Vrije Universiteit Brussels – Belgium
Faculty of Sciences

In collaboration with

Ecole des Mines de Nantes – France

Universidade Federal do Rio de Janeiro – Brazil

2000

A contextual help system for assisting OO designers in using design patterns

A Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science (Thesis research conducted in the EMOOSE exchange project funded by the European Community)

By: Olivier Motelet

Promotor: Prof. Theo D'Hondt (Vrije Universiteit Brussels)
Co-Promotor: Prof. Cabral Lima (Universidade Federal do Rio de Janeiro)

A CONTEXTUAL HELP SYSTEM FOR ASSISTING OO SOFTWARE DESIGNERS IN USING DESIGN PATTERNS

Abstract

Design patterns are very useful and important tools for improving software design, but they have a complexity cost. Considering that, design patterns should be applied only when it is really necessary.

This thesis is the result of a research effort and draws the basements of a contextual help system that assists OO software designers in using design patterns in a right way. Our system suggests the use of design patterns not only as limited reusable pieces of design but mainly as a design style proposal which should help the designer without limiting his creativity.

Our project proposes an original structure for describing design patterns. On the base of this presentation, it introduces an advanced contextual support for evaluating the suitability of design patterns and for navigating through them towards solutions.

Keywords

Design patterns, design style, design patterns learning, design patterns presentation, navigation through the knowledge, contextual help, open assistant, creativity, innovative design.

ACKNOWLEDGMENTS

This thesis project was the context of one of the most exciting experience of my life. It was not only a great scientific adventure full of doubts and joy, but also a significant advancement of my life perceptions.

This period could not have been so beneficial without the support of a so human environment.

I am particularly thankful for the constant and efficient support of Prof. Cabral Lima during this thesis. He was very attentive to all my needs and always succeeded in motivating my enthusiasm. He is sincerely a worthy person.

I am also grateful for the generous welcome of the UFRJ and UENF which provided me all the necessary accommodations for working in great conditions.

Thanks to my parents and sister who never end to demonstrate their limitless love.

“Muito obrigado, jà tenho saudades de vocês” for all my friends of Campos who accepted me as I am without any other expectations.

I would like to dedicate this thesis to the sundry winds of Brazil which offer me so much peace.

CONTENTS

DETAILED CONTENTS

1. INTRODUCTION

1.1. Prolegomenon

This thesis project is a partial fulfillment of the requirements for the degree of Master of Science in Computer Science and for the degree of engineering of the Ecole des Mines de Nantes. As a part of the EMOOSE exchange project funded by the European Community, this research was performed with the collaboration of the Universidade Federal do Rio de Janeiro (UFRJ).

During the first few weeks of this thesis, I was pleasantly welcomed by the computer science department of the UFRJ, one of the more important university from Brazil. They offered me all the accommodations necessary for proceeding to this thesis project.

This thesis was entirely supervised by Pr. Cabral Lima in his quality of Professor of the computer science department of the Federal University of Rio de Janeiro (UFRJ).

In this time, Pr. Cabral Lima kept scientific contacts with the Universitade Estadual do Norde Fluminense (UENF), former member of the Institution Network of the EMOOSE project. This young university is located in Campos, a calm and pleasant 500000 inhabitants city of the Rio de Janeiro state. As a matter of fact, the UENF is really an environment favorable for the development of a thesis.

On the kind suggestion of Pr. Cabral Lima, the UENF invited me in their establishment. Therefore, during the four last moths of this thesis, I could work in the very pleasant atmosphere of this institution.

1.2. Motivation

In the last decade, the significant benefits of software reuse were rapidly considered by the software community. A lot of efforts were done in order to develop better programming languages, tools, and techniques: COBOL, structured analysis and design, then Object Oriented languages and finally Object Oriented Architecture & Design processes. Those methods demonstrated especially the importance of *patterns*, tools primarily introduced in the architecture domain, as encapsulations of general adaptable solutions to common problems. In this direction, Gamma and his colleagues [GoFo95] introduced the *design patterns*, as descriptions of communicating objects and classes customized to solve general design problem in a given context.

The software community recognized the communication benefits of design patterns and their use was rapidly generalized. Presently, design patterns are significantly widespread and well-known. However, their application is not simple since they have a complexity cost. Then still today, it is necessary to let important learning period to the designers in order to avail a correct use.

During our research, we encountered several tools for facilitating the application and implementation of design pattern, but, it seems that there is no concrete support for using the design patterns. Then, the choice of a design pattern for solving an encountered design problem is still completely legating to the designer's skills. In response to this appearing deficit, this thesis plans to offer a support for using the design patterns.

