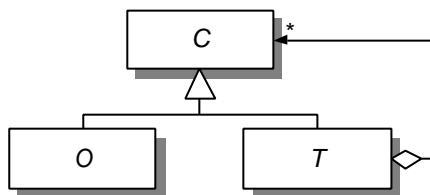


Perspectives on Patterns
COT/4-17-V1.0



Centre for Object Technology

*Centre for
Object Technology*

Revision history: June 1999 v1.0 Final version

Author(s): Aino Cornils,
 Department of Computer Science,
 University of Aarhus

Status: Public

Publication:

Summary:

This is the so-called “Progress Report” which is handed in as partial fulfilment of the requirements for the Ph.D. degree. This report serves two purposes: a description of the research I have been doing so far as part of my Ph.D. program, as well as a description of my plans for future research. I have worked with patterns from several perspectives, but always with the aim of exploiting their usefulness as much as possible. My work with patterns has followed three lines; preserving the benefits of design patterns, solving the trace problem by saving the core of patterns in a library, and introducing the concept of patterns to a Danish audience by giving the “use” perspective on patterns. Finally, my plans for future research are described.

© Copyright, Aino Cornils, 1999

The Centre for Object Technology (COT) is a three year project concerned with research, application and implementation of object technology in Danish companies. The project is financially supported by The Danish National Centre for IT-Research (CIT) and the Danish Ministry of Industry.

Participants are:
Maersk Line, Maersk Training Centre, Bang & Olufsen, WM-data, Rambøll, Danfoss, Systematic Software Engineering, Odense Steel Shipyard, A.P. Møller, University of Aarhus, Odense University, University of Copenhagen, Danish Technological Institute and Danish Maritime Institute

Perspectives on Patterns

“Progress Report” in partial fulfilment
of the requirements for the Ph.D. degree

Aino Cornils

Computer Science Department, University of Aarhus, Denmark.

e-mail: apaipi@daimi.au.dk

June 1999

Contents

1	Introduction	3
2	The History of Patterns	3
3	The Chronology of My Work	3
4	Preserving the Benefits of Design Patterns	4
4.1	An Analysis of Design Patterns	5
4.1.1	The Analysis	6
4.1.2	Applying the Analysis	9
4.1.3	Results	11
4.2	Related Work	12
5	Solving the Tracing Problem by Certain Language Features	13
5.1	Implementing the LDPs	14
5.1.1	Flyweight	14
5.2	Discussion	16
5.3	Related Work	17
6	Dissemination of Patterns	18
7	Patterns in the Development Process	19
7.1	Patterns and Object-Orientation	19
7.2	Future Work	21
8	Planned Research Activities	21
8.1	Implementing the LDPs	21
8.2	Patterns for Concurrency	23
9	Other Research-related activities	24

1 Introduction

This report serves two purposes: a description of the research I have been doing so far as part of my Ph.D. program, as well as a description of my plans for future research. I have worked with patterns from several perspectives, but always with the aim of exploiting their usefulness as much as possible. My work with patterns has followed three lines; preserving the benefits of design patterns as described in Section 4, solving the trace problem by saving the core of patterns in a library as described in Section 5 and introducing the concept of patterns to a Danish audience by giving the “use” perspective on patterns as described in Section 7. My plans for future research are described in Section 8.

2 The History of Patterns

Some twenty years ago, the architect Christopher Alexander published the book, “A Pattern Language” ([Alexander77]). In this book he presented 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*. 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 ranging from the smallest detail to the largest pattern. People should design for themselves their own houses, streets and communities, using his patterns. This because his observation was that “most of the wonderful places of the world were made by people, not architects”.

In 1987, Ward Cunningham and Kent Beck decided to use some of Alexander’s ideas to develop a small ‘pattern language’ for guiding novice Smalltalk programmers. They were working with Smalltalk and designing user interfaces, and wanted to help others benefit from their experience. It was their intention to make a pattern language which would cover the field of object-oriented programming, and they expected to cover it with 100-150 patterns. They wrote up the results and presented them at OOPSLA’87 in Orlando in the paper “Using Pattern Languages for Object-Oriented Programs”[Beck87]. Other computer scientists followed their trail and described their experience by patterns. Most famous of the collections of patterns is the ‘GoF-book’ [Gamma et al. 95]. The success of this book led to the birth of a seemingly never-ending stream of books of patterns, both with more design patterns and with new types of patterns, among these analysis patterns and software architecture patterns.

3 The Chronology of My Work

The motivation behind this Ph.D. project came from attending a course (at the Department of Computer Science at the University of Aarhus), called “Design Patterns and Frameworks”. It was held by Görel Hedin and besides introducing me to [Gamma et al. 95], it set me on a quest concerning the usefulness of patterns, as I will elaborate on later.

Görel Hedin encouraged my colleague, Ellen Agerbo, and me to write a paper on implementing the design patterns from 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 of [Gamma et al. 95]. The paper, ‘Implementing Design Patterns in BETA ’ [Agerbo97a], which contained this work, was accepted on

a workshop at the ECOOP'97 conference, and contained first attempts to change the religious¹ view 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." [Agerbo97b]. The thesis contains a thorough discussion of all the patterns based on their implementation in BETA and our work with a new idea; making a 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.

At this point in time I started my Ph.d. together with my colleague Ellen Agerbo. My Ph.D. was sponsored by Centre for Object Technology. Our thesis was accepted as a part of our Ph.D.

We wrote a paper "How to Preserve the Benefits of Design Patterns." [Agerbo98] which was a further development of material from our master's thesis. 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 popular trend, since the confusion with the large number of patterns was a common 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 others. Unfortunately Ellen Agerbo decided to quit her Ph.D. and I did not want to continue our common work on my own. Others will, though. At the conference there was an event called "A bowl of patterns", 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 myself.

Up until then I had been interested solely in design patterns. This changed when I wrote my introduction to patterns (see Section 6). Since the intention with this was to show how to use patterns in a development process, I extended my work field to analysis patterns and architectural patterns.

