns written in PaL, and claim that the use of this library will provide the developer with traceable and flexible implementations of design patterns. Having the language support design patterns directly, by making a pattern-oriented programming model, is interesting, but it is my viewpoint that the patterns should be kept abstract, and that the programming languages should not be changed for them. The programming languages can be changed, because the solutions found in patterns can point to lack of power in the constructs, but it is dangerous to change the languages directly support the patterns.

8.3 Conclusion

Since the LDPs are reused by including roles in the classes of the application, as described in section 8.1.1, the use of LDPs would annotate where the design patterns were used in the application. This automatic annotation is a very important contribution to the documentation of software systems. It is in fact the precondition of the third advantage of using design patterns; "They enhance the documentation of software designs".

A number of the language features in Beta prove especially useful in connection with the LDPs by supporting genericity and reuse of models. This is further elaborated on in [AC97a] where we show how the intent of a design pattern could be kept in an LDP for 10 out of the 12 "true" design patterns, and that it in 6 out of these 10 cases is due to *virtual classes* and *nested classes*. The possibility of reusing enough of a design pattern to apply it from an LDP and still keeping the intent reduces the *implementation overhead*, a problem connected to the use of design patterns identified by Jan Bosch in [Bos97].

The fact, that it is possible to make a useful LDP out of a design pattern, proves that it is possible to make a reusable implementation of it. And since the design patterns in [GHJV95] formulate good design- or implementation-ideas, the language features that support them must be considered flexible and useful in relation to reuse of design.

Chapter 9

The Design Pattern Tool $\mathcal{DPDOC}$

One of the benefits gained from using design patterns is that they provide documentation at an abstract level. For a reader familiar with the design patterns it may be sufficient to simply write the name of a design pattern, e.g. “Decorator”, in order to document the functionality, the responsibilities, and even the motivation behind a whole subsystem of classes. This was adressed in the last chapter, where the tracing problem was solved by use of a design pattern library. Unfortunately, a solution to the tracing problem is not enough. It happens all too often that the system is changed according to some new requirements while the documentation stays as is. For example, a design pattern may be introduced without recording it in the documentation. It is also common to make changes without consulting the documentation which may lead to the introduction of inconsistencies in the code. For example, adding a new concrete decorator, but failing to implement the delegation in the intended way, can cause existing code to work incorrectly.

9.1 Providing Better Documentation

If the design patterns were incorporated in the programming languages as language constructs, this problem could be solved by the compiler. Actually making language constructs out of all the design patterns is infeasible, though, for two main reasons. Firstly, as also argued in Chapter 5, it can be worthwhile to make new language constructs out of some design patterns, but certainly not all of them since programming languages should be kept simple. Secondly, we expect design patterns to evolve, both due to the discovery of new design patterns but also when it is discovered that some design patterns are alike or applications of one another.

In [Hed97] it is described how it might be possible to create a tool that enables designers to get the benefits of language constructs when using design patterns, but without actually changing the programming language.

The approach is based on the idea of viewing a pattern application as a language construct which identifies the program elements that play the particular roles in the pattern, and which specifies the rules that these program elements must follow. The design patterns must first be projected onto the roles and the

rules of the design patterns. The roles can be of two types: *defining roles* and *derived roles*. The only classes, methods, and variables that must be explicitly marked in the code are the elements playing the defining roles. The other elements playing roles in the design pattern can be automatically derived from the defining roles, hence the name "derived roles". This work could possibly be eased by looking at the specifications of design patterns made by Amnon Eden [Ede00].

We will use the **Decorator** pattern from the GoF book as an example. The problem described in **Decorator** is that a designer wants to extend the functionality of a class, but finds subclassing too inflexible. The suggested solution is to use a separate class for each piece of extending (or "decorating") functionality and to aggregate objects of these different decorator classes as needed. An advantage of this pattern is that the extended functionality of an object can be easily added or removed at run-time, and the design is thus more flexible than subclassing. A disadvantage is that the design results in many small decorator objects and thus may be more space intensive than a solution based on subclassing. Figure 9.1 shows the class diagram for the suggested solution. *ConcreteComponent* is the class whose functionality is to be extended. The different extensions, "decorators", are modelled by the classes *ConcreteDecoratorA*, *ConcreteDecoratorB* , and so on; all of them specialisations of the abstract class *Decorator*. The top class in the hierarchy, *Component*, models any component in this system, i.e. either a concrete component or a decorator. The extension of a component is accomplished by connecting a decorator to it and requiring the decorator to delegate all operations to the component (in addition to their added behaviour). Several concrete decorators can be attached to a concrete component in a cascade structure.

In the **Decorator** design pattern the defining roles would be: *Component*, *Decorator* and the *decorated component* (the reference to a *Component* object from the *Decorator*). The roles derived from these defining roles would be: *Concrete Component(s)*, *Concrete Decorator(s)*, *Operation(s)*(the abstract operations in *Component*) and the *Implementation(s)* (the implementations of the *Operations* found in the *Decorator* and the *Concrete Decorators*)

9.2 The Tool

In this section we present the implementation of a prototype tool, $\mathcal{DPDOC}$, which supports documentation with design patterns based on the ideas discussed in [Hed97]. The implementation technique builds on the use of *reference attributed grammars*, RAGs [Hed99]. RAGs are stronger than conventional AGs because they allow references between nodes in the abstract syntax tree and thereby support the easy specification of non-local dependencies. The reference attributes can be freely used to access non-local information. For example, to specify name analysis, identifier uses can be directly connected to their declarations and used for type checking. For object-oriented languages, the class hierarchy can be specified explicitly by reference attributes between the classes and used for many different kinds of analyses. For the documentation of design

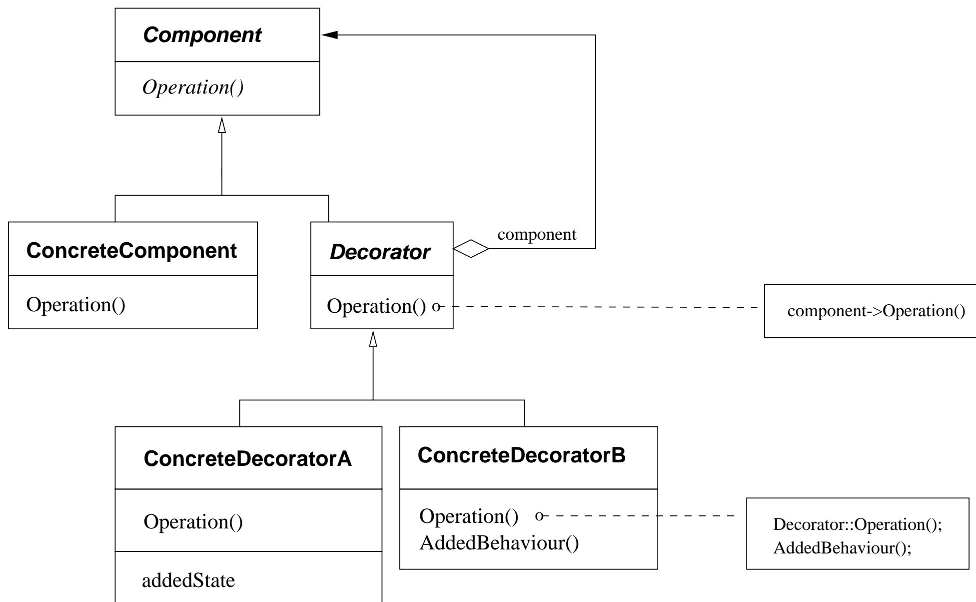


Figure 9.1: The **Decorator** design pattern structure

patterns used in a program we represent the design pattern instances explicitly and use reference attributes to connect these instances with the program code. This explicit representation of design pattern instances differs from the implementation approach suggested in [Hed97] and substantially improves the specification as will be discussed in further detail in Section 9.8.

$\mathcal{DPDOC}$ is based on (and built upon) an interactive language development tool, APPLAB (APPlication language LABoratory) [BHN99], which supports specification by means of RAGs. APPLAB is primarily used to test grammars for new languages while they are being developed. Users can edit both programs and grammars at the same time, thus making APPLAB a highly interactive and flexible environment for language design.

APPLAB supports editing and static-semantic analysis of programs written in a subset of Java called *PicoJava*. PicoJava includes key object-oriented constructs like classes, subclassing, and methods and some basic constructs like statements, variables, and types. PicoJava has been implemented in APPLAB with the aim of providing a platform for experimentation with analyses of object-oriented languages without including all the details of a full-blown language [Hed99]. Since $\mathcal{DPDOC}$ is built upon APPLAB it consequently supports the use of design patterns in PicoJava programs. This is sufficient for a prototype, since PicoJava contains all the language constructs necessary to apply the design patterns in the GoF book.

The implementation approach used in $\mathcal{DPDOC}$ separates the design pattern specification from the programming language specification, with only a narrow interface between them. This makes it possible to replace PicoJava with another (complete) object-oriented language without having to change the specification of the design patterns!

The design patterns are specified and implemented in the same way as lan-

guage constructs: they are modelled by a grammar with syntactic and context-sensitive rules which specify the roles and rules of the available patterns. This allows the rules of the applied design patterns to be checked, similarly to the way a compiler performs compile-time checks. The tool is able to, based on a few pieces of information from the user, automatically bind elements in the program to roles in an applied design pattern. The user provides the names of the classes (or other constructs) that play the defining roles in the pattern, and the tool then finds the remaining derived roles and checks the correct use of the pattern. The design pattern grammar is split into several modules in order to support extensibility so that the set of available design patterns can be easily customised to the designer's needs. In particular, a set of general modules are provided that specify common aspects of the design patterns, thereby constituting a specification framework for the design patterns.

Currently, the tool supports the following patterns from the GoF book: **Decorator**, **Composite**, **Visitor**, **Observer**, **Mediator**, **Adapter**, **Bridge**, **Chain of Responsibility** and **Factory Method**. These patterns were chosen because we wanted to test different aspects of the tool. Some of them were chosen because they are challenging to check rules for. Some because they were almost alike and we wanted to show the difference in the documentation even though the design patterns had the same structure. Some because we wanted to investigate how easy it was to extend the tool with patterns that were more or less like the existing patterns in the tool.

Several pattern tools, both commercial and academic, have been developed in the last few years. The developers of these tools have focused on different characteristics.

The task of making the use of patterns visible in the code, is called *the tracing problem* [Sou95], [Bos97]. It is desirable for pattern tools to help solve this problem by improving what we shall call *pattern visibility*, enabling the user to see the use of the patterns in the code. Such tools can be helpful in the documentation of the system. Empirical results from documenting design patterns in code have showed that pattern documentation can speed up program changes as well as improve their quality [PUP97]. These tests were performed with "static" handwritten documentation of the code with patterns. With a tool to automatically maintain the documentation, we claim that these results could be even better. *Rule checking* is another characteristic found in some tools. The tool developers have interpreted the abstract solutions in the design patterns to find the rules that must then be abided by, when the patterns are used and the tools are able to check whether these rules are followed. These checks can catch a certain class of errors that would otherwise call for extensive writing of test programs. We expect such checks to result in improved code quality and increased programmer's confidence in the code, similar to that reported for automated testing as in eXtreme Programming [Bec99].

DPDOC helps pattern visibility and performs rule checking and it has a functionality not found in other tools: *Role derivation*. The tool is able to, based on a few pieces of information from the user, automatically bind elements in the program to roles in the pattern. *DPDOC* also differs from other tools in its language-based approach to specifying the patterns which we claim provides

a general and precise basis for building this kind of tools.

9.3 Implementation

In this section, we describe the architecture and the implementation of $\mathcal{DPDOC}$ and discuss why it is beneficial to implement the prototype in APPLAB by specifying the design patterns in RAGs. The user interface for the application programmer and the $\mathcal{DPDOC}$ programmer is described in relation to this.