1.3. Objectives and approach

Considering the general enthusiasm of the software community about design patterns, it may be interesting to notice the deficit of concrete support for using design patterns. Actually, the problem is quite complex since it involves a determination of the design patterns' use.

1.3.1. A position on the design patterns' use

The positions on this topic are cleaved and are subjects of numerous discussions in the design patterns mailing lists ([DpLi00], [Wiki00]).
To review the situation, the debate revolves between two extreme positions:

- considering design patterns as strict design solutions to accurate situations.
- considering design patterns as elusive design models supporting a refinement of the design.

It is essential to give some stable theoretical basements to any practical work. Then, before starting any study involving the design patterns' use, it should be necessary to take a clear position on this point.

Therefore, after analyzing in details various views on the design pattern's use, we adopted our own one: we consider design patterns as significant communication tools but we believe that **their use cannot be fixed.**

We based rigorously our thesis on this median assertion. This position did not direct us to obvious technical solutions, since it does not permit any strict and computable uses of design patterns. However, we believe in the accuracy of this position and consider it as a substantive way towards reliable results. Consequently, despite the appearing difficulties, we still carried on in this direction.

1.3.2. Objectives

The first title of this MSC thesis was "*an intelligent tutoring system to help OO system designers using design patterns*". All along this project, we kept our position on the deign patterns' use. Then, it appeared progressively that it was more realistic to consider a help system for using design patterns more as an open assistant than an intelligent tutoring system ([RLKS98]).

1.3.2.1. Intelligent Tutoring System

An Intelligent Tutoring System (ITS) is globally a collaboration between three main modules ([Lima92], [Lima97],[Lima98]):

- *Domain Knowledge model*: it is the base of knowledge of the system.
- *Pedagogic knowledge model* : it contains the pedagogic skills of the system.
- *User model*: it is an encapsulation of the behavior and reasoning of the user.

The principle of a tutoring system is generally to direct the user towards an "accurate behavior" and to allow him to acquire knowledge and skill about the domain of the ITS.

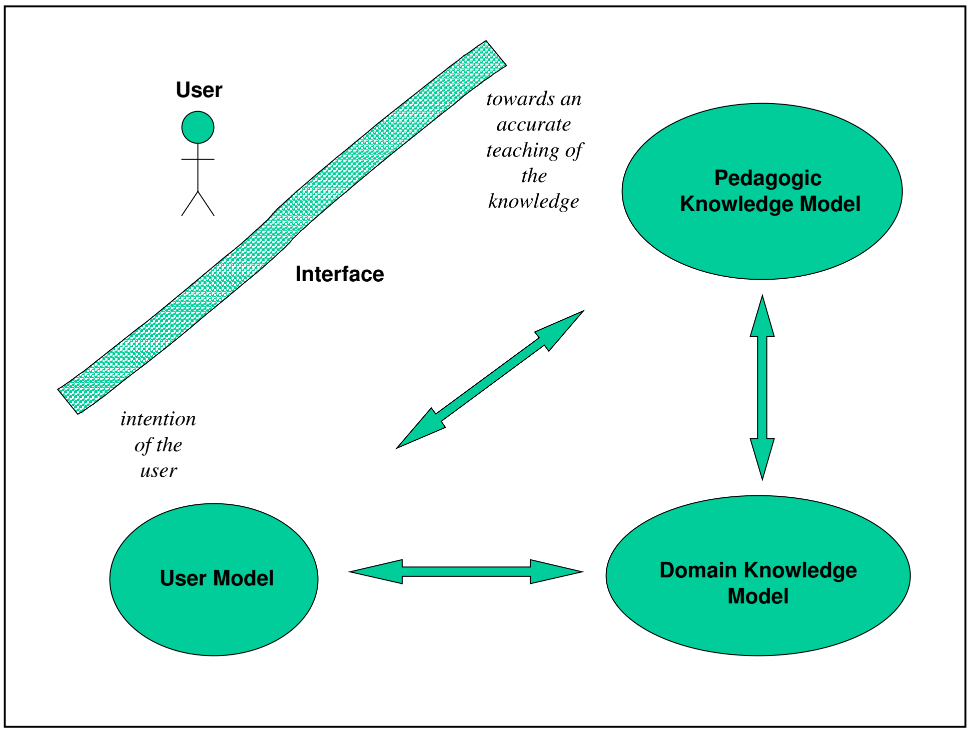

User
towards an accurate teaching of the knowledge
Pedagogic Knowledge Model
Interface
intention of the user
User Model
Domain Knowledge Model

Figure 1: Intelligent Tutoring System

The Domain Knowledge Model knows how to solve the problem and owns a real knowledge representation of the domain. The tutoring is the interface with the user: it directs the session, develops pedagogical strategies, suggests exercises, can explain and understand the mistakes of the user. The user must be also modeled since the system must be able to occur the level of the user's knowledge. The main difficulty of such a system is the necessity of representing the user's knowledge.