4 Preserving the Benefits of Design Patterns

This section outlines the conclusions Ellen Agerbo and I found when making our masters thesis and OOPSLA-paper. Together with Section 5 it represents the first year and a half of my Ph.D.

The rapid evolution of design patterns has hampered the benefits gained from using design patterns. The increase in the number of design patterns makes a common vocabulary unmanageable and the tracing problem obscures the documentation that should be enhanced by using design patterns. From our results in the thesis we discovered that the problems with design patterns solvable by our classification actually allowed us to regain the benefits of using patterns.

These benefits are the following:

1. Design patterns encapsulate experience.
2. Design patterns provide a common vocabulary for computer scientists across domain barriers.
3. Design patterns enhance the documentation of software designs.

¹Don't question the patterns in the "bible" [Gamma et al. 95].

We believed that the forming of design patterns should be restrictive, and suggested a way of evaluating existing design patterns which would lead to a decimation of the number of design patterns.

We proposed a set of guidelines to follow when evaluating a design pattern, and I will present the results of these guidelines applied to the design patterns of [Gamma et al. 95].

4.1 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 design 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 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 Page² 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 what 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 to document software systems. 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; will make it too laborious to find and use the encapsulated experience, and it 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 by in order to be accepted. The problem with this approach is that too much control in the innovative phase of discovering new design patterns will inevitably 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 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.

I will present the analysis in the form of a set of criteria, that we have used for an evaluation of the design patterns that are presented in [Gamma et al. 95]. Our analysis did not go so far as to identify the *true* design patterns and throw away the rest; instead it focused on assembling a core of *Fundamental*

²<http://hillside.net/patterns/patterns.html>

Design Patterns which should capture good object oriented design on a high enough level so that it can be used in various kinds of applications. The design patterns that were not judged to be fundamental were 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 [Gamma et al. 95], which means that the criteria are based on a rather narrow set of design patterns. If the analysis were 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 did believe, however, that the criteria could form a sound starting point in a much needed discussion on the quality of design patterns. And at the conference we were confirmed in our opinion.

In [Agerbo97b] we have shown that by using the guidelines of this analysis, we can remove half the design patterns from the core of Fundamental Design Patterns, so that out of the original 23 design patterns in [Gamma et al. 95] only 12 remain. I will give some examples of how the guidelines of the analysis are applied on a few of the design patterns. I will not be equally detailed in all three examples however, since it is the application of the first guideline that is the most interesting from a historical point of view. This exploration of BETA 's strong abstractions was what led us to the analysis of patterns in the first place. Examples of applications of the last two guidelines will only be described briefly. For the complete analysis I refer to [Agerbo97b].

4.1.1 The Analysis

I 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 was 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 [Gamma et al. 95] 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 [Gamma et al. 95, 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 [Gamma et al. 95] 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 [Gamma et al. 95] 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 communities 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 [Alpert98, p. 3] where the authors justify the need for gathering the design patterns from [Gamma et al. 95] 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.”

- The same design pattern can exist under different names in different programming languages. It will be hard to compare two design patterns coming from different groups of design patterns, since the backgrounds in the given programming languages will almost certainly have an impact on the presentation of the design pattern.
- If a programmer who has worked in one 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 [Coplien94], that contains a collection of C+ idioms.

If we 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 would 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 universally understandable for programmers regardless of their background. Since these design patterns can be categorised as mere variations or applications of a design pattern, we chose 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 Dependant Design Pattern. 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:

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 vary. 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 possess a common ground of knowledge, that will let them know the answers to certain problems without too much thought. In [Gamma et al. 95] 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.

4.1.2 Applying the Analysis

We applied the analysis on the design patterns in [Gamma et al. 95]. 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. I present an example of the application of each guideline on a Design Pattern. For the detailed analysis of all the design patterns I refer to [Agerbo97b].

Factory Method

The purpose of this Design Pattern is to create objects whose exact classes are unknown until runtime. This is done in [Gamma et al. 95] by instantiating the objects in virtual methods that can be bound at runtime as shown in Figure 1.

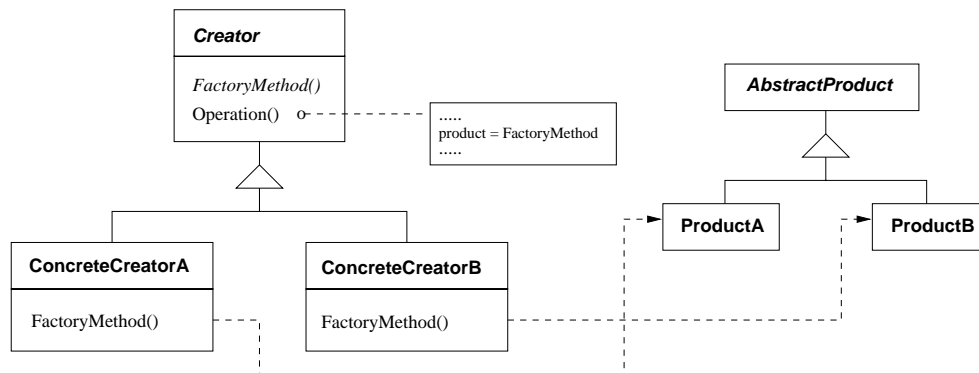


Figure 1: The **Factory Method** Design Pattern

In BETA the goal of this Design Pattern can be achieved quite differently. The concept of virtual classes is explained in depth in [Madsen89], is implemented in BETA ([Madsen93]) and has been proposed as an extension to JAVA ([Thorup97]). To show how the use of virtual classes will solve the problem behind **Factory Method**, we need an expansion of the OMT-based notation that has been used in [Gamma et al. 95]. We have chosen to use the notation in Figure 2 for a further binding of a virtual class. VP is in the class P declared to 'at least' have the type V, and this type is then extended in a subclass of P to have the type subV.