9.3.1 Application- and $\mathcal{DPDOC}$-programmer interface

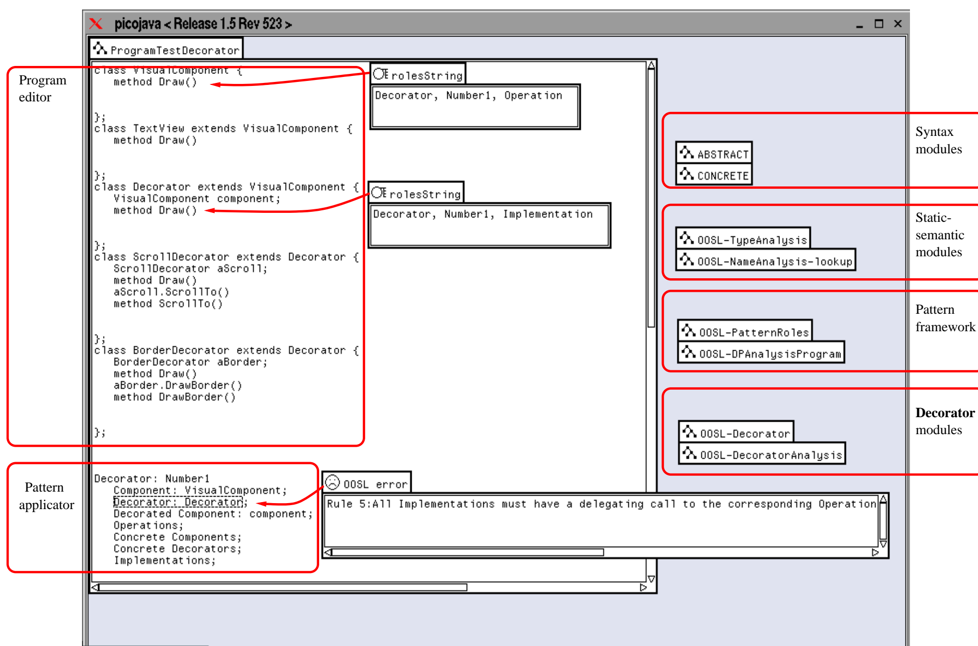


Figure 9.2: The user interface

Figure 9.2 shows the application programmer's view of $\mathcal{DPDOC}$ and a selected set of the grammar modules defining the underlying programming language and the pattern application language. The arrows and rounded boxes are not a part of $\mathcal{DPDOC}$, but additional information drawn on top of the screenshot to emphasise the different parts of the user-interface and the grammar modules.

The application programmers view is the window called *ProgramTestDecorator*. It contains two parts: a program editor and a pattern applicator. In the program editor, the application programmer simply uses the features of the editor (in this case APPLAB) to write the programs. Below the program editor, we find the pattern applicator.

The editor and the pattern applicator together show a program on which the application programmer has applied the **Decorator** design pattern. Let us

look at how this was done. After having written a program, the application programmer chose the design patterns they wanted to apply, in this case the design pattern **Decorator**. There are now four parameters to supply: the name of the pattern instance and the names of the elements in the program playing the three defining roles. The application programmer called the pattern instance *Number1* and let the *VisualComponent* class play the role of the *Component*, the *Decorator* class play the role of the *Decorator* and the *component* variable play the role of the *component* variable (called DecoratedComponent in *DPDOC*). In supplying the names of elements playing the defining roles, there are three possibilities. Either the names could be typed, or they could be chosen from a menu containing all user defined names in the program, or the *Names* menu could be used, which lets the application programmer choose from a selected set of names. The *Names* menu is described in detail in Section 9.5.

For the *DPDOC* programmer that maintains the tool, the right part of Figure 9.2 is important. It shows the names of 8 grammar modules, each defining its own part of *DPDOC*. These are the modules that should be changed when a *DPDOC* programmer wants to add new design patterns, modify design patterns or change *DPDOC*s underlying language. The way to extend the tool with more design patterns is described in Section 9.4 and the way to change the underlying language is mentioned in Section 9.3.2.

In the "Syntax modules" box we find two grammar modules: "ABSTRACT" and "CONCRETE". They define the abstract and the concrete syntax of the programming language and the design pattern application language. The "ABSTRACT" module defines the language constructs in the underlying programming language, what patterns the tool supports, and what roles these patterns have.

Figure 9.3 shows a selected set of the grammar modules for *DPDOC*, and for each of these, a selected set of rules. (In this figure, the syntax modules (ABSTRACT and CONCRETE) have been partitioned into different modules for the design patterns and the programming language. The tool currently supports only a single ABSTRACT and a single CONCRETE module, but in future versions we plan to support multiple syntax modules.)

The "Static semantic modules" box in Figure 9.2 shows a sample of the grammar modules defining the static-semantic analysis of the programs; type checking and name analysis in particular. The grammars make use of reference attributes to describe and make use of non-local dependencies. For example, in the *TypeAnalysis* module in Figure 9.3, a reference attribute *decl* in *Use* nodes refers to the appropriate declaration node and is used to define the *tp* attribute. (These aspects are discussed further in [Hed99].)

The modules *PatternRoles* and *DPAnalysisProgram* in the "Pattern framework" are part of the pattern specification framework, described in detail in Section 9.3.3. This framework contains the implementation of the core functionality in the pattern specifications and is easy to extend with more design patterns. The module DPAnalysisProgram contains definitions that constitute the core of the documentation: For each element in the program, the semantic rules update the string attribute *rolesString* to contain which roles the element plays in which design patterns. For each element, the end-user can enquire

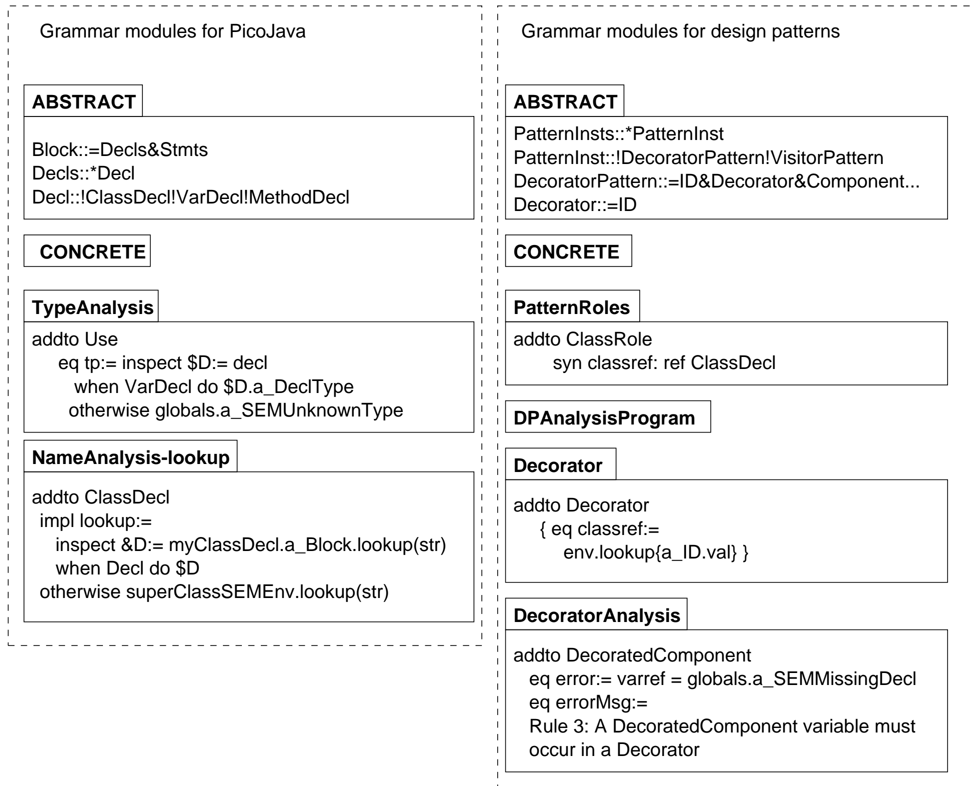


Figure 9.3: Excerpts from the grammar modules

which roles it plays, as shown in Figure 9.2, where the roles of two elements have been queried.

The modules Decorator and DecoratorAnalysis specify the **Decorator** design pattern. In the Decorator module, the roles for the design pattern are specified. The DecoratorAnalysis module defines the rule checking. An example of how the rule checking works is shown in Figure 9.2. The error box on the right side of the pattern applicator displays a message that warns the end-user that they have broken one of the rules in the **Decorator** design pattern in his implementation. They can now choose to change the code according to the warning, so that the documentation follows the code or they can choose to ignore it. $\mathcal{DPDOC}$ will not try to change the code according to the rules of the design pattern.

The pattern application code and the user program code are connected by means of reference attributes. An example of this is shown in Figure 9.4 which shows a part of the syntax tree for the program and pattern applications of Figure 9.2. Here, the pattern part of the syntax tree contains an application of the **Decorator** design pattern, and the syntax node for the role *Component* has a reference *classref* which refers to a ClassDecl node in the program part of the syntax tree, in this program to the declaration of the class VisualComponent. This reference attribute is automatically computed according to the program and pattern grammar modules.

When the program is edited, *DPDOC* automatically builds the attributes for every node in the syntax tree, according to their definitions in the grammar. For example, for each expression in the program, an attribute *tp* containing the type information is defined, as exemplified in the TypeAnalysis module in Figure 9.3. Likewise, the elements in the pattern applications are attributed according to the grammar. For example, when the user edits the name of an element in a pattern application, the tool locates the corresponding class declaration in the program, according to the attribute definition in the **Decorator** module in Figure 9.3.

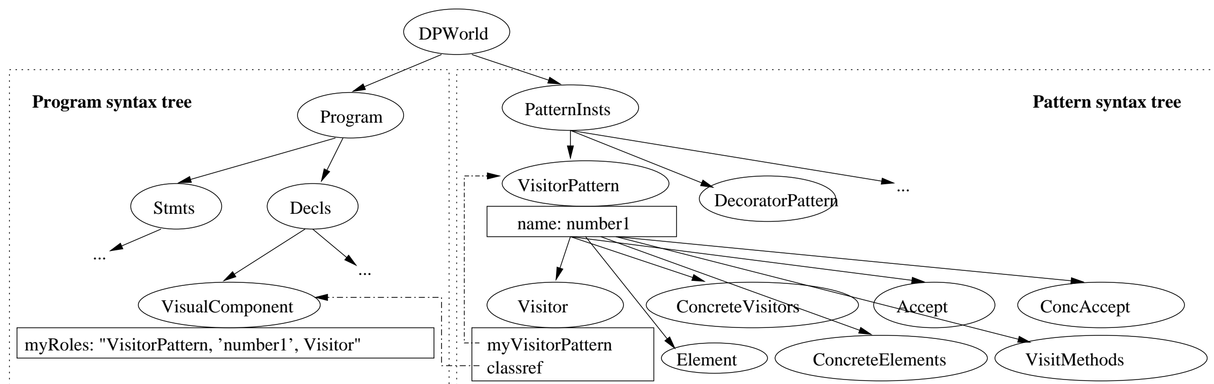


Figure 9.4: Reference attributes connect the design pattern applications to the program

9.3.2 The interface between the programming language and the design patterns

Figure 9.4 shows how the syntax tree for the pattern applications is connected to the syntax tree for the program by means of reference attributes: the *classref* attribute connects a class role like Component to the corresponding class declaration (VisualComponent in the example). To be able to declare the reference attributes, the pattern applicator grammar needs to have some knowledge of the programming language grammar. For example, the *classref* attribute has the type ClassDecl which is a production in the programming language grammar. Currently, *DPDOC* uses only the language constructs from Java that can also be found in most other object-oriented programming languages such as classes, methods and variables. The interface between the two grammars is thus narrow, and can be satisfied by most OO languages. Therefore, it is possible to change the programming language to another OO language without having to change the design pattern grammar. However, the new language may have additional constructs, e.g. multiple inheritance. To use these in the design patterns it is necessary to extend the interface accordingly.

The interface also contains attributes and functions defined by static-semantics module. This includes the environment attributes based on the block structure of the language and the lookup function of the name analysis. These attributes and functions are used for locating the appropriate declarations for class names

and other names used in the pattern applications.

9.3.3 Pattern specification framework

RAGs are an object-oriented specification formalism where the non-terminals and productions are viewed as a class hierarchy; non-terminals correspond to superclasses and productions to subclasses. The pattern specification framework defines a number of classes capturing general aspects of design patterns, and which can be subclassed for the individual design patterns. Figure 9.5 shows an example of extending the framework to define the **Decorator** design pattern. The superclass *Role* defines the core functionality of a role, e.g. the name of the role. In this prototype, we have chosen to support six different kinds of roles: a single class, several classes, a single method, several methods, a single variable and several variables. Each of these kinds of roles has given rise to a class in the framework with their own default functionality common for this particular kind of role, e.g. the *classref* reference in the *ClassRole*.

In Section 9.1 we mentioned that we had chosen to implement certain design patterns in $\mathcal{DPDOC}$ in order to see how easy it was to extend the tool with more or less similar design patterns. Using the framework as is, it was equally easy to implement design patterns no matter how much they differed from the design patterns already implemented. In a future version of the tool, when the framework is extended based on the similarities in "families" of design patterns it will be even easier to extend with more design patterns from the same "family" as design patterns already implemented in $\mathcal{DPDOC}$.