1.3.2.2. Present situation

According to our position, the design patterns' use cannot be fixed and the designer must keep the possibility to utilize them for innovative purposes. As a consequence, an Intelligent Tutoring System for using design patterns could not identify nor the intention of the user nor an accurate use of design patterns.

Therefore, we decided to develop a prototype of " *a contextual help system for assisting the OO designer in using design patterns*". Our new objectives could seem less ambitious but they are much more realistic in the specific context of the design patterns' use.

1.4. Thesis Report Organization

In this thesis, we study the way to enhance the learning of design patterns and the research of design patterns solution to design problems. On those basements, we propose a prototype of help system for assisting the OO designer in using design patterns.

In the present chapter, we presented the objectives of this thesis and our approach. The additional chapter of this report are:

- *Chapter 2:* "State of the art - Positions on the use of Design Patterns"

  This chapter attempts to define clearly design patterns, to present the proceeding design patterns related activities and to take position on the design patterns' use.

- *Chapter 3:* "Helping the designer to know and use Design Patterns"

  According to the position we take in the second chapter, this chapter discusses the topics susceptible to enhance the design patterns' use and learning.

- *Chapter 4:* "A reoriented presentation of the Design Patterns"

  As a result of the discussion of the previous chapter, this chapter introduces a proposal of new format for describing design patterns oriented for promoting an easier and more accurate use of design patterns.

- *Chapter 5:* "A help system"

  This chapter introduces the prototype of help system we have developed, as a support for the new presentation of design patterns presented in the previous section, and as a substantial assistant for helping the designer to use design patterns.

- *Chapter 6:* "Conclusion"

  This section summarizes our research and opens to further directions to be got for future research projects.

- *Chapter 7:* "Industrial and economic impact of our research"

  This final chapter pretends to expose the significance of our work in the software industry.

This report contains also the following annexes:

Annex 1	Example of OO Design Pattern: Composite.
Annex 2	Consequences and relationships of Design Principles.
Annex 3	Consequences and relationships of Design Patterns.

Annex 4	Intent description of the design pattern.
Annex 5	Shot screens of Design Patterns and Design Principles.

Lists of tables, figure and references can be found at the end of this report.

2. State of the art - Positions on the use of Design Patterns

This first chapter positions our work in the research field of design patterns. It is divided in four parts.

The first part defines the notion of design patterns as a specialization of the concept of patterns. Then, a second part presents the general way design patterns are used by the software community. The third part is a state-of-the-art of the research in design patterns related activities. Finally, the last part explains our position on the usage of design patterns.

2.1. What are Design Patterns?

Presently, the Design Patterns are widespread and well-known tools. So well that the majority of the software community has more or less an idea of them. However it may be significant to begin this chapter by defining them to adjust all the thoughts.

On that purpose, we will first talk about the notion of software patterns and then we will define the Design Patterns themselves.

2.1.1. Software patterns

2.1.1.1. Origins

The architect Christopher Alexander defined for the first time, in [Alex77], the concept of pattern as :

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such way that you can use this solution a million times over, without ever doing it the same way twice."

He added few years after in [Alex79]:

"Each pattern is a three part rule, which express a relation between a context, a problem and a solution."

The patterns were born as a way to encapsulate general adaptable solutions to common problems. This notion targeted first the architecture domain. But rapidly, other domains like management, communication or software extended the concept to their own use.

2.1.1.2. Patterns' impacts in the software community

In the early 1990's the software community began to show interest in the concept of patterns. The following years, patterns entered to the vernacular as a result of seminars, conference sessions and journal publications [CoSc95].

Software patterns appeared numerous and various. Consequently, an exhaustive lists of the existing software patterns should be too large to enter the scope of this thesis. However, it is interesting to give an overview of the scope of the software patterns. On that purpose, it is pointed out the most significant differences between those approaches as:

- *apply to different phases of software development*

  Pattern approaches are not bound to some specific part of the software development. They can be find in each activity. Patterns documented today range from those that concern requirement engineering to those that concern process management or software architecture. Some approaches also include several phases and are thus means for transformation.