The similarity to the notation for inheritance is not coincidental. As with a specialisation P of a superclass SuperP, where it can be said that a P is 'at least' a SuperP, the further binding VP will 'at least' be the class VP that it extends.

Using this notation 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

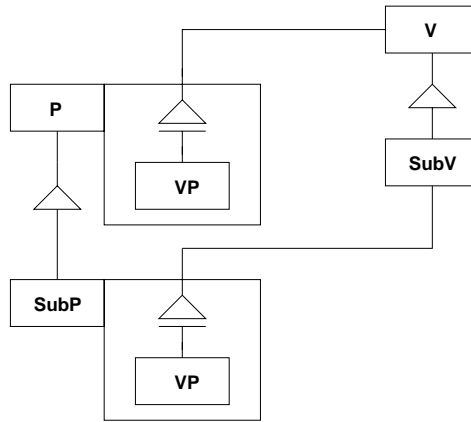


Figure 2: Further virtual bindings in subclasses

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.

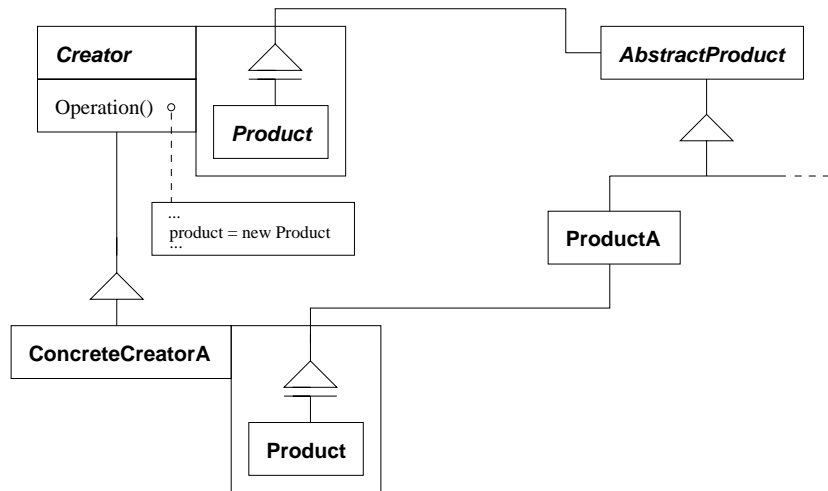


Figure 3: **Factory Method** modelled using virtual classes

An advantage in using virtual class patterns is that it is not necessary to rewrite a new **FactoryMethod** for each concrete product class. 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 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.

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 are 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.

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.

There *is* more information in an **Observer** than in a **Mediator** since the communication between the **Subject** and **Observers** is fixed, which 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.

Strategy

This **Strategy** Design Pattern defines a family of algorithms, encapsulates each one and makes them interchangeable. **Strategy** lets the algorithm vary independently from 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.

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.

4.1.3 Results

For each of the design patterns in [Gamma et al. 95], we have in [Agerbo97b] 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 4.

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

In conclusion, 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 4.

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.

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		3: Reuse of existing code.
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 4: Analysis of design patterns from [Gamma et al. 95]

4.2 Related Work

Since design patterns are a reasonably new concept, most of the efforts so far has 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” ([Gil97]) where Joseph Gil and David H. Lorenz have offered a taxonomy of the design patterns from [Gamma et al. 95] 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 [Agerbo97b]. Their resulting taxonomy is difficult to compare to ours directly, since they allow the same design pattern to appear in several categories, and their reasoning is 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; it will be especially hard to agree on what design patterns are formalisations over inherent object oriented ways of thinking — [Gil97] claims that three of the design patterns from [Gamma et al. 95] fall into this category, none of which we have categorised in the same way. However the fact that two set of Guidelines with some common classifications have evolved independently indicates that they can be used as valid starting points for a dialogue on the quality of design patterns.

5 Solving the Tracing Problem by Certain Language Features

This section covers the rest of the results from our thesis. It began as an attempt to capture the core of each design pattern, using BETA 's special language constructs to save as much as possible. Since it became evident that it was worthwhile, that there was a core to save, we tried to find that part of each of the design patterns from GoF that could be saved in a library. And were amazed of how useful it could be.

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 is known as the *tracing problem*.

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 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 [Gamma et al. 95], 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 rigidity on any application in which the LDP might be applied. The design pattern may be *fixed*, in the sense that it may not be possible to adapt it in ways other than those 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. 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. We will describe how the use of composition can simulate multiple inheritance, theoretically. In our example, though, we will use inheritance for the sake of simplicity.

5.1 Implementing the LDPs

In [Agerbo97b] we have discussed how and to what extent the Fundamental Design Patterns could be placed in a library of design patterns. I will 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 LDPs in [Agerbo97b] we assumed that such a library would be used in a language with multiple inheritance or the possibility to simulate multiple inheritance by composition, because the use of LDPs will mean that the classes in the application inherit from the classes in the LDP.

The following example is based on the descriptions of the design patterns found in [Gamma et al. 95], and require the book at hand for full understanding.

5.1.1 Flyweight

The application-dependent issues to consider when making an LDP 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 5.