One of the main functionalities in $\mathcal{DPDOC}$ is providing the user with textual descriptions of which roles the elements of the program plays. The object hierarchy resulting from this is shown in Figure 9.4 (the pattern syntax tree). Most of this functionality is provided in the framework, and it is therefore very easy to add design patterns to $\mathcal{DPDOC}$, simply by adding subclasses to the framework *Role* classes.

9.4 Extension of the tool

To extend $\mathcal{DPDOC}$ with a design pattern, four steps must be performed:

- Extend the abstract syntax with the structure of the design pattern, i.e. the roles, and extend the concrete syntax with the end-users view of the design pattern. It is important to use the pattern specification framework by subclassing the role classes with the roles in the design pattern in order to benefit from the functionality found in the pattern specification framework.

- Extend the names facility module with the rules for naming the defining roles. This rarely amounts to much if the framework is used, since almost all of the defining roles can reuse the functionality placed as default by the framework. And most of the roles that can be restricted are already placed in the category "deriving roles".

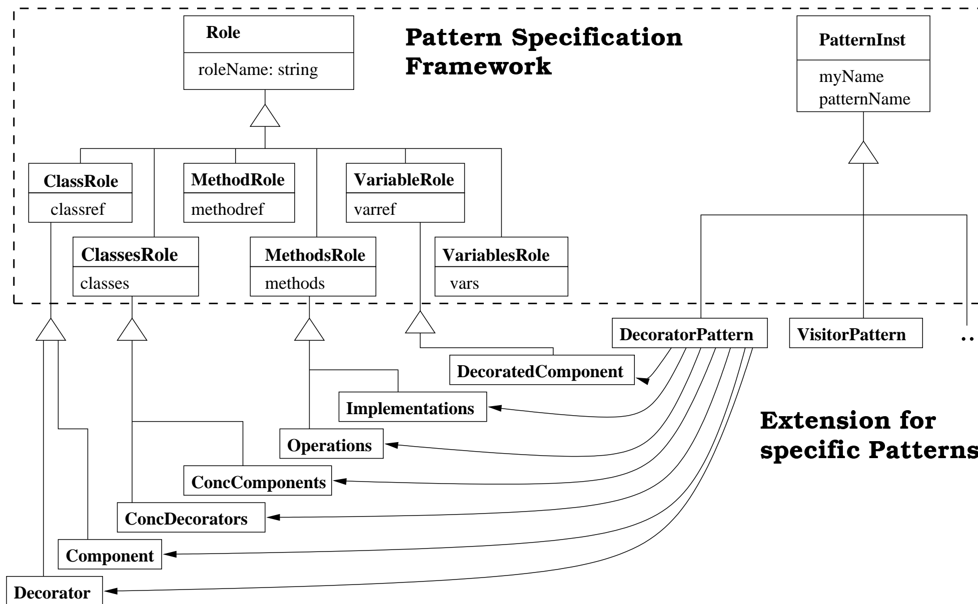


Figure 9.5: The pattern specification framework

- Extend *DPDOC* with a new module named after the design pattern. This module must contain the semantic rules for providing the references to the elements in the program. The defined roles are found by searching for the right names in the program. The references for the derived roles must be computed by the tool from the references in the defining roles. Again we stress that the use of the pattern specification framework eases this task.

- Extend *DPDOC* with a new module named after the design pattern with the suffix "analysis". This module must contain the semantic rules for checking that the rules of the design pattern are abided by in the program. The references found in the "roles"-module are followed to the program and the relationships between the elements playing the different roles are investigated. Since the rules in the design patterns differ a good deal, there is less support from the framework in this task.

An example of the modules defining the roles and the rules of a design pattern (**Decorator**) can be seen in Appendix B.

9.5 The Names Facility

DPDOC includes an editing facility called the *Names* menu: When the user wants to use the name of a class, method, or variable previously defined, a list is provided of names that are available at that place in the program. The list is constructed according to the scope rules of the programming language and is specified in the grammar. The *Names* facility is also used in the design pattern applicator in order to make it easier for the user to tie the defining roles to

names in the user program. Here, the list of names is generated according to the rules for the design patterns.

Suppose an end-user is applying the **Decorator** design pattern to his program. If the defining role *Decorator* has already been tied to the class *VisualComponent*, the *Names* facility can be used to advantage when annotating the *component* role with a name from the program. The *Names* menu will now list the variables declared in *VisualComponent*, in accordance with the rules for **Decorator**, and the user can select one of these.

Nonterminals	**Attributes and Semantic Rules**
Roles	↑ names: **ref NodeBag** names := **new NodeBag** **son** ID.names:= names
ClassRole	names:= env.visibleNames
MethodRole	↑ myBlock: **ref** *Block* names:= **foreach** $D: *VarDecl in* myBlock.visibleNames **do** $R := (**init new NodeBag**) $R.add($D)
operation	myBlock:= componentBlock

Table 9.1: Names module. The syntax is described in Appendix B

Table 9.1 gives a simplified, but essentially complete view of how easy it is to extend the *Names* facility in $\mathcal{DPDOC}$. The class roles have the classes in the program environment as name space. Methods and variables can as a rule only be chosen from specific classes playing a role. Some design patterns have roles that can extend this module, others not. Some design patterns (e.g. **Visitor** from the GoF book), with just defining roles, that are subclasses of the *ClassRole* will not give rise to an extension of the *Names* module.

9.6 Exploiting APPLAB

We chose to extend the already existing tool, APPLAB with the design pattern specifications in RAGs for these three major reasons:

Firstly, since our aim was to treat design patterns as language constructs, we had to use a tool, which could help us develop the language for describing the design patterns in an easy way. Since APPLAB is an interactive language laboratory, this tool was an obvious choice. APPLAB enables the language designer to see the consequences of design choices immediately. This interactive feature is also useful in the actual use of $\mathcal{DPDOC}$, because the roles played by the elements in the program are available at all times during the development of programs.

Secondly, the reference attributes provided by the RAG supports direct communication between nodes that are distant from each other in the syntax tree; a node c an access attributes of a distant node via a reference attribute. This is important because it enables nodes in the program part of the syntax tree to communicate directly with nodes in the design pattern part of the syntax

tree. Additionally, this support for non-local communication is beneficial for specifying the underlying object-oriented language which has plenty of non-local dependencies (e.g., inheritance).

Thirdly, the object-oriented specification notation and the modularisation of RAGs supported in APPLAB is beneficial in order to separate the specification of the underlying programming language from the specification of the design patterns. The narrow interface between these parts makes it possible to extend or change the programming language and the design pattern specifications almost independently from each other. In particular, support for new design patterns can easily be added by adding new grammar modules.

9.7 Future work on DPDOC

DPDOC is a prototype tool. It was (and is being) developed to study how a tool can support the use of patterns and documentation in the best way. We use it in our research to experiment and evaluate possible characteristics in pattern tools. DPDOC has been evaluated by students from *The University of Aarhus* with respect to both functionality and user-friendliness.

So far, we have focused the development of DPDOC on proving the viability of the language-based approach to integrating tool support for design patterns. An interesting way to continue the work might be to provide support for a more intuitive and graphical user interface. We have several ideas for how to do this within the language-based approach. One idea is to improve the user interface for connecting the pattern applications to the user program: As we have described in this paper, the connection is currently done by entering class names, method names, etc. in the pattern application code. A more intuitive way would be to support user-defined reference attributes that could be set by direct-manipulation, e.g. by clicking on the appropriate class or method in the user program.

We also intend to support navigation in terms of the pattern applications and roles. A simple extension to APPLAB to allow navigation along a reference attribute will allow us to specify navigational links, for example to navigate from the program code to the pattern application code or vice versa, or between different points in the program code that belong to the same pattern.

Another way of improving the user interface would be to provide visualisations of the pattern applications, e.g. in the form of UML-like diagrams similar to the solution diagrams in the GoF book. Visualisations can be integrated with the APPLAB system as described in [MH00].

An interesting area to explore would be that of generating design documentation from programs, similar to how many systems, e.g. JavaDoc [Jav] and the Eiffel *short* mechanism [Mey88] support the generation of API documentation. The strength of DPDOC in this setting would be that it is easy to specify the generation of different kinds of documentation from the program, either visual or textual. From a specification point of view, the main thing to improve is the modularisation of the syntax, allowing the syntax of the programming language and the pattern application language to be defined in separate modules. (This

change is in principle trivial to accomplish). In addition, it would be desirable to make the pattern specification framework independent of class names in the programming language grammar. For example, the framework currently defines a reference attribute *classref* which is a reference to a *ClassDecl* node, thus assuming that the programming language has a nonterminal/production named *ClassDecl* . It would be desirable to have a mechanism for decoupling this dependency, allowing the programming language to use any name for its nonterminal/production modelling class declarations. This can be accomplished in several different ways. We plan to use a mechanism inspired by the "part objects" of BETA [MlP92], but using some kind of multiple inheritance or Java-like interfaces would also be possible.

9.8 Related Work

RAGs were originally devised to support easy specification of static-semantics, in particular for object-oriented languages where there are more non-local dependencies than in other language paradigms. For example, RAGs allow the class hierarchy to be represented as explicit connections between syntax nodes. RAGs have also proven useful for a variety of other tasks, including specification of worst-case timing analysis [PH99] and generation of software visualisations [MH00]. In both these applications, the name analysis providing connections between identifier use and declaration sites forms a basis upon which the other analyses are built. This is the case also for the design pattern specification technique as suggested in this paper: the name analysis attributes are used in the specification of the connections between the design pattern instances to the corresponding locations in the program code.

As mentioned, our work builds on the earlier work reported in [Hed97] which also suggested using (reference) attributed grammars as a basis for design pattern specification. However, in this earlier work, the programming language grammar was simply extended with additional rules. Our present work using an explicit representation of the design pattern instances has several advantages: First, the specification can be factored in a natural way according to the different design patterns, whereas in the earlier approach, language constructs like ClassDecl had to contain rules for all possible design pattern roles, and many conditionals were needed to handle the different cases. Second, the explicit representation of design patterns allows much of the functionality to be captured in the specification framework of role classes which can be conveniently specialised for the different design patterns. Third, the explicit representation constitutes a reification of the design patterns which makes them concrete in the user interface. Furthermore, the earlier approach was never implemented, whereas our present approach provides an implementation for all the major design patterns in the GoF book.

Several other tools for design patterns, both commercial and academic, have been developed in the last few years. The functionality of the tools range from simple detection of the use of patterns in a program to rule checking tools with a functionality that can be compared to that of $\mathcal{DPDOC}$. In these tools, the

most interesting aspect to compare is the way the roles and the rule checking are specified. We also find the issue of extensibility and flexibility interesting. A thorough survey on tools supporting the use of patterns can be found in [Vil97]. Some tools, e.g. [Bro], perform pattern detection by pattern matching the code with the pattern structures to find instances of patterns in the code. These tools are not able to tell two patterns with the same structure apart and are little help in relation to documentation of the system.

Another group of pattern tools focus on code reuse, that is, these tools contain implementations of patterns, which can be glued into the users code, where they choose it. Examples are [Cod](containing a library of C++ templates), [Mod] (generating code in Delphi) and [FFVY96] (which automatically generates code in C++). [FFVY96] is a little more sophisticated than the first two, since it is possible to choose between various trade-offs in the pattern solution before the code is glued to the rest of the system. The focus in the development of these tools is code reuse, not documentation or rule checking.

A growing group of tools does perform rule checking, that is, they help the user use the patterns correctly by checking some rules made by the tool developers from the guidelines in the design patterns.

In "Monitoring Compliance of a Software System With Its High-Level Design Models" [SSC96], Sane et al. describe how their tool, "μchoices", can confirm whether the C++ implementation of a system maintains its expected design models and rules. The new approach seen in this paper is that the tool checks both statically and dynamically by both checking the code and the run-time performance. The static rule checking is specified by implementation of rules in Prolog. These rules are expressions of the authors' formalisation of patterns in positive evidence and violations. When the user claims that certain classes play certain roles in a pattern, positive evidence that they are in fact following the pattern is checked. If they are, the implementation is checked for violations of the rules. A Prolog inference engine matches the program data to the design model rules to produce relations that can demonstrate whether the rules in the patterns are abided by. By checking dynamically the tool can help to reassure that the rules are followed when the system is run even if the code appears to allow a deviation from the design. This approach exhibits advantages and disadvantages similar to those shown by languages with dynamic run-time type checking. There is no immediate support in "μchoices" for changing the underlying programming language to something else than C++ or extending the number of patterns, for which rule checking is supported. It is possible that the system would be able to derive roles in the program, as DPDOC does it, because of the way the rule checking is implemented, but it is not described how it could be done.