- *have different level of relationships*

  Some approaches present stand alone patterns that relate to each other to make up a language or a system. Pattern languages can serve for example for the user guide of a software. A first pattern presenting an overview of the system points to other patterns describing more specific features. Those ones will then point on patterns presenting still more specific characteristics and so on.

- *are more or less domain-specific (dependency)*

  Some patterns describe problems that are more or less domain-specific. For example a pattern that describe how to use a database is far more specific than a pattern that describes how to create a complex object.

- *are described differently (format)*

  There are numerous forms of describing a pattern. Of course this depends on which type of problem the pattern addresses – a pattern that describe how to organize people to manage a software project will differ from a pattern that describes how to arrange a three layered architecture. But there are also format differences in patterns that addresses similar problems.

2.1.1.3. A Software Pattern Example

In order to give more consistency to this software patterns' overview, we suggest to look at an example: the Software Design Pattern *Composite* (cf. Annex 1). For positioning *Composite* in the scope of software patterns, we precise it as a pattern

- applying to the design phase,
- presenting eventual but not compulsory relationships with other Design Pattern,
- not domain-specific,
- described in a very specific format.

Composite is a Design Pattern and in those terms, it enters directly into the concern of this thesis. In the next section, we present more specifically the origins and the definition of Design Patterns.

2.1.2. Design Patterns

2.1.2.1. Origins

One of the main attraction of the object paradigm is to program in terms of real entities and so to be able to easily reuse them. However in complex systems, as E. Gamma said:
"the strict modeling of the real world leads to reflect today's realities but not necessarily tomorrow's. The abstractions that emerge during design are key to making a design flexible."

As a matter of fact, the designers are very often dealing with complex systems involving several objects. And they desire the reusability of more than one object but of set of collaborating objects. In this case, the need for flexibility becomes a complex problem and involve structured interactions between objects. For a same context and a same problem, we can find many different solutions depending on the designer's own style.

The experience has shown that it is difficult to give general rules defining what is a good style ([Eden98a], [Ga&al96]). Indeed, the criteria which could make a good design (flexibility, clarity, efficiently...) are often contradictory.
For example, it is generally observed that increasing reusability of several object will also increase the complexity of the system. Practically it is due to the fact that increasing the flexibility involve very often the number of objects growing up. Consequently, the interactions are more numerous and complex which make the design more difficult to understand.

In this example, we could consider that a good solution for a problem should be a solution increasing the less as possible the complexity of the design in taking into account the needs for flexibility of the system.

Finally, designing with style is not something that designer can apply in a cookbook fashion. And it takes generally a long time for novices to learn what good object-oriented design is all about. What make a good designer is not only his knowledge about the object-oriented paradigm but also his experience. Experienced designers will not solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past.

The idea of Gamma and his colleagues, known a the Gang-Of-Four, was to capture and formalize those solutions. On that purpose, they introduced the notion of Design Patterns in the early 1990's [GoFo95].

2.1.2.2. A Definition

The GoF introduced the notion of design patterns as:

"The description of communicating objects and classes customized to solve general design problem in a particular context." [GoFo95]

The design patterns encapsulate reusable strategies of interaction between objects and describe them with a possible intent in a particular context. They added also that

"each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change." [GoFo95]

The Design Patterns target some specific type of purpose: enhancing the reusability. This particular proposal of variation is generally described as their intent.

They furnished with this definition a set of twenty-three Design Patterns they described uniformly. Our previous example *Composite* is one of them.

As a matter of fact, the benefits of Design Patterns appeared not only as a way to encapsulate design experience but also to provide a common vocabulary for designers across domain-barriers.

2.1.2.3. Available Design Patterns

Since their introduction, the number of Design Patterns has continuously increased involving a general confusion which rapidly prejudiced the idea of common vocabulary. Consequently, the pattern community worked on some rules to limit the set of Design Patterns. As a result, some researches even critic the mythic set defined by the GoF [AgCo98].

In this sense, it is inevitable for us to make also a restriction. Indeed, it could have been harmful to base our work on a so confusing situation. Consequently, we made some choice in the diversity of design patterns.

Despite of the profusion of Design Patterns, we noticed that the GoF's ones remain the most widespread and known patterns in the software community. Then to accord with the majority, we will limit our work to this set of Design Patterns.

