How to preserve the benefits of Design Patterns

Ellen Agerbo Aino Cornils
Computer Science Department, University of Aarhus, Denmark.
e-mail: {agerbo | cornils}@daimi.aau.dk

DRAFT: This paper has been accepted at OOPSLA '98.
Please do not cite this version.

Abstract

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. We present an analysis of Design Patterns that will strongly reduce the number of Fundamental Design Patterns and show how strong language abstractions can solve the tracing problem and thereby enhance the documentation.

1 Introduction

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:

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

The objective of this paper is to promote the point of view that the formation of Design Patterns should be restrictive, and to suggest ways to evaluate existing Design Patterns, leading to a reduction of the number of Design Patterns.

In this thesis 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 [Gamma et al. 95].

For the Design Patterns that are considered genuine Design Patterns after this evaluation, we have investigated how they could be defined as as "Library Design Patterns" in a class library and reused by use of inheritance or delegation — any such Design Pattern will in this paper 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 [Bosch97]. Another advantage is that by using a LDP it will be possible to *trace* that the Design Pattern is used in an application, which consequently will promote the documentation benefit.

2 An Analysis of Design Patterns

The generally accepted 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, such that designers of software applications can benefit from work previously done within 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 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 exists numerous patterns, and that the amount is continuously increased by PLoP conferences and discussion groups.

The obvious consequence is that the number of Design Patterns will grow to a level, where it becomes impossible to maintain an impression of which 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. This 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 to be 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

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

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 paper 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 [Gamma et al. 95]. Our analysis does not go so far as to identify the *true* Design Patterns and to throw away the rest; instead, it focuses on assembling a core of *Fundamental* Design Patterns (FDPs) which should capture good Object Oriented design on a sufficiently high level so that it 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 [Gamma et al. 95], 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 in a much needed discussion on the quality of the Design Patterns.

In [Agerbo97] we have shown that by using the guidelines of this analysis, we have evaluated the Design Patterns so that out of the original 23 Design Patterns in [Gamma et al. 95] only 12 Design Patterns qualify as Fundamental Design Patterns. We give some examples of how the guidelines of the analysis are applied on a few of the Design Patterns — for the complete analysis we refer to [Agerbo97].

2.1 The Analysis

We present an analysis whose purpose it is to define 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 as an FDP. It will be possible to make a stricter analysis by adding further guidelines without changing the original guidelines.

2.1.1 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 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 [Alpert et al. 98, 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 given programming languages will almost certainly have an impact on the presentation of the Design Pattern.

- If a programmer who has accustomed to work 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 [Coplien94], that contains a collection of C++ idioms.

If we on the other hand 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.

2.1.2 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, such 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*. For each of these families, the original Design Pattern (which can be either a Fundamental or a Language Dependent Design Pattern) will figure as the head of the family. 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.

2.1.3 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 [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.

2.1.4 Design Patterns reconsidered

The product of an analysis will be the Design Patterns divided into three categories: The Fundamental Design Patterns, which are language-independent original ideas; the Language Dependant Design Patterns, which are covered by a language construct and the Related Design Patterns, which are applications or variations of a Fundamental or Language Dependant Design Pattern. For all three categories of the Design Patterns the actual implementations will vary from one language to another, and it could therefore be useful to collect implementation hints in language specific catalogues (as it is done in eg. [Alpert et al. 98]). But it is important to keep the design ideas as far from actual languages as possible, such that all designers can gain from them regardless of background..

2.2 Applying the analysis

We have 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. For the obvious reasons of space, we will not present the evaluations of all 23 Design Patterns in this paper, but instead we present an example of the application of each guideline on a Design Pattern. For the detailed analysis of all the Design Patterns we refer to [Agerbo97].

2.2.1 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.