In [KB96] Kim and Benner describe a C++-oriented tool, POE (Pattern Oriented Environment), which can help designers implement patterns in the intended way. The design pattern instances are represented as lists of instances of *Class*. The *Class* type contains the constraints connected to the use of it. Among these, the *Properties*, which are the relations and operations connected to the *Class* instance. When a design pattern is instantiated, the user can make bindings between the design pattern classes and the user defined classes. POE

implements validation algorithms to ensure that different pattern instances and role bindings are used properly. It is possible to extend the tool with more patterns using the *Class*, *Relations* and *operations* types. Thus the means to specify patterns is limited to these types.

FRED (FRamework EDitor) as described in [Vil98] is a development environment especially designed for framework development and specialisation in Java. A pattern is described using templates, which are sets of possible implementations with constraints that all its instances must conform to. One limitation of FRED is that constraints only apply to method and field signatures, data types and larger constructs, and thus can not verify that code within a method body is valid. The implementation of the role binding functionality is influenced by POE.

Our tool, $\mathcal{DPDOC}$, makes the use of design patterns visible on a detailed level, where every element in a program has the possibility to play a role in a design pattern. Additionally the tool perform rule checking, not just in the moment when the design patterns are applied, but consistently through the work on the program. The current version does not support code reuse or design pattern detection, but has a functionality not mentioned above: *Role derivation*. All support is specified in RAGs and the user can add support for new patterns as needed. The specification framework makes such additions comparatively easy. Also unlike other tools, the $\mathcal{DPDOC}$ approach keeps a narrow interface to the programming language and can be re-targeted to other languages. The other tools are all targeted towards a specific language, some of them including vendor-supplied ports to a few languages.

In the paper: "Tool Support for Object-Oriented Design Patterns" [FMvW97], Florijn et al. describe their pattern tool. The tool is built on a fragment model described in [Mei96], which allows the elements that can be used in the tool to be fragments and not just classes and methods. Fragments can represent not only the syntactic elements of a language, but also associations and inheritance relations and therefore provide numerous possibilities for describing relations between roles in design patterns and elements of a program. In the implementation the tool specifies the roles and rules of the design patterns as fragment collections of small fragments with associations implemented in Smalltalk. The way the tool is extended with a new design patterns is by reusing these types of fragments to build the design patterns with roles and rules using fragments and associations. To make an instance of a design pattern found in the tool, the specification of the design pattern (the graph of fragments) is cloned. The rule checking is expressed as compositions of predicates defined on fragment types. Much like in $\mathcal{DPDOC}$ some fragments have query operations, that checks whether e.g. a class playing a certain role contains a method playing a certain role. [FMvW97] also supplies support for rule-checking after the code is glued into the program.

This tool is the one which comes closest in functionality to $\mathcal{DPDOC}$, but nevertheless they differ in some important points. Firstly, the ability to let other elements in the program than classes and methods play roles in design patterns is mirrored in our tool. But $\mathcal{DPDOC}$ enables the $\mathcal{DPDOC}$-programmer to let *everything* that can be represented as a node in an abstract syntax tree play a

role in the design pattern, because the design patterns are specified in their own grammar module which can contain references to any kind of node in an abstract syntax tree. Secondly, *DPDOC* is developed to be language-independent, which makes it more flexible than [FMvW97], which can only be applied to Smalltalk programs. Last, but not least, the rule-checking in [FMvW97] is only done on the roles that are hard coded into the program by the user. The system is not able to derive which other roles from the design pattern are played by elements in the program, which is a feature in our tool. And to the best of our knowledge ours is the only tool with this feature.

9.9 User evaluation of the tool

To get a picture of the usability of the tool, a group of students at the University of Aarhus were asked to evaluate *DPDOC*. The evaluation was conducted by only three students and was thus, by no means, a complete evaluation of the tool. The result was a user guidance and evaluation of the tool [ErNM00].

In short their conclusions were mainly positive. They found the tools ability to derive roles to be a very clever and useful property, since it made the documentation of the code with patterns very effective and made it easier to remember how to use the patterns. The possibility of using different patterns at the same time in the same code was also received in a very positive way. When the user changes the code in a way that does not follow the rules of the design patterns applied, the tool will warn the user, but not automatically change the code to abide by the rules. The students found this “mildness” dangerous. It was their concern that the users who carelessly used the tool, would make things worse by being able to document code in a misleading way. Our answer to that is that firstly it was not our intention to make a conservative system, it was our intention to make a system, that would help the programmer, not make things harder, and secondly the warning given by the system is still better than nothing at all as status quo is now.

There was one aspect of the tool, that they disliked; the user interface. They found it very frustrating. This was of course partly due to the fact that they were new to the system, but also, as we must admit, due to the fact that the system is more user polite, than really user friendly. Redesign of the user interface is clearly a major part of the future work of *DPDOC*.

9.10 Conclusion

DPDOC is a working prototype of a tool which can make it safe and easy for beginners to use design patterns and automatically maintain valid documentation of software. The tool supports design pattern visibility, rule checking, and automatic role derivation. *Design pattern visibility* is supported by reifying design pattern applications into explicit language constructs and allowing program elements to be tied to the reified design pattern roles. This allows the user to see explicitly which design patterns are applied in the code, and what roles are played by the different elements. This documentation of the

applied design patterns is tied directly to the code via reference attributes, and the documentation does therefore not become out of date when the program is changed. Automatic *rule checking* is supported by checking that the rules for applying each pattern are abided by. Finally, *role derivation* is supported by automatically deriving many of the roles in a design pattern application, based on a small set of defining roles.

The tool is built on APPLAB and therefore inherits some functionality from this, like the *Names* facility, the syntax-directed editing and static-semantic check according to the grammar. Since we have developed $\mathcal{DPDOC}$ as an extension of APPLAB by specifying the design patterns as language constructs in a grammar, it makes use of the built-in static-semantic analysis to check a program. This way the rules for using design patterns are implemented exactly as rules for using language constructs and $\mathcal{DPDOC}$ works as a *lint* for design pattern applications. Because we have used RAGs it is possible to specify the design patterns in a very detailed and thus, very precise manner.

The object-oriented nature of the grammar formalism allows common aspects of the specification to be factored out into a pattern specification framework, making the addition of support for new patterns simple.

Since the architecture of the tool consists of two largely separate parts, the programming language and the design pattern applicator, with a narrow interface between them, it would be possible and even relatively easy to change the underlying programming language and the tool would be immediately useful for other languages than PicoJava.

By using modularisation and RAGs in the development of $\mathcal{DPDOC}$ we have created an extensible and strong prototype of a design pattern tool.

Chapter 10

System Development Revisited

The work previously described is based on the way of developing software described in Chapter 4. This chapter describes two different perspectives on programming, and compares them with the way of programming related to the tool $\mathcal{DPDOC}$. A short description of tools built on literate programming and aspect-oriented programming is included.

10.1 Literate Programming

Despite many efforts to reduce the number of failed projects in the field of software engineering, it has not been possible to improve the situation [BMB+98]. In the past years many things have been developed such as faster computers, larger memory, type-safe languages, easy but type-unsafe languages. Developers and programmers can be as fast, reliable and good as their experience and tools let them, but one of the problems remaining is to make them communicate their decisions to others or themselves a few months later. As long as human beings need to be able to read code it has to be possible to document it in a clever and easy way. By clever is meant: more elegant than a document attached to the code, and by easy: as automatic as possible.

Although written in 1984, Knuth's statement: "I believe the time is ripe for significantly better documentation of programs."[1] is still valid.

The design pattern tool, $\mathcal{DPDOC}$, lets the user see the documentation together with the program without having to manually insert comments in all different pieces. There are references in the program to documentation, i.e. the GoF book. When writing code in $\mathcal{DPDOC}$, the user has the opportunity to use the design ideas described in design patterns to improve the code and the documentation. The tool guides the user to write code, that follows the rules of the design patterns. Additionally, it maintains the documentation semi-automatically by, based on the information given by the user, deriving which elements in a program play which roles in which design patterns. Although this sounds like the answer to all the problems we face in system development, it can be worthwhile to look at other attempts.

[1] Donald Knuth "Literate Programming" [Knu84],p. 99

Donald Knuth proposed literate programming as a way to promote the attitude that it is our main task to concentrate on explaining to *human beings* what we want the computer to do, instead of just explaining it to the computer. In literate programming, the programs and the documentation are written together as literary works, and the code is just pieces of explaining lines, almost like comments. To illustrate that the theory of literate programming works, Knuth implemented a tool, the WEB[2] system, to prove his point.

As can be seen in Figure 10.1, the WEB tool takes as input a special WEB program containing both the documentation and the code, written together by the programmer. It weaves the documentation parts of it into TEX-code, with the lines of code neatly crossreferenced together in the right places. In parallel it tangles the code parts into Pascal code, readable by a Pascal compiler.

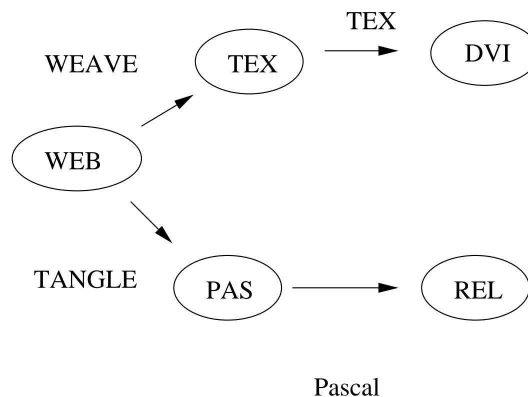


Figure 10.1: Dual usage of a WEB file

The WEB program written by the programmer has to be written following a special syntax, not surprisingly resembling LATEX a lot. It is possible to write the code top-down or bottom-up or both ways, and in as many small pieces as the programmer fancies, because the code will be tangled together by WEB in the right way, when the syntax is followed.

Comparing WEB and *DPDOC* my focus is on the ability to apply, and document with, design patterns.

In short, there is no automatic checking in WEB that can keep the code consistent with specific design rules, e.g. the rules of the solutions of design patterns. And since WEB is passive and not active it is not able to derive roles in the program. On the other hand it is possible to write documentation in a readable way, around every tiny bit of the program, whereas in *DPDOC*, the only support for documentation is references to documentation of design patterns.

10.1.1 Example - Elucidative Programming

Elucidative programming is a perspective on programming based on literate programming. The motivation for developing it, instead of merely using Knuth's

[2]He chose the name WEB partly because it was one of the few three-letter words of English that hadn't already been applied to computers :-)

ideas of literate programming is described by Kurt Nørmark in [rma00]:

> *It is our hypothesis that Literate Programming is beyond the reach of the average programmer. The ambition of literate programming is too high, the program artifacts are too far from mainstream, and current programming environments will suffer too much if adapted to the 'literate ideas' in a WEB-like elaboration*
>
> — Kurt Nørmark in "Elucidative Programming"

This hypothesis has led to elucidative programming, which preserves the idea of documented program understanding from literate programming, but is reworked in different ways. The source program is left intact and not spread over the documentation sections and the program understanding is described in a document which is firmly related to named constituents in the source program.

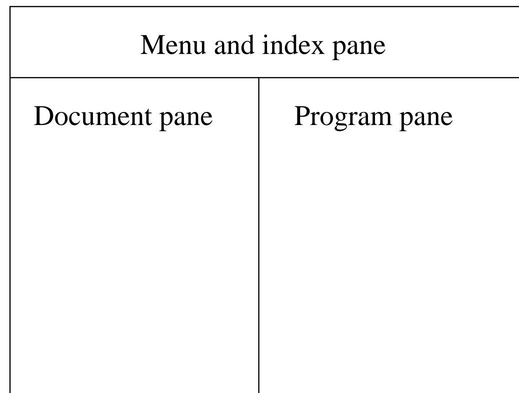


Figure 10.2: The navigational proximity between documentation and program.

The navigational proximity in elucidative programming tools illustrated in Figure 10.2 is weaker than the physical proximity found in literate programming, but it holds other benefits. Firstly, it makes it possible to describe several program entities in one section of the documentation, which can be necessary with for example design patterns. Secondly, a single program entity may be discussed in several documentation sections.

Compared to $\mathcal{DPDOC}$, the Java elucidator described in [rma00] is a more polished, but weaker tool. It enables the user to write his documentation in the tool, with references, not only from design patterns, to the code. This makes the tool useful for describing the code with other things than design patterns[3]. On the other hand, the author himself poses the following questions related to the usefulness of the elucidative tools:

- Is it realistic to expect that programmers retain their program understanding in a free style story or essay?
- Is it possible to keep the documented program understanding up-to-date and valuable during the maintenance phase of the program.