2.1.3. Description of GoF's Design Patterns

The specification of each one of the twenty-three patterns listed in the GoF patterns catalog [GoF 95] is formatted along a fixed structure, described in section "Describing Design Patterns" in the Introduction chapter as the following:

- ***Pattern Name and Classification***
  The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary. The pattern's classification reflects the scheme we introduce in Section 1.5.

- ***Intent***
  A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

- ***Also Known As***
  Other well-known names for the pattern, if any.

- ***Motivation***
  A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

- ***Applicability***
  What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

- ***Structure***
  A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [RBP+91]. We also use interaction diagrams [JCJO92, Boo94] to illustrate sequences of requests and collaborations between objects. Appendix B describes these notations in detail.

- ***Participants***
  The classes and/or objects participating in the design pattern and their responsibilities.

- ***Collaborations***
  How the participants collaborate to carry out their responsibilities.

- ***Consequences***

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

- ***Implementation***
  What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

- ***Sample Code***
  Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

- ***Known Uses***
  Examples of the pattern found in real systems. We include at least two examples from different domains.

- ***Related Patterns***
  What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

This format is consistent with the definition of design patterns. Indeed, the design patterns encapsulate some design strategies in a particular context: the design strategies are described in the sections *structure, participants* and *collaboration*, and the particular context is defined in *intent, applicability* and *consequences*. Some other sections serve more for the understanding of the design patterns as *name, motivation, related patterns* and *know uses* do. *Implementation* and *sample code* attend to facilitate the use of the design patterns.
We notice that this description format is not only attempting to suit to the definition of the design patterns but also have some learning and use guide purposes.

2.2. General use of Design Patterns

The following part discusses the present use of the design patterns. On that purpose, it first presents the software community' opinion about the design patterns. Then it explains more precisely how they are used at present.

2.2.1. A General Enthusiasm

OO Design Patterns became quickly a systematic tool in the OO software industry. The huge success of the GoF's book [GoFo95] is a good indicator of this ebullience[1].

[1] More than 150,000 copies of the book were sold since its publication. Moreover, the Gang of Four's contribution to software development was acknowledged at the Software Development '98 Conference in San Francisco.

This phenomenon is quite understandable when we look at the major benefits that design patterns brought to the software community ([GoFo95], [AgCo98]):

- They encapsulate design experience and consequently allow the reuse of general design strategies of quality.
- They provide a common vocabulary for computer scientists across domain barriers.
- They enhance the documentation of software designs.

Those benefits were supported by numerous reports of successful use of design patterns in real projects [CoBe96].

The software industry understood rapidly the interest of using the design patterns. Several industrial experiences revealed that software development managers ask their employees to use the most as possible the design patterns. Even if a design pattern not properly used can lead to a very bad design, the benefits of the design patterns are so well-known in the software community that the simple fact to use them becomes more and more a testimony of quality. In fact, the software industry tends to systematize the use of Design Patterns in OO designs.

2.2.2. Using Design Patterns

Using the Design Patterns means to apply them in the design or the refactoring of one's software. However, as we will see in the first part of this chapter, systematic use can lead to harmful situations. Consequently for using a design pattern, a designer should know both when to use it and how to use it.

2.2.2.1. Positions on the use of Design Patterns

The general enthusiasm tending to a systematic use of the design patterns quickly warned the software community. Thus between others, Lutz Prechelt and Barbara Unger [PrUn98] made a series of experiments on the use of design patterns to expose clearly the danger of misusing. They arrived to the following statements:

- a careful documentation of the usage of the design pattern is highly recommended since it pays off well during the maintenance
- design patterns can be beneficial even when an alternative solution appears to be simpler *but* unsuitable application can also be harmful in other situations.

This last point resulted in a general warning :

"The resulting practical advice calls to apply common sense when using design patterns instead of using them in a cookbook fashion."

The design patterns cannot be used systematically but in a reflexive manner accorded to the specific situation. As Gamma said *"Patterns have costs (indirection, complexity) therefore [one should] design to be as flexible as needed, not as flexible as possible."*

2.2.2.2. Knowing when to use Design Patterns

For using design patterns, the designer must recognize the situations in which a design pattern should be necessary.

For instance, he should stop each time he encounters a complex problem and looks at the set of Design Patterns to see if one of them should help to solve completely his problem. However there is in the sole catalog of the GoF's book twenty-three Design Patterns. Considering the size of the description of one Design Pattern (Annex 1), with the entire set the user faces a very huge amount of information.