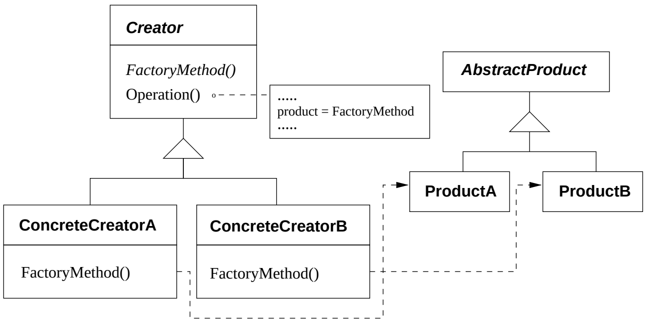


Figure 1: The **Factory Method** Design Pattern

In a language with *virtual classes* 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 ([BETA93]) 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.

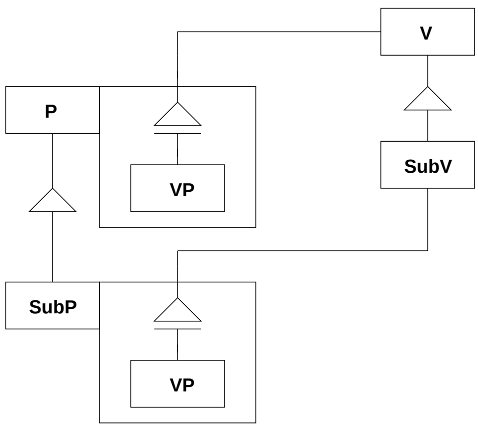


Figure 2: Further virtual bindings in subclasses

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 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 productclass virtual. This makes it possible to bind the class to be instantiated at runtime, instead of binding the *creator*-method at runtime.

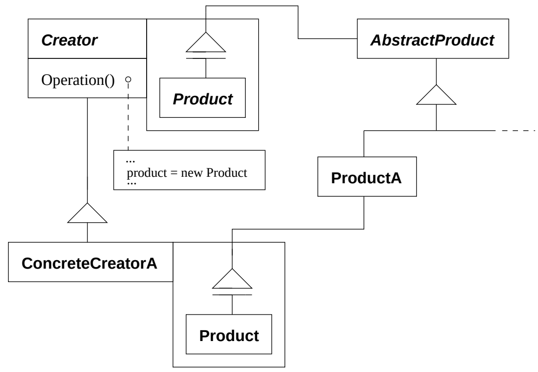


Figure 3: **Factory Method** modelled using virtual classes

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.

2.2.2 Observer

The motivation behind this Design Pattern is to define a one-to-many dependency between objects such that when one object changes state, all its dependents are notified and updated automatically. An object (a Subject) can have many representations (Observers) and when one of these representations are changed by the user, the object 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 to 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. The structure is shown in Figure 5.

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

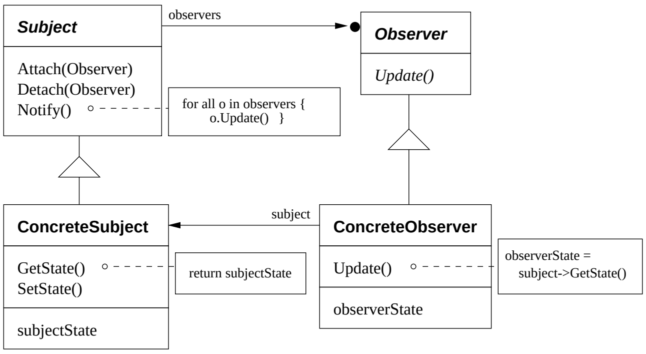


Figure 4: The **Observer** Design Pattern

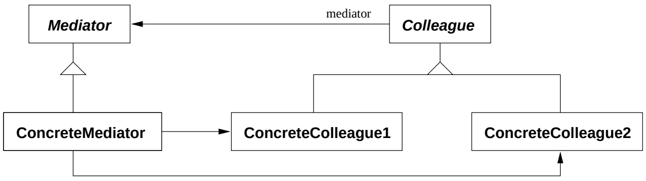


Figure 5: The **Mediator** Design Pattern

of the ConcreteMediator and the ConcreteObservers play the role of the ConcreteColleagues. Thus the ConcreteSubject will be the mediator between 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.

2.2.3 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. The structure of the **Strategy** Design Pattern is shown in figure 6.