[3]Which, I have to admit, can be useful.

To the first question, I am tempted to give a "no" as an answer, but as long as a few good programmers can be trusted to do it, it is still good to have an elucidative tool to keep the documentation linked to the code for other developers to read. The second question is answered by the facilities of $\mathcal{DPDOC}$, which automatically keeps the documentation up-to-date.

10.2 Aspect-Oriented Programming

In this section $\mathcal{DPDOC}$ is compared with AspectJ, an aspect-oriented programming tool. Aspect-oriented programming provides a divide-and-conquer solution to the engineering challenge of software systems. The challenge lies in managing the complexity of the systems, although the human mind is incapable of handling it all at once. The basic solution is separation of concerns in design, code, and implementation. This works fine in theory, since the use of objects and components can help modularise the system and keep it modularised homogeneously throughout analysis, design, and coding. Some design concerns are difficult to modularise in an elegant and consistent way, though:

- system-wide error-checking strategies
- design patterns
- synchronisation policies
- resource sharing
- distribution concerns
- performance optimisations

The code addressing these concerns tends to be spread out across the system. Because these concerns won't stay inside of any one module boundary, we say that they crosscut the system's modularity.

Imagine the situation (found in [KHH+01]) depicted in Figure 10.3, where two classes share a common concern.

One obvious solution is to modularise the system even more, to split the existing classes or components into different bits and pieces, each with their own concern. The problem with this is that the system no longer resembles the design, and that it becomes harder to understand and maintain it.

The solution to this problem lies (according to Kiczales) in aspect-oriented programming. The paper "An Overview of AspectJ´´ [KHH+01] describes the concept of an aspect as a well-modularised crosscutting concern. AspectJ is a set of several aspect languages extending Java making it possible to program control over "emergent entities´´ using aspects. Emergent entities do not exist explicitly in the component model or code, but rather arise during execution, e.g. a performance optimisation.

Technically, what is done is that an aspect weaver is created that takes the classes and the aspects as input and preprocesses it to code. Of course one could also build it as a compiler with binary code as output or as an interpreter.

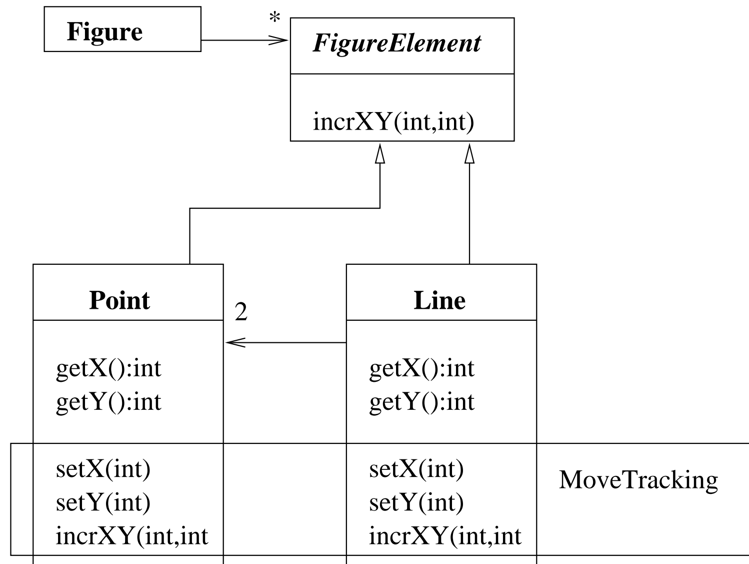


Figure 10.3: The box named MoveTracking shows an aspect that cross-cuts methods in the Point and Line classes.

AspectJ is a compiler-type aspect weaver, that can be used to improve the modularity of software systems. Programming with different carvings of the system allows:

- clean separation of programming of functionality
- programming of control over emergent entities.

AspectJ adds constructs to Java that enable the modular implementation of crosscutting concerns. This ability is particularly valuable because crosscutting concerns tend to be both complex and poorly localised, making them hard to deal with.

This approach of being able to have different views of code, is also dealt with in $\mathcal{DPDOC}$, where different applications of design patterns can be seen from a specific pattern view to the code. The question that suggests itself is whether $\mathcal{DPDOC}$is already aspect-oriented.

Although AspectJ can be used in a way that allows AspectJ code to be removed for the final build, aspect-oriented code is not always optional or non-functional. In the current version of $\mathcal{DPDOC}$ there is not added functionality to the program by annotating it with design pattern applications. It could be possible to rebuild the system to actually add extra functionality to the code, if needed. This would work as if the code was annotated with library design patterns (LDPs), with the implementation from the LDPs glued into the code in the different aspects. The possibility of supporting aspect-oriented programming with APPLAB, and in effect, $\mathcal{DPDOC}$, has also been investigated by Görel Hedin and Eva Magnusson in [HM01].

Chapter 11

Conclusion

In this chapter a short summary of the thesis is provided together with a discussion of future work and a conclusion of the dissertation.

11.1 Summary

In this dissertation many aspects of patterns are covered and solutions to some of the problems with applying design patterns are proposed. First the concept of patterns within software is introduced. After that the background for patterns, described both by the history of patterns and the philosophy behind them is given. Then the stage of system development is set to enable a discussion of the concept of patterns within software development. Thereafter, a short description of six different categories of patterns is given, to provide the reader with a glimpse of the diversity of patterns. A case study describing the use of patterns in connection with each other is provided to give a feeling of the possibilities of using different kinds of patterns together.

After thus having covered the concept of patterns, the work on the problems with design patterns is described. The main problems are the tracing problem and the large number of patterns, that destroys the possibility of getting an overview of patterns. The tracing problem is the problem of tracing the applications of patterns. It is often impossible to see, which patterns are applied where in the code. First, a classification is provided that will make it easier to get an overview of the design patterns almost regardless of the number of discovered design patterns. The design patterns are divided into three categories: Fundamental design patterns, related design patterns and language-dependent design patterns. The design patterns can then be divided into families, thus making the finding of the right design pattern an easier task. Then a proposal for the solution to the tracing problem is given. The core of the solution is a library of design patterns, in which the core of each design pattern is implemented as library design pattern, an LDP. Unfortunately, the solution depends on very strong language constructs, and the idea is thus refined later in the dissertation. There, a new solution, based on a software tool, $\mathcal{DPDOC}$, is provided to solve the tracing problem and in addition make it simple and safe to apply patterns.

In Appendix A is described the proposed extensions to the Beta programming language and the UML notation needed to describe the ideas in the dissertation.

11.2 Future Work

Future research can be imagined in a number of the areas covered in this dissertation. A few of the main areas of development are described below.

Alexander's new ideas

For Christopher Alexander, unlike for software developers, the concept of patterns was new more than 20 years ago and he has developed his ideas somewhat since then. James Coplien presented in August 1997 Christopher Alexanders latest work entitled: "The Nature of Order" to the Chicago Patterns Group. Brad Appleton took notes from the talk and it is based on these I present future work in relation to Alexanders new ideas.

In **The Nature of Order**, Alexander has further developed his notion of the *quality without a name* and instead uses the term *wholeness*. Wholeness emerges as a result of naturally occurring processes which are based on loci known as *centres*. A *center* is "something we notice" as a structure; something that captures our eye. Patterns are examples of such centres.

> *Patterns are configurations of centres that serve some context. Whereas patterns have names and are only a point in the namespace of the system, centres are truly nameless and deal with deep structures in the human psyche and beyond*
>
> — James Coplien, "On the Nature of *The Nature of Order*"

Based on experiments with the objective sense of beauty, Alexander has found the presence of fifteen fundamental properties: Levels of scale, strong centres, alternating repetition, boundaries, positive space, good shape, local symmetries, deep interlock and ambiguity, contrast, graded variation, roughness, echoes, the void, simplicity and inner calm and not-separateness. Since Alexanders work on patterns triggered such a useful and needed concept in software development, it is tempting to hope for more inspiration from that source and some, like James Coplien, have already started studying his latest work.

Extending the classification

Currently the classification of design patterns deals with the 23 from the GoF book. Of these only 12 passed as fundamental design patterns, and since the patterns in the GoF book are some of the most general of the design patterns, it is most likely that an extension in the number of the design patterns classified will give an even smaller percentage of fundamental design patterns.

Fine-tuning *DPDOC*

Naturally, as with the classification of design patterns, it would be worthwhile to extend the tool with more design patterns. Another important improvement would be to make *DPDOC* user friendly as opposed to its current state of user politeness. This can be done in a number of ways, as described in Section 9.7.

Patterns and software development

Many who take it as their responsibility to explain the way to apply patterns in system development, leave speculation as to the role of patterns in software development behind them and simply apply patterns, wherever they can be applied. In Chapter 4, this issue has been investigated. There it is claimed, that patterns are not in the same category as techniques, technologies, methods or processes, but that they are orthogonal to them. But perhaps this point bears further discussion: In a sense patterns are both more and less than orthogonal. They are more then orthogonal, because they infiltrate all other parts of system development. The aspect of pattern philosophy have a taste of a lightweight process, like extreme programming, the literary aspect resembles a technique, the pattern tool aspect is related to technologies and the application of a single pattern can be said to be a method. They are less than orthogonal, because they do not, together with traditional development, span an entire field.

It could be both interesting and beneficial to investigate more thoroughly the role of patterns in system development.

11.3 Conclusion

The software pattern movement is still in its infancy. Supported by the growing experience with patterns throughout the field, I am convinced that patterns can be used to improve the understanding of good software development. Everybody who wants to become skilled within a field, has to learn the tricks of the trade, and patterns provide a good way of doing so. The research presented pursues the goal of making it easy to find the right pattern for the task, apply it, and document the code appropriately.

11.3.1 Patterns in Development

The claim proposed in Chapter 4 was that software patterns actually do make life better for developers or software architects. This was based on the argument that good design solutions can be said to induce elegant code, that is easily readable and comfortable to work with. If code is elegant, it can be argued that it improves the human condition of the developers and thereby posesses the quality without a name. The developers are the ones who step into the code like people step into a house. When designing or coding, the developer walks into the design model or the program and can feel comfortable or not. If the design or code has the quality without a name she will feel better than if it does not.

11.3.2 The Classification

In Chapter 7 the objective was to use classification to help regain the benefits of using design patterns.

To preserve the benefits of design patterns, the design patterns were partitioned into fundamental design patterns, language dependent design patterns and related design patterns, with the fundamental design patterns fully providing the benefits. Only 12 of the 23 design patterns from [GHJV95] were classified as fundamental design patterns. From this it was concluded that it is beneficial to have a critical approach to design patterns, because it minimises the number of fundamental design patterns and thereby makes the area of design patterns easier to get on top of.

11.3.3 The Library of Design Patterns

In Chapter 8, it is described how the use of library design patterns, LDPs, can preserve the design patterns in a library, and how the use of these would guarantee automatic annotation in a program of the objects participating in an application of a design pattern.

Thus the use of LDPs will enhance the documentation of software systems, and it will to some extent eliminate the implementation overhead if the chosen implementation language posesses the necessary language abstractions.

11.3.4 The Design Pattern Tool $\mathcal{DPDOC}$

In Chapter 9 a working prototype of a tool, $\mathcal{DPDOC}$, is described, which can make it safe and easy for beginners to use design patterns and automatically maintain valid documentation of software. The tool allows the user to see explicitly which design patterns are applied in the code, and which roles are played by the different elements. This documentation of the applied design patterns is tied directly to the code, and the documentation therefore doesn't become out of date when the program is changed. Automatic *rule checking* is supported by checking that the rules for applying each pattern are abided by. Finally, *role derivation* is supported, automatically deriving many of the roles in a design pattern application, based on a small set of defining roles.

11.4 Summa summarum

With this dissertation in hand, developers can extend their tool box to include patterns and with some practise master the application of them.

Appendix A

Extensions of Beta and UML

In this appendix we will describe some proposed extensions to the object-oriented language BETA described in [MBMP93]. We will also describe an extension of the UML notation, that we find useful in connection to our work with design patterns.

A.1 Proposed extensions of Beta

In the analysis of some of the design patterns described earlier in this thesis it is investigated whether the possibility of having a list whose elements are virtually declared would be advantageous for the design pattern in question. In this section the prerequisites for having such lists in BETA are analysed.

First the concept of *virtual nonterminals* is introduced; then it is demonstrated how these can be used for simulation of virtual patterns as known in BETA_{97} (the version of BETA available in 1997) and finally the concept of a *virtual-list* is introduced.