The GoF book furnished a synthetic list of the Design Patterns where they are described only with their intents. Unfortunately, anyone who has already used this list, knows that the intent of a design pattern is very often insufficient to say if the design pattern should solve the problem.

Moreover stopping when one arrives to a complex problem is not always a reliable method. The design patterns solves general problems which could be applied at different levels of granularity. Maybe the user should identify a problem and not find a solution with any design pattern only because the real problem comes from a higher level of granularity than the one he identified.

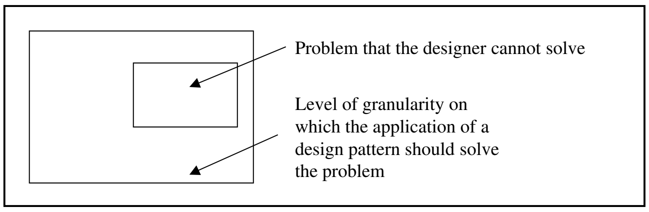

Problem that the designer cannot solve
Level of granularity on which the application of a design pattern should solve the problem

Figure 2: Granularity of problems detection

Finally, the really best way remains to know the whole design patterns' catalog to recognize directly the situations involving the use of one of them. It does not mean that the designers should know in details all the design patterns but that he should have a general knowledge of their application field. Actually we believe that for applying properly the design patterns, the user needs to have a synoptic knowledge of the entire set.

2.2.2.3. Knowing how to use Design Patterns

The use of a design pattern includes also its application on the specific problem of the designer.
The design patterns are describing design strategies for solving general problem. As such they are not an implementation we could reuse directly. For solving a particular problem with a design pattern, the designer needs to adapt the abstract design structure encapsulated in this design pattern to his specific situation.

Applying a design pattern is to implement this design pattern according to a particular context: the context of the problem.

Consequently, for applying a design pattern, the designer must understand the role of each object of the abstract design structure and their collaboration. Afterwards, he should be able to adapt the design patterns to his problem in identifying the participants of the abstract structure with an actor of his particular problem. For example, to apply the *Composite* design pattern (presented in Annex 1), the user needs to understand the role of the objects *Component*, *Leaf* and *Composite* and their interactions. Then he should be able to recognize which object in his specific problem could take the role of *Component*, which ones the role of *Leaf*, and which ones the role of *Composite*.

For most of the people, this process is still a hard task. Indeed the designer needs to understand so well an abstract structure that he should be able to implement it according to its own problem. Because people generally like to have the feeling to stay in a known context, they find often easier to understand a concrete application than directly the abstraction. Then the classical mechanism for the application of a design pattern becomes:

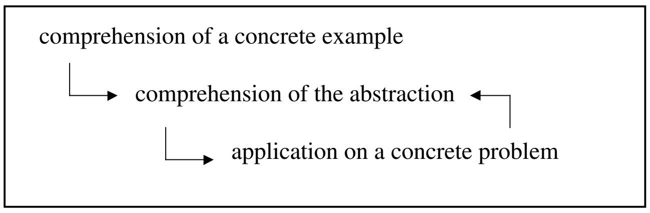

comprehension of a concrete example
comprehension of the abstraction
application on a concrete problem

Figure 3: Induction process

In the same manner that a first example will help to understand the abstract design structure, attending to apply concretely the design pattern will still more refine the comprehension. Definitely, the understanding of a design pattern is involved by induction.

Finally, we notice that for being able to apply correctly a design pattern, the designer needs to see and participate to some concrete experimentation of its abstract structure.

2.2.2.4. Tutorials for learning about design patterns

From the scope of our research, it seems that the tutorials for learning about design patterns are limited to books and courses. We did not find any tutorials using other supports (for ex: electronic).

The books are catalogs of description of design patterns. The two major ones are the GoF [GoFo95] and the Java tutorial [Coop99]. They use largely examples to support the descriptions. We have already seen that design patterns' descriptions in the GoF book contain two examples: one in the section *Motivation* and one in the section *Sample Code*. The section *Implementation* is also available to help the designer to apply the abstraction on his practical context. The Java tutorial is still more example-oriented since almost each part of the description is enriched with a concrete example and some Java code.

The courses [Keri00] are based on exercises. They appears often as supplementary supports to the reference books. The design patterns are presented in a logical order frequently from the most general to the most specific. The students experiment each design pattern on their own. Using open questions, the courses focus on a deep understanding both of the mechanism behind the design pattern and its application field.