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`

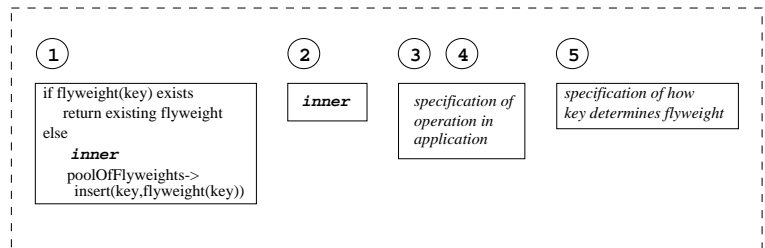
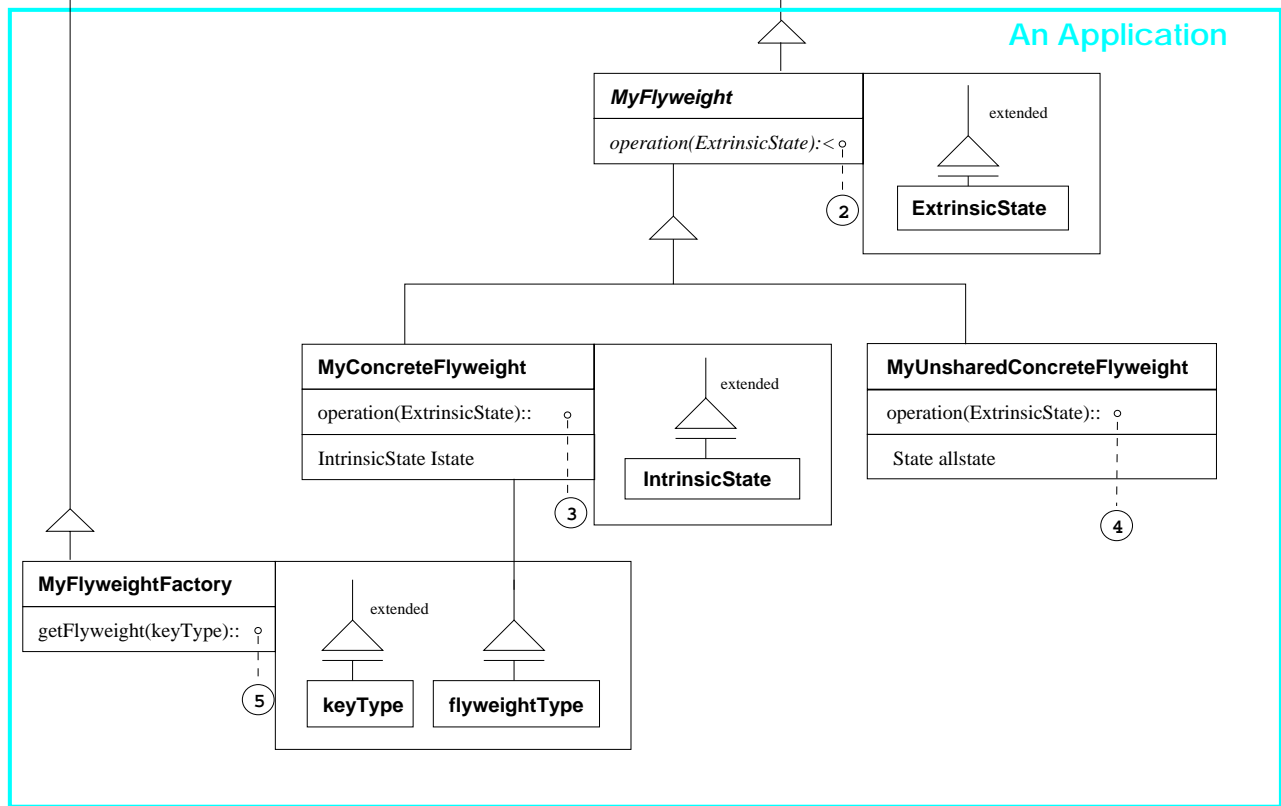
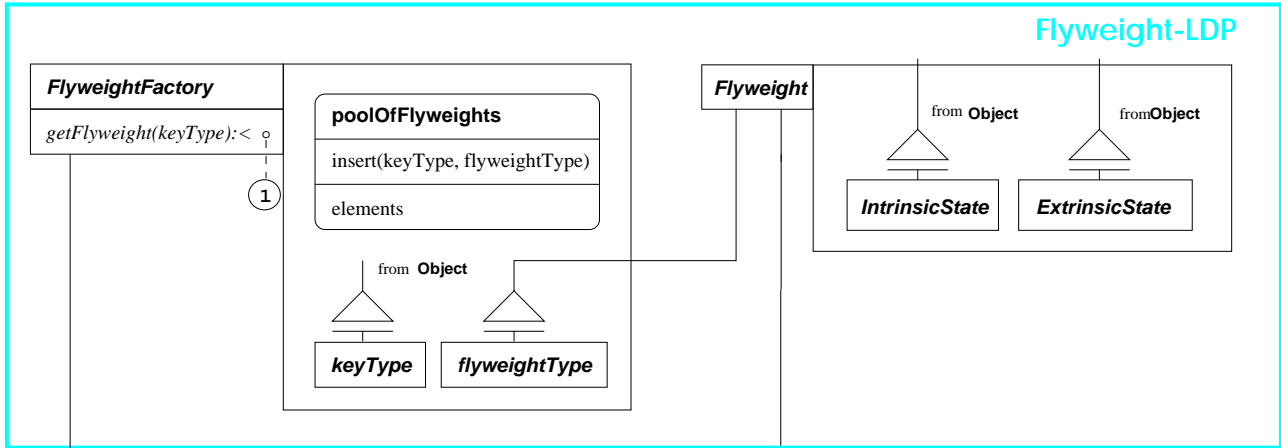


Figure 5: Flyweight-LDP

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 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 met whenever this LDP is applied in an application.

5.2 Discussion

Part of the goal of our thesis was to investigate to what extent it would be possible to make useful LDPs out of the Fundamental Design Patterns.

By a useful LDP we mean an LDP which captures the intent of the Design Pattern by describing the class hierarchy and the collaboration between the classes.

For some of the design patterns it was not possible to make a useful LDP, because the amount of information saved in an LDP would be too small. These design patterns could still be included in a library of design patterns, because the use of them would annotate where the design patterns were used in the application. This is a very important contribution to the documentation of software systems.

The design patterns, for which we find it possible to make a useful LDP in BETA, are marked with a \checkmark in the table in Figure 6.

A number of the existing or proposed language features in BETA prove especially useful in connection to the LDPs, by supporting genericity and reuse of models.