A.1.1 Virtual Nonterminals in Beta

In [KMMPN83] a generalisation of the *virtual* concept is proposed that would make it possible to parameterise a pattern with any syntactic category of the language. This generalisation is called *virtual nonterminals*. The idea is that whenever it is syntactically correct to insert a string S as derivable from the nonterminal X, it would also be syntactically correct to insert the virtual nonterminal ⟨name: X⟩ instead of S. If a virtual nonterminal ⟨name: X⟩ is inserted in a pattern P it would then in subpatterns of P be possible to bind name to a string of category X.

As an example of the use of virtual nonterminals consider a while-loop with a conditional exit, where the following parameters should be deferred to subclasses:

- The condition for keeping the loop running.
- The "preprocessing" of the current element of the loop if a such is needed.
- The condition for leaving the loop.

- The label to which the program execution should jump when exiting.
- The selection for choosing the next element in the loop.

Using virtual nonterminals this could be implemented in the following way:

```
P: (#
    While:
    (# do
        Loop:
        (if ⟨Cond: Evaluation⟩
        // True then
            ⟨PreImp: Imperative⟩;
            (if ⟨On: Evaluation⟩
            // True then leave ⟨Exit: Label⟩
            // False then
                ⟨PostImp: Imperative⟩;
                restart Loop
            if)
        if)
    #)
  #)
```

The while-pattern might then in subclasses of P be extended to the following use:

```
subP: P
    (#
      L:
      ⋮
      While': While
          (#
            Cond:: aList.NonEmpty;
            PreImp:: aList.Current → Key;
            On:: Key = Subject
            Exit:: L
            PostImp:: aList.Next
          #)
    #)
```

An ordinary specialisation of a pattern Q as it is known from BETA$_{97}$, would, when using virtual nonterminals, be declared in the following way:

```
Q: (#
    Attribute-part AT: Specification of the attributes
    ⟨virt_AT: Attributes⟩
    Action-part ACB: Specification of actions before possible extension
```

```
        ⟨virt_AC: Action-Part⟩
        Action-part ACA: Specification of actions after possible extension
    #)

  Q1: Q
    (#
        virt_AT::  Specification of extending attribute-part
        virt_AC::  Specification of extending action-part
    #)
```

This would eliminate the need of the keyword inner since all it now takes is a virtual nonterminal of the type ⟨name: Action-Part ⟩. Furthermore it makes it possible to cover the situation where more than one inner is wanted, by simply having several virtual nonterminals of the above type. The drawback of this solution is however that it must be explicitly stated whenever a class can be subclassed by providing the virtual nonterminals.

A.1.2 Simulating Virtual Patterns Using Virtual Nonterminals

A virtual pattern, V, as known from BETA_{97}, can be declared in a pattern P and further bound in subpatterns of P in one of two ways:

A. V is declared by an existing class in the system:

   V:< Q,

   where the extension is made by binding V to a subclass of Q; Q1:

   V:: Q1

   The classes Q and Q1 are assumed to be described as in Section A.1.1.

B. V is declared anonymously by extending an existing class in the system with an object-descriptor:

```
V:< R(#
        Attribute-part AT: Specification of the attributes
        Action-part ACB: Specification of actions before possible extension
        inner
        Action-part ACA: Specification of actions after possible extension
      #),
```

   where the extension is made out of an anonymous class determined by an object-descriptor:

```
V:: (#
       AT1: Specification of extending attribute-part
       AC1: Specification of extending action-part
    #)
```

Using virtual nonterminals it is directly possible to simulate a virtual pattern as shown in the following:

A. The declaration of Q is made using virtual nonterminals, and the specialisation of Q into Q1 is done by binding these as shown in Section A.1.1. The obvious solution when declaring V is therefore to let V be the empty extension of Q. Thereby V has the same virtual nonterminals as Q, which in a subpattern of P can be bound to the same terminals as in Q1:

```
P: (#
      V: Q(# #)
   #)

subP: P
   (#
      V.virt_AT:: Q1.virt_AT
      V.virt_AC:: Q1.virt_AC
   #)
```

It is necessary to use the scope operator to indicate that it is the virtual nonterminals of V that are to be bound, since it could be possible for P to declare several virtual methods qualified by Q. In subclasses of P it should then be possible to extend these differently, which is done by binding the various virtual nonterminals differently.

B. The declaration of V is made so that it is possible to extend R further by adding attributes and extending the action-part. The virtual nonterminal of the action-part is placed where it should be possible to extend V's behaviour (analogue with inner):

```
P: (#
      V: R
         (#
            Attribute-part AT: Specification of the attributes
            ⟨virt_AT: Attributes⟩
            Action-part ACB: Specification of actions before possible extension
            ⟨virt_AC: Action-Part⟩
            Action-part ACA: Specification of actions after possible extension
         #)
   #)
```

And the binding of V in subpatterns of P will be done in the following way:

```
subP: P
   (#
      V.virt_AT::   Specification of extending attribute-part
      V.virt_AC::   Specification of extending action-part
   #)
```

A.1.3 Virtual Lists

In [KMMPN83] it is furthermore proposed that it should be possible to declare a list of virtual nonterminals; a so-called *virtual-list*. The notation for this is proposed as

$$\{\delta\}\textit{i-list}\{\gamma\}$$

which means that it can derive to $\delta\{\gamma\delta\}^*$, where the asterisk means zero or more occurrences.

δ and γ are arbitrary BETA language elements, and δ may contain zero or more indexed virtual nonterminals of the form ⟨name(i): X⟩.

Since a virtual nonterminal can simulate a virtual pattern, this proposal implies the possibility of having a list of virtual patterns as a language construct in BETA.

Our proposal for a syntax to describe a list L of virtual patterns declared by the pattern Pattern is as follows:

```
L:< [eval] Pattern
```

eval must evaluate to a nonnegative number, but since lists in BETA are dynamic, they can be extended arbitrarily in subclasses and the exact value of eval will therefore only be useful seen from a modelling point of view.

In the light of the discussion from Section A.1.2, a list of virtual patterns can be simulated using virtual-lists as follows:

A. The elements in the list L are declared by an already existing pattern Q:

```
P: (#
     L:< [eval] Q
   #)
```

and further bound by subclasses of Q:

$$\mathsf{L[1]::\ Q1}^{s_1} \quad \mathsf{L[2]::\ Q1}^{s_2} \quad \ldots \quad \mathsf{L[n]::\ Q1}^{s_n}$$

where each $\mathsf{Q1}^{s_i}$, $1 \leq s_i \leq m$ for $1 \leq i \leq n$, specifies one of the possible m specialisations of Q.

Using virtual-lists this can be simulated as follows:

```
P: (# {⟨name(i): Names⟩: Q(# #)
      } i-list {}
   #)
```

Q and its subclasses $\mathsf{Q1}^{s_i}$ are as described in Section A.1.1.

In subclasses of P the bindings of the nonterminals are then made according to the specific demands in subP, with n being the size of the list as determined in the given subclass and the (not necessarily distinct) superscripts s_1, s_2 or s_n denoting one of the m possible subclasses of Q.

```
subP: P
    (#
        name(1)::  Name of virtual pattern 1
        name(1).virt_AT:: Q1^s1.virt_AT
        name(1).virt_AC:: Q1^s1.virt_AC
        name(2)::  Name of virtual pattern 2
        name(2).virt_AT:: Q1^s2.AT
        name(2).virt_AC:: Q1^s2.AC
              ⋮
        name(n)::  Name of virtual pattern n
        name(n).virt_AT:: Q1^sn.AT
        name(n).virt_AC:: Q1^sn.AC
    #)
```

B. The patterns of the list are declared by an existing class in the system, that has been extended with an object-descriptor:

```
P: (#
    L:< [eval] R(#
                    Attribute-part AT: Specification of the attributes
                    Action-part ACB: Specification of actions before extension
                    inner
                    Action-part ACA: Specification of actions after extension
                #)
    #)
```

and are in the n entries of the list L in subP further bound by anonymous extensions as shown in Section A.1.2 with attribute-parts and action-parts $AT1^1$ and $AC1^1$ through $AT1^n$ and $AC1^n$.

Using virtual-lists the corresponding declaration of the virtual patterns would be as follows:

```
P: (# {⟨name(i): Names⟩: R
            (# Attribute-part AT: Specification of the attributes
                ⟨virt_AT: Attributes⟩
                Action-part ACB: Specification of actions before extension
                ⟨virt_AC: Action-Part⟩
                Action-part ACA: Specification of actions after extension
            #)
        } i-list {}
    #)
```

In subclasses of P the binding of the nonterminals is then made according to the specific demands in subP:

```
subP: P
    (#
        name(1):: Name of virtual pattern 1
        name(1).virt_AT:: Specification of extending attribute-part
                              corresponding to AT1^1
        name(1).virt_AC:: Specification of extending action-part
                              corresponding to AC1^1
        name(2):: Name of virtual pattern 2
        name(2).virt_AT:: Specification of extending attribute-part
                              corresponding to AT1^2
        name(2).virt_AC:: Specification of extending action-part
                              corresponding to AC1^2
              ⋮
        name(n):: Name of virtual pattern n
        name(n).virt_AT:: Specification of extending attribute-part
                              corresponding to AT1^n
        name(n).virt_AC:: Specification of extending action-part
                              corresponding to AC1^n
        #)
```

Above we have demonstrated how virtual nonterminals can simulate virtual patterns and how virtual-lists can simulate lists of virtual patterns.

Rooted in this fact as well as in the observation that the modelling aspects of the two concepts are alike, we will in the rest of this paper make little or no difference between the two concepts *virtual-lists* and lists of virtual patterns.

A.2 Extended UML notation

Since we are evaluating the design patterns with with respect to language constructs out of the ordinary, e.g. virtual classes, we have been forced to expand the UML notation to support certain features from BETA.

The features that necessitate an expansion of the UML-notation are the following:

- Nested classes
- Virtual classes
- Singular objects

A.2.1 Notation for Nested Classes

We have in the diagram notation for nested classes to a large extent copied the notation presented in [Ein97].

A class P containing a nested class N will be depicted in the following way, with the boxes depicting the classes P and N following the ordinary rules for depicting classes in the UML notation:

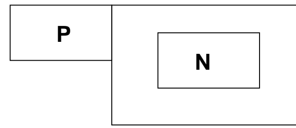


Figure A.1: Nested classes

As P can be as specialisation of another class, N can likewise be a specialisation of some class SuperN. This will be depicted as shown in Figure A.2.

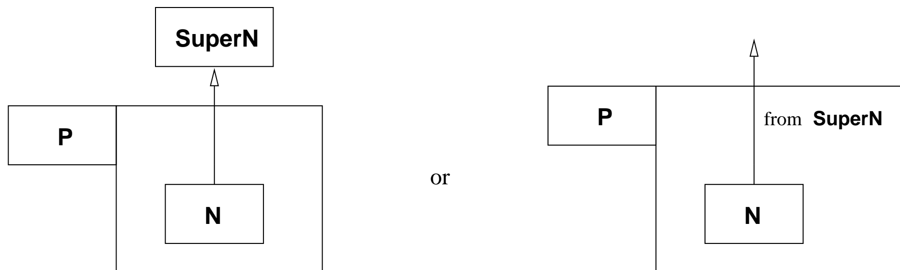


Figure A.2: Inheritance in nested classes

The notation in the diagram to the right in Figure A.2 is meant to provide a shorthand notation for making large diagrams easier to understand, or for making it possible to split these into fragments where not all classes are present.

A.2.2 Notation for Singular Objects

A singular object in Beta will always be described by an object descriptor, which introduces an anonymous specialisation of an existing class (in the trivial case this class will be the class Object). The notation for singular objects will therefore be the standard UML notation for objects generated from classes; rounded boxes as the objects, ordinary boxes as the originating classes, and dashed lines between the classes and objects to indicate the generation of the objects. The difference from the ordinary generation of objects will be visualised by omitting the name from the generating class; thereby indicating that it is an anonymous object descriptor instead of a named class. Since the anonymous object descriptor will effectively describe a specialisation of some class, this will permit further bindings of virtual patterns from the superclass, so the superclass will have to be visible in the notation. The notation will consequently be as shown in Figure A.3.

A.2.3 Notation for Virtual Classes

The concept of virtual classes is implemented in Beta ([MBMP93]) and has been proposed as an extension to Java ([Tho97]).