Those tutorials target on the learning of all the design patterns in a whole. They do not bring any new solutions to apply the design patterns properly and correctly and accord with the two propositions we discussed previously:

- necessity of a synoptic knowledge of the design patterns' set,
- necessity of experimenting concretely the design patterns via examples and applications.

2.2.3. Conclusion

The general enthusiasm in front of the design patterns' benefits lead to a more systematize application of these tools. However the design patterns are not obvious abstraction (that is why they are so useful) and their application still remains quite complex and requires a good knowledge of the whole set to be done properly.

Consequently, as described in the next chapter, the research in this field is now mainly centered on some CASE tools (Computer Aided Software Engineering) for design patterns.

2.3. Design Patterns related activities

2.3.1. Introduction

In the literature, we found four categories of research in design patterns' related activities:

Specification: The design patterns was introduced without formal basements. The *specification* domain is attempting to the formation of a rigorous specification language for describing them.

Application: This activity works on the automation of the application of the design patterns. Concretely it aspires to generate source code that implements a particular design pattern according to the specification of the programmer.

Recognition: Design patterns are also communication means between designers. In this sense, the *Recognition* domain aims to track the instances of specified pattern in the source code of legacy system and then to enhance the communication.

Suggestion: The use of the design patterns is still a complex task which requires a good knowledge of these tools. The *suggestion* activity attempts to automate the research of design pattern for solving problems.

The first research direction works on the design patterns themselves and the three others pertain to the ways CASE tools can facilitate the practice of design patterns.

The next part presents the state-of-the-art of the *specification* activity. Then we will analyze the state of the research in the domain of the CASE tools.

2.3.2. *A specification of Design Patterns*

In this part, we review the point that reaches the research in terms of specification of Design Patterns. First we will explain the necessity of specifying the Design Patterns. Then we will discuss the limits of the specification and their reasons inherent in the Design Patterns themselves. Finally we will describe briefly the different research works we encountered in this domain.

2.3.2.1. Some Needs for Design Patterns' specification

As we saw in § 2.2.2.2, the highly detailed descriptions of the Design Patterns in the GoF's book are generally inefficient in expressing clearly the application field. This phenomenon, as claimed by Eden [Eden98a], is not surprising since Design Patterns are based on textual descriptions which are inherently ambiguous, and on the induction (via examples) which leads to infinite directions. Consequently, many ambiguities arise which cannot be resolved unequivocally.
For example, the patterns mailing lists [DpLi00] often engage in prolonged discussions whether a particular piece of code manifests an instance of one design pattern or another. For becoming fundamental elements of software construction, the abstraction underlying the design patterns must mature to a definite and unambiguous specification.

Actually this research area attempts to formalize the design patterns to enable [Eden98a]:

- Resolving questions of relationships between patterns, e.g.: Is one pattern a special case, a variation, or a component of another, and so forth;
- Resolving the question of validation (i.e., does "this piece of code" implement "this pattern"?)
- Tool support in the activities related to patterns: Application (implementation) and recognition.

2.3.2.2. Limitations of Design Patterns' formalization

Design Patterns attempt to solve general design problems. They are by definition generic tools over the domain barriers. The context in which the design patterns are applied, is flexible and depends on the particular domain.

The *intent* of a design pattern establishes a relation between the design strategy encapsulated by the design pattern and its general application. But in [GiLo98], Gil and Lorentz highlight that

"the intent transgress the boundaries of exact science, which is limited to "how" rather than "why" questions.

As any tool, the design pattern must have an intent but this intent should be broad, open to variation and never inscribed in the tool itself. "

The intent of design patterns cannot be fixed. Because to fix it is to limit the use of the design pattern to a specific case: the one described by the intent. The general intent of a screw-driver is to drive screws. However, we do not limit its use to that sole purpose and do not hesitate to use it as a lever or as other various thinks.

Ddesign patterns are design strategies, design mechanisms. As such, they can be used for other purposes than the one described in the intent.

As we show it in previous sections, it is necessary to formalize the abstraction mechanisms encapsulated in the design patterns. But this formalization should target the rules that govern the behavior of those abstraction mechanisms and not their uses. We must formalize the design structures encapsulated in the design patterns and not their intent.

For instance, the interest of an abstraction mechanism like the *inheritance* is not to impose only one kind of use: we know how it works and we apply it to our own purpose. Obviously, we are aware of the general uses of *inheritance*: we are not going to reinvent the wheel every times. However, we are not limited to those uses because we know the rules governing the *inheritance* principle and can apply it for other innovative purposes.

Consequently, the intent cannot be fix inside the design pattern definition. And a formal and systematic approach of the design patterns cannot include the intent.