When comparing the applicability of the **Strategy** Design Pattern with the intent of the **State** Design Pattern in [Gamma et al. 95, pp. 305], it will appear as if **State** solves the same problem as **Strategy**, thus making **Strategy** redundant. Both aim at encapsulating 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

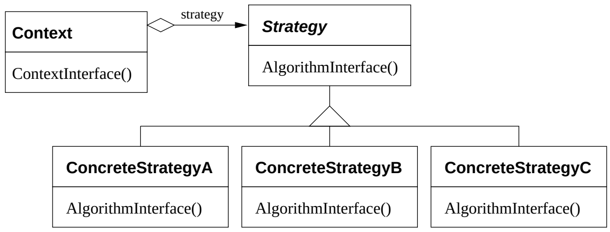


Figure 6: The **Strategy** Design Pattern

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 [Gamma et al. 95]; 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.

2.2.4 Results

For each of the Design Patterns in [Gamma et al. 95], we have in [Agerbo97] 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 the table in Figure 7.

The seven Design Patterns marked as LDDPs are good design ideas that are covered by generally known language constructs. These Design Patterns are the ones that should spur on the evolution of programming languages to encompass stronger language constructs.

There are only two Design Patterns marked as RDPs, but there will be many more once the analysis is applied to other catalogues of Design Patterns. Some of the Design Patterns that will fall into this category are obvious variants of Design Patterns, such as "State Patterns" ([Dyson et al. 96]) and "Variations on the Visitor Pattern" ([Nordberg96]) where the authors propose several variations on the Design Patterns from [Gamma et al. 95]. Other Design Patterns will only after thorough reading prove to be variations on existing patterns. An example is the Design Pattern "Late Creation" ([Bäumer et al. 96]) which in fact is a variation on the **Abstract Factory** Design Pattern. The Design Pattern proposed in this paper lies very close to what is described in

Name	Category	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 7: Analysis of Design Patterns from [Gamma et al. 95]

the Implementation section of [Gamma et al. 95] as *Defining extensible factories*. Finally there is some likelihood that some of the designers of Patterns over the years will have come up with almost identical ideas and solutions, but have named the resulting Design Patterns according to their own taste. This is already acknowledged in [Gamma et al. 95] where each Design Pattern has a section with the name **Also Known As**.

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. 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.

3 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 is known as the *Tracing Problem* and has been discussed in [Bosch98] and [Soukup95].

A proposal to solve this problem is to use “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 it is not necessary 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 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 those that 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. 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.

3.1 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.

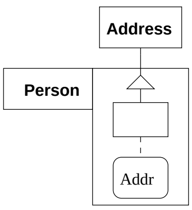
In BETA this advantage could also be achieved with composition by creating a singularly defined part object Addr:

```
Addr: @ Address(# ... extension ... #)
```

as an instance of a locally defined anonymous subclass of Address. An example of this and the structure in our expanded OMT-notation is shown below.

```
Address:
(#
   Street:@ Text;
   Town:@ Integer;
   printLabel:< (# do inner; (*print Street, Town*) #)
#);

Person:
(#
   Name:@ Text;
   Addr:@ Address
         (# printLabel:: (# do ... ; (*print Name*); #)
         #)
#)
```



Since printLabel is defined as a virtual method in the class Address, and Addr is a singular instance of a locally defined subclass of Address it is possible to further bind printLabel in Addr. This way, the method printLabel 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. Furthermore it will always be evident that a Design Pattern is used, since the application-specific hierarchy will be built by inheritance whereas the roles played from the LDPs will be played by nested objects. Using multiple inheritance this distinction will not be as easily made.

3.2 Implementing the LDPs

In [Agerbo97] we have discussed how and to what extent the Fundamental Design Patterns could be placed in a library of Design Patterns. In this article 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 LDPs 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 LDPs 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 [Gamma et al. 95], and require the book at hand for full understanding.

3.2.1 Flyweight

The application-dependent issues to consider when making an LDP out of **Flyweight** 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.