A virtual class VP may be declared in a class P in three different ways:

A. VP is qualified by an existing class in the system, V, and is further bound in a subclass of P to a subclass of V:

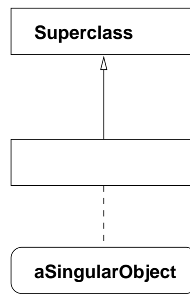


Figure A.3: Singular Objects

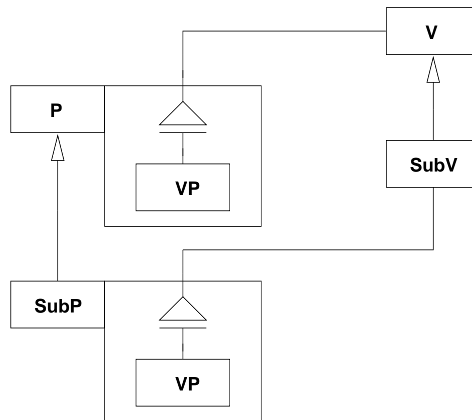


Figure A.4: Further virtual bindings in subclasses

B. VP is qualified by an existing class in the system, V, and is in a subclass of P further extended by directly adding the code, thereby obtaining an anonymous class:

C. VP is in P described directly in an anonymous class and is further extended in a subclass of P by adding code directly:

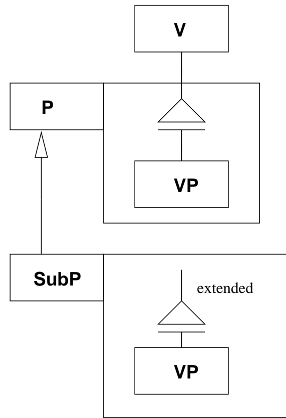


Figure A.5: Virtual classes; anonymous extension

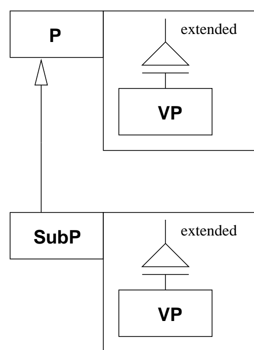


Figure A.6: Anonymous virtual classes

Appendix B

The Specifications of Decorator in DPDOC

This appendix shows some example specifications. Tables B.1–B.2 correspond to the grammar module OOSL-Decorator of Figure 9.2 and show the specification of the roles of the **Decorator** pattern. Tables B.3–B.4 correspond to the OOSL-DecoratorAnalysis module and show the specification of the rules for the pattern.

The tables should be read as follows:
The left column shows the name of the node being extended and the right column describes the attributes and their equations. The nodenames are *slanted* and the keywords are in **boldface**. The attributes can be of three types; synthesized declared by use of an $\uparrow$, inherited (in the attribute grammar sense, not in the object-oriented sense) declared by use of a $\downarrow$, and local declared by use of a $\rightarrow$. Synthesized attributes are used to propagate information upwards in the syntax tree, inherited attributes are used to propagate information downwards in the syntax tree and local attributes are just used within the node. The values of the attributes can be defined by equations, which are described by: "attribute name" := "expression".

Nonterminals	**Attributes and Semantic Rules**

DecoratorPattern	**son** Roles.myPattern := **this** *DecoratorPattern*
	myName := a-ID.val
	patternName := “Decorator”
	son Roles.env := env
	son Roles.globals := globals
	a-DecOperations.myComponent := a-Component.classref
	a-ConcComponents.myComponent := a-Component.classref
	a-ConcDecorators.myDecorator := a-Decorator.classref
	a-ConcComponents.myDecorator := a-Decorator.classref
	a-ConcDecorators.operations := a-DecOperations.methods
	a-ConcComponents.myConcDecorators := a-ConcDecorators.classes
	a-DecoratedComponent.DecVar := a-Decorator.classref.a-Block
	a-DecImplementations.Implementations2 :=
	    foreach $M: *MethodDecl* **in** a-DecOperations.methods **do**
	    $R := (**init new NodeBag**)
	    $R.union(a-ConcDecorators.findMethod($M));
	a-DecImplementations.Implementations1 :=
	    foreach $M: *MethodDecl* **in** a-DecOperations.methods **do**
	    $R := (**init new NodeBag**)
	    $R.union($M.matchListForMethod
	    (a-Decorator.classref.a-Block.a-Decls)));

Component	classref := env.lookup(a-ID.val) roleName := “Component”

Table B.1: Specification of roles for the **Decorator** design pattern, part 1

Nonterminals	**Attributes and Semantic Rules**
ConcComponents	↓ myDecorator: **ref** *ClassDecl* ↓ myComponent: **ref** *ClassDecl* ↓ myConcDecorators: **ref NodeBag** classes := myComponent.bagOfSubclasses.diff(myConcDecorators.add(myDecorator)) roleName := “Concrete Component”
Decorator	classref := env.lookup(a-ID.val) → TempBag: **ref NodeBag** roleName := “Decorator”
ConcDecorators	↓ myDecorator: **ref** *ClassDecl* classes := myDecorator.bagOfSubclasses ↓ operations: **ref NodeBag** findMethod: **func ref NodeBag** (M: **ref** *MethodDecl*) := **foreach** $C: *ClassDecl* **in** classes **do** $R := (**init new NodeBag**) $R.union(M.matchListForMethod($C.a-Block.a-Decls)) roleName := “Concrete Decorator”
DecOperations	↓ myComponent: **ref** *ClassDecl* methods := **foreach** $M: *MethodDecl* **in** myComponent.a-Block.a-Decls **do** $R := (**init new NodeBag**) $R.add($D) roleName := “Operation”
DecImplementations	↓ Implementations1: **ref NodeBag** ↓ Implementations2: **ref NodeBag** methods:= Implementations1.union(Implementations2) roleName := “Implementation”
DecoratedComponent	↓ DecVar: **ref** *Block* varref:= DecVar.lookup(a-ID.val) roleName := “Decorated Component”

Table B.2: Specification of roles for the **Decorator** design pattern, part 2

Nonterminals	**Attributes and Semantic Rules**
DecoratorPattern	a-Decorator.myImplementations := a-DecImplementations.Implementations1 a-Decorator.myDecComp := a-DecoratedComponent.varref a-Decorator.operations := a-DecOperations.methods

Decorator	↑ error: **boolean**
	error := error1 **or**
	(error2 **or**
	(error3 **or**
	error4));
	↑ errorMsg: **string**
	errorMsg := **if** error1 **then** errorMsg1
	else if error2 **then** errorMsg2
	else if error3 **then** errorMsg3
	else if error4 **then** errorMsg4
	else nostring;
	↑ error1: **boolean** ;
	↑ errorMsg1: **string** ;
	↓ myComponent: **ref** Component ;
	error1 := **not** classref.subclassOf(myComponent.classref);
	errorMsg1 :="Rule 1:A Decorator should be a subclass of Component";
	↑ error2: **boolean** ;
	↑ errorMsg2: **string** ;
	error2 := classref.a-Block.lookup(myDecComp.a-ID.val)
	=globals.a-SEMMissingDecl;
	errorMsg2 :=
	"Rule 2:A Decorator should have a DecoratedComponent variable";
	↑ error3: **boolean** ;
	↑ errorMsg3: **string** ;
	↓ myImplementations: **ref NodeBag**
	↓ myDecComp: **ref** *VarDecl*
	→ referenceval: **string** ;
	referenceval := myDecComp.a-ID.val;
	error3 :=
	foreach $M: *MethodDecl* **in** myImplementations **do**
	$R := (**init false**)
	$R **or** (
	foreach $S: MethodCallStmt **in** $M.a-Block.a-Stmts **do**
	$B := (**init true**)
	$B **and not** ((myDecComp.a-ID.val=(
	inspect $U1 := $S.a-Use
	when QualUse **do**
	inspect $U2 := $U1.a-Use
	when SimpleUse **do** $U2.a-ID.val
	otherwise nostring
	otherwise nostring)) **and**
	($M.a-ID.val=(
	inspect $U1 := $S.a-Use

Table B.3: Specification of rules for the **Decorator** design pattern, part 1

Nonterminals	**Attributes and Semantic Rules**
Decorator	**when** QualUse **do** **inspect** \$U2 := \$U1.a-UnQualUse **when** MethodUse **do** \$U2.a-ID.val **otherwise nostring** **otherwise nostring**))))); errorMsg3 := "Rule 5:All Implementations must have a delegating call to the corresponding Operation"; ↓ operations: **ref NodeBag**; ↑ error4: **boolean** ; ↑ errorMsg4: **string**; error4 := **not** operations.size(-)=myImplementations.size(-); errorMsg4 := "Rule 6: A Decorator must have an Implementation of all the Operations"
ConcDecorators	↑ error: **boolean** ↑ errorMsg: **string** errorMsg:= "Rule 4: A ConcreteDecorator must have an implementation of every operation" error := **foreach** \$C: *ClassDecl* **in** classes **do** \$R := (init false) \$R **or** (**foreach** \$M: *MethodDecl* **in** operations **do** \$S := (init false) **if** (findImplementation(\$M.a-ID.val , \$C)=\$M) **or** (findImplementation(\$M.a-ID.val , \$C)=**none**) **then** (\$S **or true**) **else** (\$S **or false**)); findImplementation: **func ref** *MethodDecl* (str: **string**,cls: **ref** *ClassDecl*):= **inspect** \$M := cls.a-Block.lookup(str) **when** *MethodDecl* **do** \$M **otherwise** **if** (cls.a-SuperOpt.superClass.a-Block.lookup(str) **in** *MethodDecl*) **then** cls.a-SuperOpt.superClass.a-Block.lookup(str) **else none** ;
DecoratedComponent	↑ error: **boolean** error:= varref=globals.a-SEMMissingDecl ↑ errorMsg: **string** errorMsg:= "Rule3: A DecoratedComponent variable must occur in a Decorator"

Table B.4: Specification of rules for the **Decorator** design pattern, part 2

Appendix C

Publications, teaching and presentations

Throughout my three years as a Ph.D. student, I have presented computer science in different ways, e.g. as publications, presentations, teaching. Below is a list of publications, teaching activities, presentations and other activities.

C.1 Publications

C.1.1 Conferences

"How to Preserve the Benefits of Design Patterns´´ Aino Cornils and Ellen Agerbo, Proc. of OOPSLA'98, Vancouver, Canada, October 1998 published by ACM

"Statically Checked Documentation with Design Patterns´´ Aino Cornils and Görel Hedin Proc. of TOOLS Europe 2000, Mont st. Michel, France, June 2000 published by IEEE.

C.1.2 Workshops

"Implementing GoF Design Patterns in BETA´´ Aino Cornils and Ellen Agerbo LSDF'97 i ECOOP'97, Jyväskylä, Finland, July 1997 published as Högskolan Karlskrona research report.

"Tool Support for Design Patterns based on Reference Attribute Grammars´´ Aino Cornils and Görel Hedin Proc. of WAGA'00, Ponte de Lima, Portugal, July 2000, published as INRIA research report.

C.1.3 Technical reports

"Mønstre - en indføring i analyse-, design- og arkitekturmønstre´´ Aino Cornils, Johnny Olsson and Ole Vedel Villumsen Technical report COT/4-07-v2.2, April 1999

C.2 Presentations

I have presented the four conference and workshop papers at the conferences. In addition to this I have presented the papers to other audiences, among these Lund Institute of Technology, University of Aarhus and various industry partners of Centre for Object technology.

I have given presentations on the conference on Java and object-orientation, JAOO, both in 1999 and 2000.

Both to industry partners within Centre for Object technology and other companys, such as DSB, and Unibank, I have given introductions to patterns and frameworks.

Other groups of people, as Datalogforeningen in Århus and Copenhagen, Java Erfaringsgruppen, The pattern group (PLOP) Denmark have received presentations and tutorials on patterns and framework.

C.3 Teaching

Spring of 1998:

Teaching assistant on dAds, a first-year course at the University of Aarhus, containing algorithm and complexity theory.

Fall of 1998:

Teaching assistant on dProg2, a second-year course at the University of Aarhus, containing theory of object-orientation, an object-oriented programming language.

Co-teacher on AOOP, Aspects of Object Oriented Programming, a postgraduate course at the University of Aarhus, containing aspects of object orientation ranging from the theory of type systems and philosophy behind object-orientation to the implementation of programming language constructs.

Spring of 1999:

Two courses on object orientation, patterns and frameworks to people from the industry.

Spring of 2000:

Teacher on DPF, Design Patterns and Frameworks, a postgraduate course at the University of Aarhus, containing theory and practise of design patterns, and an overview of analysis patterns, anti patterns, architectural patterns, organisational patterns, and frameworks.

Fall of 2000:

Co-teacher dPaSS (the course formerly known as dProg2).

Co-teacher AOOP.

Spring of 2001:

Co-teacher DPF.

C.4 Other activities

In 1998, I visited Ralph Johnson, one of the authors of the GoF book, at the departmant of computer science, University of Illinois.

In the fall of 1999, I visited LTH, Lund Institute of Technology. It was a five months stay with Görel Hedin as supervisor.