One of the most beneficial language features are the proposed *virtual lists* [Kristensen83] used in connection with *dynamic lists*. These features used together support LDPs for: **Builder**, **Bridge**, **Composite** and **Chain of Responsibility**, and eases the implementation of LDPs for **Flyweight** and **Iterator**. The mechanism of *virtual classes* is also a very useful feature in reuse. The LDPs by this mechanism are: **Bridge** and **Flyweight**³. The proposed *renaming scheme* [Madsen92] supports LDPs for: **Bridge** and **Composite**. This feature could in addition solve one of the general problems with LDPs; that the names in an LDP should be adapted to the application in which it is applied.

Another general problem could be solved by using composition as an alternative to multiple inheritance. If an application uses a library of design patterns by inheriting from the LDPs, a language without multiple inheritance will run headfirst into a wall, when classes play roles from more than one design pattern. Using composition instead of inheritance in languages without multiple inheritance solves this problem.

³and the LDDPs **Command** and **Observer**

Design Pattern	LDP possibility
Abstract Factory	
Builder	✓
Bridge	
(run-time)	✓
Composite	✓
Decorator	✓
Flyweight	✓
Proxy	
Chain of Responsibility	✓
Iterator	✓
Mediator	
Memento	
State	

Figure 6: Taxonomy of the LDPs

Since the LDPs are reused by including roles in the classes of the application by nesting an instance of a locally defined anonymous subclass of the wanted LDP-class, 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; “design patterns enhance the documentation of software designs”.

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 [Gamma et al. 95], as mentioned in [Agerbo97b] formulate good design- or implementation-ideas, the language features that support them must be considered to be flexible and useful in relation to reuse of design.

From this it is clear that BETA is a very good choice of programming language, when one wants to implement and reuse LDPs, mainly because of its *virtual classes*, *generalised block structure* and *dynamic lists*. It would be even better if *virtual lists* and a *renaming scheme* was incorporated.

5.3 Related Work

The tracing problem has become a generally recognised problem within the field of design patterns. Görel Hedin has in [Hedin97] proposed 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 annotate 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. The largest difference between this approach and ours, is that Hedin’s solution can work as a debugger for design patterns where our solution can not guarantee that the design patterns are applied correctly. Instead it will partly reduce

the implementation overhead. I intend to test our ideas together by implementing our LDPs in Görel Hedins tool, APPLAB, as part of my studies. This plan is explained in detail in Section 8.

Jiri Soukup has also tried to solve the tracing problem. In his article “Implementing Patterns” ([Soukup95]) 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 are only pointers and other data required for the design pattern. The problem of this solution is that all the structure of the design pattern is lost, since everything is now contained as methods in the *pattern class*.

6 Dissemination of Patterns

As mentioned earlier, I believe that patterns are useful and that research in this area should produce results that eases and thereby encourages the use of patterns. This belief also triggered my latest work, a Danish introduction to the use of patterns [Cornils99].

Working with people from the industry through COT⁴, I got to see to what extent patterns were used in Denmark. Considering the good experiences a number of developers have had with using patterns [Coplien1996] it seemed logical that developers in Denmark could benefit from using the patterns too. Unfortunately, as Ellen and I had predicted in our paper [Agerbo98], they were confused by the large number of patterns and often discarded them before trying to use them. Our work on a reduction of the number of patterns could make things better, but such a classification was not likely to take place overnight. Right now they needed motivation for using patterns and some guidelines for which patterns to use and how to use them. Triggered by this, I, in association with Johnny Olsson⁵ and Lisbeth Bergholt⁶, wrote an introduction to patterns to the Danish audience.

In the report we focused on the use of patterns by describing what the concept covers and giving an explanation on how and when patterns can be used in a development process. Our audience was software developers who know something about object-oriented software development. As discussed in Section 7.1 it is important to know the basics about object-orientation in order to understand the descriptions of the patterns. As reference material we used only three books, the reason being that it is best to start by using only a limited number of patterns. One can always extend the number, but it is hard to keep from being confused when presented to a large number of patterns.

We chose the books

”Analysis Patterns” by Martin Fowler [Fowler97], ”Design Patterns – Elements of reusable Object-Oriented Software” by Gamma, Helm, Johnson and Vlissides, (Gang of Four) [Gamma et al. 95] and ”Pattern-oriented Software Architecture” by Buschmann et al. [Buschmann97], because they together cover both analysis and design of a development process and because they are popular and have been used by many people over the last years. The fact that they are popular gives them the important status of a common vocabulary and gives a rough indication of usefulness.

⁴Centre for Object Technology

⁵WM-data, Denmark

⁶Teknologisk Institut, Denmark

7 Patterns in the Development Process

The various part activities involved and where the different types of patterns can be used in a model of the system development process is shown in Figure 7. The process is partitioned in part activities, each with a certain problem and solution domain. The partitioning in 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 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 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 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 single 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 three types of 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 7. 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 a change in choice of architecture pattern after implementation is more painful than a change in choice of design pattern, due to the larger size of the elements that comprise the architectural patterns.

7.1 Patterns and Object-Orientation

In my introduction to patterns I defined my target group to be software developers who know something about object-oriented software development. This could lead to the misinterpretation that I do not believe that patterns are useful in other language paradigms than the object-oriented. But this is not the case.

On the face of it, the solutions we find in the patterns look object-oriented. They are identified in an object-oriented context and are therefore described by objects and relations.