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

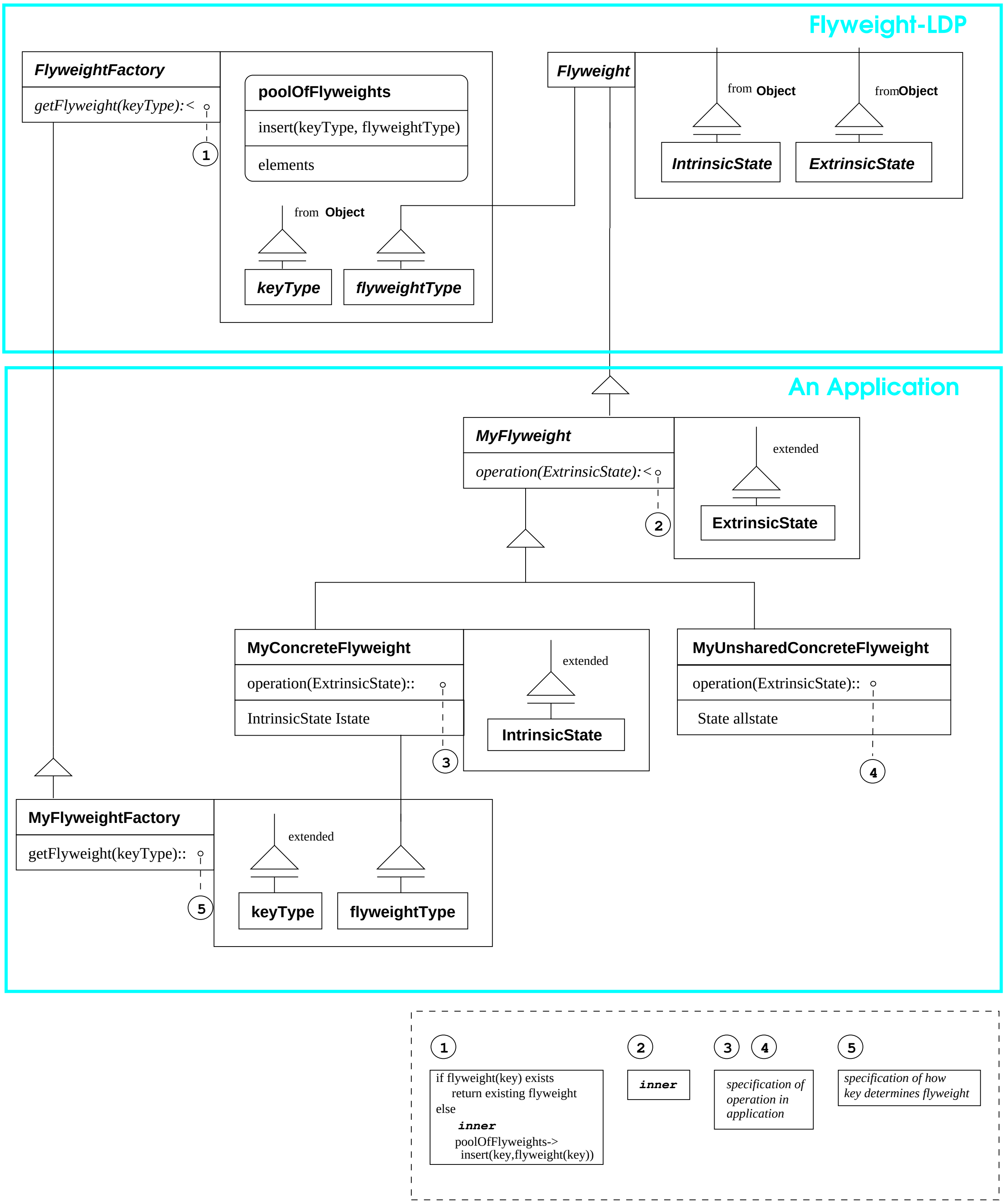


Figure 8: **Flyweight**-LDP

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.

3.3 Discussion

In section 3.1 it is described how the LDPs are reused by letting the classes in the application take on roles from the Design Pattern by nesting instances of locally defined anonymous subclasses of the LDP. This means that 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.