Reviewing papers for conferences as OOPSLA, ECOOP, CC and others.

The first two years of my Ph.D. I was involved in the promotion of natural sciences in general and computer science in particular to children in the age of 10-17.

Courses followed:

Aspects of Object Oriented programming (University of Aarhus)
Design Patterns and Frameworks (University of Aarhus)
Study group on Object oriented type systems (University of Aarhus)
Rastergraphical systems (University of Aarhus)
Compiler optimisation(University of Aarhus)
Kommunikationsteori (Lund Institute of Technology)

Bibliography

[ABW98] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison-Wesley Publishing Company, 1998.

[AC97a] Ellen Agerbo and Aino Cornils. Implementing gof design patterns in beta. In *LSDF'97 at ECOOP'97, Jyväskylä, Finland*, 1997.

[AC97b] Ellen Agerbo and Aino Cornils. Theory of language support for design patterns. Master's thesis, Department of Computer Science, Aarhus University, 1997.

[AC98] Ellen Agerbo and Aino Cornils. How to preserve the benefits of design patterns. In *Proceedings of OOPSLA'98*, pages 134–143, 1998.

[Ale64] Christopher Alexander. *Notes on the Synthesis of Form*. Harvard University Press, 1964.

[Ale75] Christopher Alexander. *The Oregon Experiment*. Oxford University Press, 1975.

[Ale77] Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[BC87] Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. In *OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming*, 1987.

[BC89] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In N. Meyrowitz, editor, *Proceedings of OOPSLA'89*, volume 24 of *Special Issue of SIGPLAN Notices*, pages 1–6, October 1989.

[BCC+96] Kent Beck, James O. Coplien, Ron Crocker, Lutz Dominick, Gerard Meszaros, Frances Paulisch, and John Vlissides. Industrial experiences with patterns. http://www.bell-labs.com/ cope/Patterns/ICSE96/icse.html, 1996.

[Bec97] Kent Beck. *Smalltalk Patterns: Best Practices.* Prentice Hall, 1997.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Publishing Company, 1999.

[BHN99] E. Bjarnason, G. Hedin, and K. Nilsson. Interactive language development for embedded systems. *Nordic Journal of Computing*, pages 36–55, 1999.

[BKPS92] F. Buschmann, K. Kiefer, F. Paulisch, and M. Stal. The meta-information-protocol, run-time type information for c++. In *Proceedings of the International Workshop on Reflection and Meta-Level Architecture'92*, pages 82–87, 1992.

[BMB+98] William J. Brown, Raphael C. Malveau, William H. Brown, Hays W., III McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, 1998.

[BMR+97] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley and Sons, 1997.

[Boe88] B. W. Boehm. A spiral model of software development and enhancemen. *IEEE Computer*, 1988.

[Bos97] Jan Bosch. Design patterns & frameworks: On the issue of language support. In *LSDF'97 at ECOOP'97, Jyväskylä, Finland*, 1997.

[Bro] Kyle Brown. Design reverse-engineering and automated design pattern detection in smalltalk. http://www2.ncsu.edu/eos/info/tasug/kbrown/thesis2.htm.

[BW95] Kyle Brown and Bruce Whitenack. Crossing chasms. http://members.aol.com/kgb1001001/Chasms.htm, 1995.

[CCD+98] Michael Christensen, Andy Crabtree, Christian Heide Damm, Klaus Marius Hansen, Ole Lehrmann Madsen, Pernille Marqvardsen, Preben Mogensen, Elmer Sandvad, Lennert Sloth, and Michael Thomsen. The m.a.d. experience: Multiperspective application development in evolutionary prototyping. In *Proceedings of ECOOP'98*, volume 1445 of *LNCS*, Bruxelles, Belgium, July 1998.

[CDH+99] Michael Christensen, Christian Heide Damm, Klaus Marius Hansen, Elmer Sandvad, and Michael Thomsen. Software architectural evolution in the dragon project. In *Workshop on Object-Oriented Architectural Evolution at ECOOP'99*, 1999.

[cet] Cetus links - object-orientation. http://www.cetus-links.org/.

[CH00a] Aino Cornils and Görel Hedin. Static-semantic checked documentation with design patterns. In *Proceedings of TOOLS Europe 2000*, 2000.

[CH00b] Aino Cornils and Görel Hedin. Tool support for design patterns using specification with reference attributed grammars. In *WAGA'00. Third Workshop on Attribute Grammars and their Applications*, 2000.

[cM98] Robert c. Martin. *Pattern Languages of Program Design 3*, chapter Acyclic Visitor, pages 93–105. Addison-Wesley, 1998.

[Coa92] Peter Coad. Object-oriented patterns. *CACM*, 1992.

[Cod] Codefarms. http://www.codefarms.com.

[Cop92] J.O. Coplien. *Advanced C++: Programming Styles and Idioms.* Addison-Wesley, Reading, MA, 1992.

[Cop95a] James O. Coplien. *A generative Development-Process Pattern Language*, chapter 13. Addison Wesley, 1995.

[Cop95b] J.O. Coplien. *Pattern Languages of Program Design.* Addison-Wesley Publishing Compagny, 1995.

[COV99] Aino Cornils, Johnny Olsson, and Ole Vedel Villumsen. Mønstre — en indføring i analyse-, design- og arkitekturmønstre. Technical Report COT/4 - 07, Centre for Object Technology, 1999.

[DN66] O.-J. Dahl and K. Nygaard. Simula: an ALGOL-based simulation language. *Communications of the ACM, CACM*, 9(9):671–678, 1966.

[Ede00] Amnon Eden. *Precise Specification of Design Patterns and Tool Support in Their Application.* PhD thesis, Department of Computer Science, Tel Aviv University, 2000.

[Ein97] Daniel Einarson. Using inner classes in design patterns, 1997.

[ErNM00] Tabita Enig, Henrik Kjær Nielsen, and Lars Mellergaard. Dpdoc - a user guidance and evaluation. Technical report, Institute of Computer Science, University of Aarhus, May 2000.

[FFVY96] F.J.Budinsky, M.A. Finnie, J.M. Vlissides, and P.S. Yu. Automated code generation from design patterns. *IBM Systems Journal*, 1996.

[FL00] Peter Forbrig and Ralf Lämmel. Programming Design Patterns. In *Proceedings TOOLS-USA 2000.* IEEE, 2000.

[FMvW97] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In *Proceedings of ECOOP'97*, 1997.

[Fow97] Martin Fowler. *Analysis Patterns – Reusable Object Models.* Addison-Wesley, 1997.

[Fow99] Martin Fowler. *Refactoring : Improving the Design of Existing Code.* Addison-Wesley, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of reusable Object-Oriented software.* Addison-Wesley, 1995.

[GL97] Joseph Gil and David H. Lorenz. Design patterns vs. language design. In *Workshop on Language Support for Design Patterns and Object-Oriented Frameworks (LSDF), ECOOP '97*, 1997.

[Gra98] Mark Grand. *Patterns in Java, Volume 1, A Catalog of Reusable Design Patterns Illustrated with UML.* John Wiley & Sons, 1998.

[Hed97] Görel Hedin. Language support for design patterns using attribute extension. In *Workshop on Language Support for Design Patterns and Object-Oriented Frameworks (LSDF), ECOOP '97*, 1997.

[Hed99] Görel Hedin. Reference attributed grammars. In *WAGA'99. Second Workshop on Attribute Grammars and their Applications. Amsterdam*, The Netherlands, March 1999.

[HM01] Görel Hedin and Eva Magnusson. Jastadd—a java-based system for implementing front ends. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

[HrLK00] Görel Hedin and Jørgen Lindskov Knudsen. On the role of language constructs for framework design. *ACM Computing Survey, Symposium on Object-Oriented Application Frameworks*, 32, March 2000.

[Hua98] Alfred Huang. *The Complete I Ching.* Inner Traditions, 1998.

[III98] Martin E. Nordberg III. *Pattern Languages of Program Design 3*, chapter Variations on the Visitor Pattern, pages 105–125. Addison-Wesley, 1998.

[Jav] Java. The javadoc tool homepage. http://java.sun.com/products/jdk/javadoc.

[KB96] Jung Kim and Kevin Benner. An experience using design patterns: Lessons learned and tool support. *Theory and Practice of Object Systems (TAPOS)*, 1996.

[KC] Wolfgang Keller and Jens Coldewey. Relational database access layers a pattern language. http://www.sdm.de/g/arcus/cookbook/relzs/index.htm.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of ECOOP'2001*. Springer, 2001.

[KMMPN83] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Abstraction mechanisms in the beta programming language. In *Conference Record of the POPL'83*, pages 285–298, 1983.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 1984.

[Kru00] Philippe Kruchten. *The Rational Unified Process, An Introduction*. Addison-Wesley Pub Co, 2000.

[Lar97] Craig Larman. *Applying UML and Patterns*. Prentice Hall, Inc, 1997.

[Lea99] Doug Lea. *Concurrent Programming in Java , Second Edition: Design Principles and Patterns*. Addison-Wesley Pub Co, 1999.

[MBMP93] O. L. Madsen and K. Nygaard B. Møller-Pedersen. *Object-Oriented Programming in the* BETA *Programming Language*. Addison-Wesley Publishing Company, 1993.

[Mei96] Marco Meijers. Tool support for object-oriented design patterns. Master's thesis, Department of Computer Science, Utrech University, 1996.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[MH00] E. Magnusson and G. Hedin. Program visualization using reference attributed grammars. In *NWPER'2000 (The Ninth Nordic Workshop on Programming and Software Development Environment Research)*, Lillehammer, Norway, May 2000.

[MlP92] O. L. Madsen and B. Møller Pedersen. Part-objects and their location. In *Proceeding of TOOLS '92*, pages 283–297, 1992.

[MM97] Thomas J. Mowbray and Raphael C. Malveau. *Corba Design Patterns*. John Wiley & Sons, 1997.

[MMMNS97] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Objekt orienteret analyse og design*. Marko, 1997.

[Mod] Modelmaker. http://www.modelmaker.demon.nl.

[NWB00] James Noble, Charles Weir, and Duane Bibb. *Small Memory Software: Patterns for Systems with Limited Memory*. Addison-Wesley Publishing Compagny, 2000.

[PH99] Patrik Persson and Görel Hedin. Interactive execution time predictions using reference attributed grammars. In *Second Workshop on Attribute Grammars and their Applications WAGA99*, 1999.

[Pre94] Wolfgang Pree. Meta patterns a means for capturing the essentials of reusable object-oriented design. In *Proceedings of ECOOP'94*, pages 150–162, 1994.

[PUP97] Lutx Prechelt, Barbara Unger, and Michael Philippsen. Documenting design patterns in code eases program maintenance. In *ICSE-97 workshop on Proces Modeling and Empirical Studies of Software Evolution*, pages 72–76, Boston, MA, May 1997.

[RBWPL91] J. Rumbaugh, M. Blaha, F. Eddy W. Premerlani, and W. Lorentzen. *Object-Oriented Modeling and design*. Prentice Hall, 1991.

[Ris97] Linda Rising. Customer interaction patterns. In *The 4th Pattern Languages of Programming Conference 1997*, 1997.

[Ris98] Linda Rising. *The Patterns Handbook : Techniques, Strategies, and Applications*. SIGS Books & Multimedia, 1998.

[Ris00] Linda Rising. *The Pattern Almanac 2000*. Addison-Wesley Publishing Compagny, 2000.

[rma00] Kurt Nørmark. Requirements for an elucidative programming environment, 2000.

[Sou95] Jiri Soukup. *Pattern Languages of Program Design*, chapter Implementing Patterns. Addison-Wesley Publishing Compagny, 1995.

[SSC96] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring compliance of a software system with its high level design models. In *Proceedings of the 18th International Conference on Software Engineering*, pages 387–397. IEEE Computer Society Press, 1996.

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley and Sons, 2000.

[TC98] Michiaki Tatsubori and Shigeru Chiba. Programming support of design patterns with compile-time reflectio. In *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, pages 56–60, Vancouver, Canada, Oct 1998.

[Tho97] Kresten Krab Thorup. Genericity in Java with virtual types. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 444–471. Springer-Verlag, New York, NY, 1997.

[uiu] The patterns homepage. http://hillside.net/patterns/patterns.html.

[Vil97] J. Viljamaa. Tools supporting the use of design patterns in frameworks. Technical Report C-1997-25, University of Helsinki, Department of Computer Science, 1997.

[Vil98] J. Viljamaa. Tools supporting the use of design patterns in frameworks. In *Proceedings of the ECOOP'98 Workshop on Object-Oriented Software Architecture (OOSA'98)*, 1998.