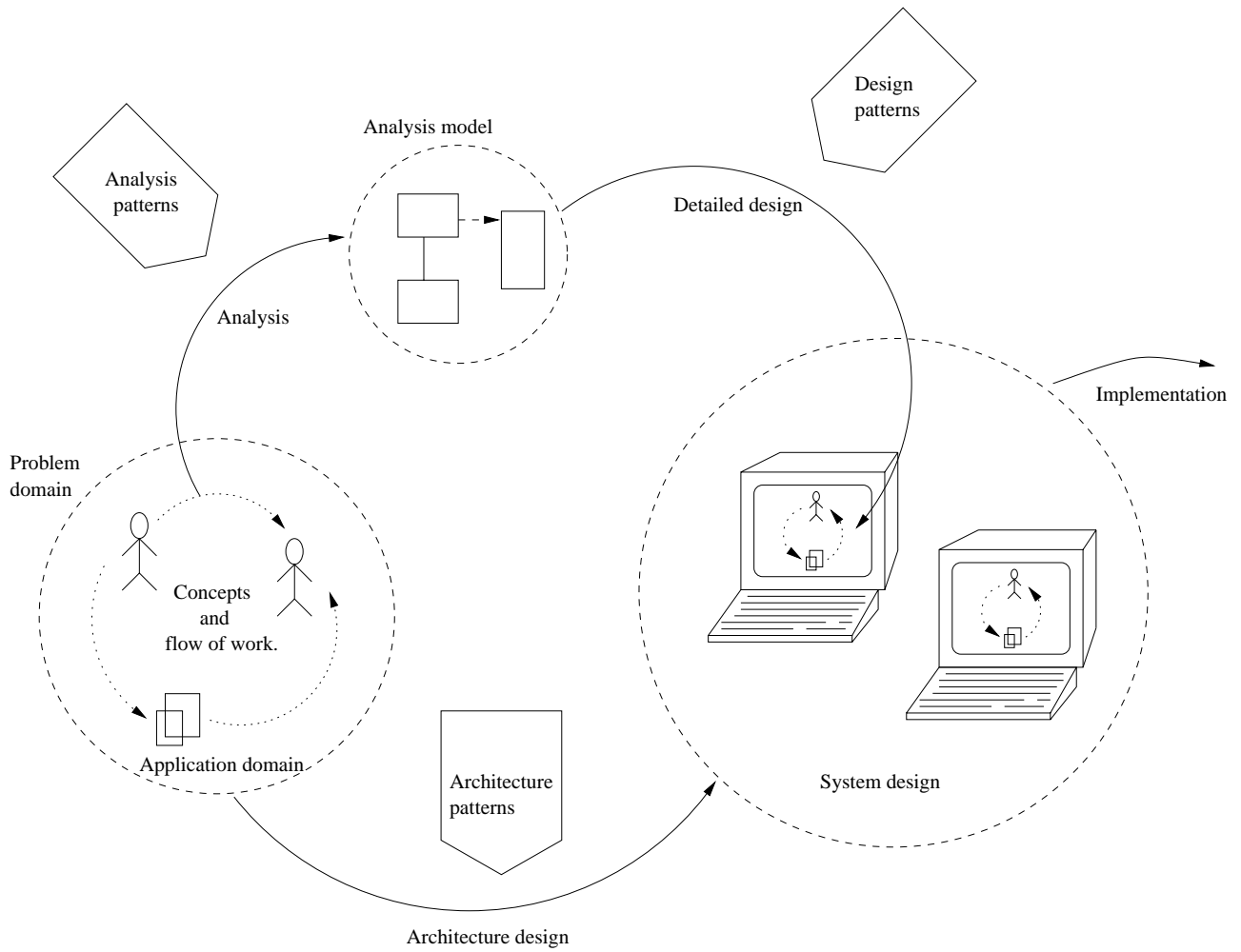


Figure 7: The focus of the different patterns in the development process.

In the following discussion we must distinguish between analysis patterns and architectural patterns on the one hand and design patterns on the other.

Analysis patterns and architectural patterns use object-orientation on a very abstract level. It is thus not necessary to use object-oriented languages to implement the design described by them.

It is a different situation for design patterns that work on an object/class level which means they rely more on the possibility of using objects. It is tempting to conclude that implementation of design patterns is limited to languages with inheritance and polymorphism. These abstractions, however, can be simulated by other language constructs, such as function pointers.

Additionally the ideas found in design pattern solutions are useful to all language paradigms. In the worst case the design patterns will lose a part of their elegance, since the object-oriented abstractions are simulated. Therefore I believe that patterns are useful for all language paradigms.

7.2 Future Work

An extension of the introduction to patterns is planned to take place this summer. It is our intension to extend it with patterns for access to relational databases based on [Brown95] and [Keller96]. Most systems have some kind of database which the application needs to access. This work plus the actual modelling of the database takes place in all parts of the development process and therefore I believe that the use of these patterns is orthogonal to the use of the three types of patterns we have described already and yet relevant for the introduction.

8 Planned Research Activities

I plan to finish my Ph.D. early 2001, a year and a half from now. Until then I will pursue two lines of research.:

The first of them is realising the LDPs, described in the masters thesis, in Görel Hedins APPLAB as described in Section 8.1. I will do this while visiting Görel Hedin at the University of Lund this autumn and winter. When I return to Aarhus in the spring, I will test the work I did in Sweden either in cooperation with people from COT, or with colleagues from the University of Lund.

Additionally I plan to identify patterns for concurrency and extend the number of implemented LDP's with these as described in Section 8.2.

8.1 Implementing the LDPs

In some ways we want to be able to treat patterns like language constructs. Both in the sense that we want to be warned when we misuse them and that we want it to be clear from the code, where they are used. But we don't want to actually make all the patterns new language constructs. According to our discussion in [Agerbo97b], it can be worthwhile to make new language constructs out of some patterns, but certainly not all of them. Additionally we expect patterns to evolve, both globally because there will be more of them but also internally, when we discover that some patterns are alike or applications of one another.

As Görel Hedin states it in her paper "Language support for Design Patterns using Attribute Extension" [Hedin97] : "We can think of a pattern application as a kind of language construct which identifies the objects that play the particular roles in the pattern, and which specifies some rules that these objects must follow."

Görel Hedin suggests implementing the core of the patterns in a tool using attribute extension thereby enforcing the rules of the patterns in applications.

Attribute extension can support the specification and application of design patterns by extending the static-semantics of the language, by allowing enforcement of rules and keeping the syntax of the base language.