A number of the language features in BETA prove themselves especially useful in connection to the LDPs by supporting genericity and reuse of models. This is further elaborated on in [Agerbo97] where we show how the intent of a Design Pattern could be encapsulated as an LDP for 10 out of the 12 Fundamental Design Patterns, and that it in 6 out of these 10 cases is due to *virtual classes* and *nested classes*. That it in this way is possible to reuse enough of a Design Pattern for it to be applied directly from an LDP while keeping the intent of the Design Pattern intact reduces the *implementation overhead*, a problem connected to the use of Design Patterns identified by Jan Bosch in [Bosch97].

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] formulate good design- or implementation-ideas, the language features that support them must be considered flexible and useful in relation to reuse of design.

4 Related Work

Most efforts concerning Design Patterns have so far 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" ([Gil et al. 97]) 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. This 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 [Agerbo97]. 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 will it be hard to agree on what Design Patterns are formalisations over inherent object oriented ways of thinking — [Gil et al. 97] 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 almost identical 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.

The tracing problem has become a generally recognised problem within the field of Design Patterns. Görel Hedin has proposed a technique for formalising Design Patterns which allows the Design Pattern applications to be identified in the source code ([Hedin97]). 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 or not a Design Pattern has been applied correctly. The largest difference between this approach and ours is that our approach will partly reduce the implementation overhead, whereas Hedin's solution can work as a debugger for Design Patterns where our solution can not guarantee that the Design Patterns are applied correctly.

Jiri Soukop has also tried to solve the tracing problem. In his article "Implementing Patterns" ([Soukup95]), he proposes to build 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*.

5 Conclusion

The objective of this article is to regain the benefits of using Design Patterns:

1. They encapsulate experience.
2. They provide a common vocabulary for computer scientists across domain barriers.
3. 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. 12 of the 23 Design Patterns from [Gamma et al. 95] 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.

We have in this paper described how the use of LDPs can preserve the Design Patterns in a library, and how the use of these would guarantee automatic annotation in a program that some object participates in an application of a Design Pattern. Furthermore, we claim that the presence of *nested classes* and *virtual classes* in the programming language will reduce the implementation overhead, since these two language features makes it possible to capture the intent of the Design Pattern in the collaborations between the objects, and to inherit the interdependencies in an application. This is described in detail in [Agerbo97], where we have shown that out of the 12 Fundamental Design Patterns, 10 can be implemented as LDPs preserving the intent of the Design Pattern.

Thus the use of LDPs will provide us with a means of ensuring the third benefit of Design Patterns, and it will to some extent eliminate the implementation overhead if the chosen implementation language possess the necessary language abstractions.

References

[Agerbo97] Ellen Agerbo and Aino Cornils (1997): *Theory of Language Support for Design Patterns.* Department of Computer Science, Aarhus University.

[Alpert et al. 98] Sherman R. Alpert, Kyle Brown and Bobby Woolf (1998): *The Design Patterns Smalltalk Companion.* Addison-Wesley Publishing Company.

[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.

[Bosch98] Jan Bosch (1998): *Design Patterns as Language Constructs.* Journal of Object Oriented Programming, May 98 pp. 18-32.

[Bäumer et al. 96] Dirk Bäumer and Dirk Riehle (1996): *Late Creation: A Creational Pattern.* PLoP '96.

[Coplien94] J.O. Coplien (1994): *Advanced C++: Programming Styles and Idioms.* Addison-Wesley, Reading, MA.

[Dyson et al. 96] Paul Dyson and Bruce Anderson (1996): *State Patterns.* PLoP '96.

[Gamma et al. 95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995): *Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company.

[Gil et al. 97] 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.

[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.

[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.

[BETA93] O. L. Madsen, B. Møller-Pedersen, K. Nygaard (1993): *Object-Oriented Programming in the* BETA *Programming Language.* Addison-Wesley Publishing Company.

[Nordberg96] Martin E. Nordberg III (1996): *Variations on the Visitor Pattern.* PLoP '96.

[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.