It is demonstrated in [Hedin97] how this could be done knowing the roles and the rules of the patterns. The roles can be of two types; defining roles and derived roles. The only classes, methods and variables that have to be marked in the code are the elements playing the defining roles. The other elements playing roles of the pattern can be derived from these roles, hence the name "derived roles". An example could be that all subclasses of a class playing a certain defining role, will play a certain derived role in the pattern. Of course this could sometimes mean, that it is impossible to make subclasses of that class without having them playing a role in the pattern. There are also two kinds of rules; role rules and collaboration rules. The role rules are the rules that must be followed when annotating elements with defining, while collaboration rules are rules describing how the elements playing the roles should collaborate, e.g. which methods they should implement. Naturally the role rules are checked when the pattern is applied in the code, while the collaboration rules must be enforced every time something reuses the code that implemented the pattern roles.

This is particularly interesting when using frameworks. If a framework is built with pattern applications at the points for extension, the *hot spots*, the programmer will be reminded how the framework should be used, when he uses it in an erroneous way. This would solve one of the major problems with frameworks; the trouble with using them in the intended way.

In the masters thesis we proposed to make a library of design patterns. As stated in the OOPSLA-paper, such a library would solve the tracing problem and reduce the implementation overhead. We identified the core of the pattern to be able to implement it in a library without violating the solution found in the pattern. This core, the stable part of the pattern, together with our application comments describes the roles and most of the rules of the pattern. I assume that the rest of the rules can be identified with the help of one of the various attempts to formalise patterns, e.g. [Eden96]. Since this knowledge of the roles and rules is what is needed for enforcing the rules of the patterns in applications with attribute extension, we can connect our work with that of Görel Hedin for mutual gain.

APPLAB is an interactive language development tool, which is implemented with reference attribute grammars. Reference attribute grammars are extensions of attribute grammars, similar to attribute extension. Reference attribute grammars can handle advanced name and type analysis problems, by allowing attributes to be references denoting nodes arbitrarily far away in the syntax tree. 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. This makes APPLAB a highly interactive and flexible environment for language design. Viewing design patterns as language constructs it would be beneficial to use APPLAB to discover how to use reference attribute grammars to enforce the pattern rules on applications in the best way possible.

Somewhat like a compiler would, if the programmer tried to misuse a language construct. APPLAB will warn programmers if they try to do something that would break the rules. It is the claim, that this would help developing systems by giving the programmers possibility to apply patterns in an easy and safe way. This way we can get the benefits of the patterns acting like language constructs without polluting the programming language with a lot of patterns, that might change over time.

I believe that this project can enhance the use of patterns, since it will be easier (and more fun) to use them now there is a tool to guide the programmers. Enhancing the use of patterns, will strengthen the common vocabulary, thereby making it easier for programmers to communicate object-oriented design across language barriers. System development in general and framework application in particular will clearly benefit from this.

8.2 Patterns for Concurrency

Designing systems with parallel processes can be very difficult and it is important, almost essential that one has experience with it. There are many things to consider before choosing the path to follow, and many things to remember even after the right path is chosen. This is why I believe patterns could be helpful for designing concurrent systems.

First of all, such a system of patterns would help novice programmers do parallel programming. Additionally, since patterns for concurrency are roughly on the same level of abstraction as the design patterns, it would be possible to extend the collection of patterns in APPLAB with the patterns for concurrency. This would result in the same benefits for patterns for concurrency as I described for design patterns in Section 8.1.

An example of a pattern for concurrency could be the design of a monitor. One of the rules would be that every entry point to a monitor should be protected. In this way it would be possible to get a warning when the programmer tried to create an unprotected access to a variable in the monitor. It would then be up to the programmer to decide whether it should be corrected or whether it was intentional.

This possibility to decide whether to follow a rule or not is not part of the thought behind implementing the design patterns in APPLAB. The idea behind using APPLAB with the design patterns was that it would be impossible to misuse the patterns, by making it impossible to break the rules. In the patterns for concurrency, however, we are aware that there exists rules that not necessarily should be followed. These rules should not be restrictions for the programmer, just reminders, that forces him to make a conscious decision. It is worth a discussion whether this possibility should also be present for the design patterns. It will not make it easier for programmers to use the design patterns, but on the other hand it will avoid getting too much rigidity in the pattern applications.

If the patterns for concurrency were implemented in APPLAB we could avoid bringing politics into our programming languages, by saying that it can be decided in each case, whether it is necessary to restrict the access. As an example Per Brinch Hansen says in his article “Java’s Insecure Parallelism” [Hansen99] :

”For a parallel programming language the most important security measure is to check that processes access disjoint sets of variables only and do not interfere with each other in time-dependent ways.”

He believes very strongly that this is the only way and everybody doing it in another way are doing it wrong. He feels that they are totally ignoring his theories, which were proved valuable more than twenty years ago. He notes that in Java parallel threads can access shared variables, either directly or indirectly, without any synchronisation. He concludes from this that one can write a Java class that uses both private and public variables accessed by both synchronised and unsynchronised methods. This way, a programmer who wants to implement a monitor with restricted access to the variables would have to remember to declare all the variables private and make the methods, which access them, synchronised. His worry is that programmers will tend to forget this from time to time and he cannot see how a compiler could detect such errors. Based on this he claims that it is insecure to do parallel programming in Java.

Using APPLAB with patterns for concurrency, we can extend the effect we normally get from the compiler without influencing the language, and solve exactly that problem.

9 Other Research-related activities

I have participated in three different conferences; ECOOP'97, where Ellen Agerbo and I presented the paper "Implementing Design Patterns in BETA" [Agerbo97a] at "Workshop on Language Support for Design Patterns and Object-Oriented Frameworks". EuroPLoP'98 where I was able to discuss my ideas on patterns with a lot of people with very strong opinions from the pattern community. OOPSLA'98, where I presented the paper "How to Preserve the Benefits of Design Patterns" [Agerbo98]. On our way to this last conference, Ellen and I visited one of the authors behind [Gamma et al. 95], Ralph Johnson at the University of Illinois. We spent four days with him and his "patterns group". This gave us an opportunity, through discussions with our American colleagues and observations of their meetings, to learn some of the differences between the Danish and the American person hierarchy. Ralph Johnson was surprisingly⁷ tolerant with our critique of the patterns and agreed with us, that the patterns in [Gamma et al. 95] are not eternal truths.

As part of my Ph.D. studies I have taught computer science to students at the University. I have taught algorithmics and complexity theory to first year students, introductory object-oriented programming to second year students and advanced object technology to second part students. Additionally I am a member of the image committee at the Institute of computer science, trying to better our image towards potential students in order to recruit more students. When I return to Aarhus after my stay in Sweden in the spring I plan to teach the second-part course "Design Patterns and Frameworks".

I have given my presentation of the OOPSLA paper again at two second part courses at our university, at the university of Lund and to a meeting in the OMT/UML user group at the Institute of Technology, Aarhus, Denmark.

In the beginning of my work with COT, I introduced the concept of patterns and some of the latest research on them to a COT group. I have worked together with people from WM-data to introduce them to the use of patterns, giving three presentations at the company. The first was held for the entire company and presented the introduction to patterns [Cornils99], the second and third concerned patterns for user interface design and patterns for access to relational databases respectively, and were held for small groups.

⁷Most surprising for his students

References

- [Agerbo97a] Ellen Agerbo and Aino Cornils (1997): *Implementing Design Patterns in BETA* Workshop on Language Support for Design Patterns and Object-Oriented Frameworks (LSDF), ECOOP '97.
- [Agerbo97b] Ellen Agerbo and Aino Cornils (1997): *Theory of Language Support for Design Patterns*. Department of Computer Science, Aarhus University.
- [Agerbo98] Ellen Agerbo and Aino Cornils (1998): *How to Preserve the Benefits of Design Patterns* OOPSLA'98.
- [Alexander77] Christopher Alexander et al.(1977): *A Pattern Language*. Oxford University Press.
- [Alpert98] Sherman R. Alpert, Kyle Brown and Bobby Woolf (1998): *The Design Patterns Smalltalk Companion*. Addison-Wesley Publishing Company.
- [Beck87] Kent Beck and Ward Cunningham (1987): *Using Pattern Languages for Object-Oriented Programs*. Workshop on the Specification and Design for Object-Oriented Programming. OOPSLA'87.
- [Bosch97] Jan Bosch (1997): *Design Patterns & Frameworks: On the Issue of Language Support*. Workshop on Language Support for Design Patterns and Object-Oriented Frameworks (LSDF), ECOOP '97.
- [Brown95] Kyle Brown and Bruce Whitenack (1995): *Crossing Chasms*. <http://members.aol.com/kgb1001001/Chasms.htm>
- [Buschmann97] Frank Buschmann et al. (1997): *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons.
- [Coplien94] J.O. Coplien (1994): *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, Reading, MA.
- [Coplien1996] James O. Coplien et al.(1996): *Industrial Experiences With Patterns* <http://www.bell-labs.com/cope/Patterns/ICSE96/icse.html>
- [Cornils99] Aino Cornils, Johnny Olsson and Lisbeth Bergholt (1999): *Mønstre — en indføring i analyse-, design- og arkitekturmønstre*. COT/4 - 07.
- [Eden96] Amnon H. Eden, Joseph (Yossi) Gil, Amiram Yehudai (1996): *A Formal Language for Design Patterns*. The 3rd Annual Conference on the Pattern Languages of Programs. (Washington University technical report WUCS-97-07)
- [Fowler97] Martin Fowler (1997): *Analysis Patterns — Reusable object models*. Addison-Wesley Publishing Company.
- [Gamma et al. 95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995): *Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company.
- [Gil97] Joseph Gil and David H. Lorenz (1997): *Design Patterns vs. Language Design*. Workshop on Language Support for Design Patterns and Object-Oriented Frameworks (LSDF), ECOOP '97.

- [Hansen99] Per Brinch Hansen (1999): *Java's Insecure Parallelism*. ACM SIGPLAN Notices No. 4 Vol 34, april 1999.
- [Hedin97] Görel Hedin (1997): *Language Support for Design Patterns usign Attribute Extension*. Workshop on Language Support for Design Patterns and Object-Oriented Frameworks (LSDF), ECOOP '97.[14]
- [Keller96] Wolfgang Keller and Jens Coldewey (1996): *Relational Database Access Layers A Pattern Language* <http://www.sdm.de/g/arcus/cookbook/relzs/index.htm>
- [Kristensen83] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard (1983): *Abstraction mechanisms in the BETA programming language*. Conference Record of the POPL '83 pp. 285-298. ACM
- [Madsen89] O. L. Madsen, B. Møller-Pedersen (1989): *Virtual classes: A powerful mechanism in object-oriented programming*. Proceeding of OOPSLA '89.
- [Madsen92] O. L. Madsen, B. Møller-Pedersen (1992): *Part-objects and their location*. Proceeding of TOOLS '92 pp. 283-297.
- [Madsen93] O. L. Madsen, B. Møller-Pedersen, K. Nygaard (1993): *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley Publishing Company.
- [Mat97] L. Mathiassen, A. Munk-Madsen, P.A. Nielsen og J. Stage (1997): *Objektorienteret Analyse og Design*. Forlaget Marko, Aalborg, 1997.h
- [Soukup95] Jiri Soukup (1995): *Implementing Patterns*. Pattern Languages of Program Design. Eds. Coplien and Schmidt. Addison-Wesley 1995.
- [Thorup97] K. K. Thorup (1997): *Genericity in JAVA with Virtual Types*. Proceedings of ECOOP '97 pp. 444-469. Springer-Verlag.