2.3.2.3. Overview of existing specification languages

2.3.2.3.1. Specification of the abstract structure of a Design Pattern

Numerous works were done in the domain of the specification since the introduction of design patterns. Most of them present, however, important limitations.

We notice the work of Helm, Holland and Gangopadhay [HHGa90], which was not targeting specially the design patterns but which gave interesting basis. They defined the notion of *Contracts:* a language extension with representations for function calls, assignments and relation between them. But *Contracts* unfortunately encapsulate only run-time characterizations and not structural relations as design patterns.

Florijn, Meijers and van Winsen [FMPW97] propose to represent the design patterns as abstractions of higher-level: the *Fragments.* A *fragment* is in fact a semantic net, or a graph with labeled arcs, whose nodes stand for the participants and the arcs describe the node's role in the pattern. Each fragment, however, represent not a "true" pattern but only a version thereof. Moreover the fragment model does not suggest any means for reasoning about patterns and their relationships. Then it does not provide an adequate abstraction.

Eden recently presented a work [Eden99] based on all the previous researches which finally defines an unambiguous specification of the design patterns.
He introduced a symbolic logic language for the specification of recurring motifs in object-oriented architecture like design patterns: LePUS [Eden98b].
LePus is particularly interesting because it can express accurately and concisely not only fundamental elements of OO architecture, such as inheritance, class or hierarchy, but also the correlations, such as isomorphisms between methods classes and hierarchies. Therefore, LePus can be used to describe the structure of a design pattern in terms of objects, hierarchies and their interactions. The language is available both in a graphic and textual version.

LePus [Eden99] is presented as a consistent base for achieving the main goals of this research activity:

- defining a reliable and precise formalism for a more rigorous use of the Design Patterns
- furnishing a good support in the development of Case Tools.

2.3.2.3.2. Specification of the relationships

Relationships between design patterns were discussed in several publications. For example, Kim and Benner [KiBe96] discuss variations of the *OBSERVER* pattern from the GoF set, and Rohnert [Rohn96] discuss variations of

PROXY. The authors of the GoF catalog themselves discussed associations between the patterns in the chapter "Organizing the Catalog" and "Design Patterns Relationship" [GoFo95], but the discussion remains very informal. Zimmer [Zimm96] began to classify those relationships. He divided the relations between the patterns of the GoF's catalog in 3 types: *"X is similar to Y"*, *"X uses Y"*, and *"Variant of X uses Y"*. Once again, the division was based only on informal skills and no reliable proofs could support this work.

However, still at present, very often members of the patterns community debate over questions pertaining to relationships between patterns, such as:

- ♦ What is the difference, if any, between pattern *p1* and *p2*?
- ♦ Is pattern *p1* a special case *(specialization* or *refinement*) of *p2*?
- ♦ Does class library *L* contain an instance of pattern *p1* or of *p2* (or of both)?

The relationships are a key notion in the understanding of design patterns and their utilization. For example, a rigorous description of similarities between two design patterns leads to a better comprehension of their utilization. Indeed, the difference between *Decorator* and *Composite* is quite subtle. To express clearly and rigorously this difference should precise the usage of each of them.

However, it is difficult to find a rigorous work available on this domain. On this subject, the Gang of Four recently declared [DpLi00]:

"When we wrote our book, we were trying to hide something. We were trying to avoid talking about one pattern being a specialization of another, or one pattern containing another as a component. We wanted to avoid going meta and just wanted to talk about patterns. … But the world is different now. People want to know the relationship between patterns and we need to tell them. The relationship here is so obvious that we need to emphasize it, not just tuck it away at the end in the related pattern section."

With the work of Eden, the formal basements are now available and the research on establishing clearly the relationships between the Design Patterns is proceeding (Vlissides and al., 2000) but remains in a very early stage.

2.3.2.3.3. Specification of the intent

As we discussed it in § 2.3.2.2, a formal approach of design patterns can be achieved only if the intent is disregarded. Consequently, the research in the formalization of the intent is almost non-existent.

We still notice the work of Gustavsson and Ersson [GuEr99]. They attempt to classify the intents as they are described in the GoF book, and define some redundancies between them. They arrive to an interesting division as it facilitates the use and the understanding of the intents of the GoF. Since their work remains informal, it just serves as a supplementary support to the GoF description of the Design Patterns and does not attempt to define a hazardous relationship with the Design Pattern structure itself.

To conclude, we think that the research in the specification of the Design Patterns reached now interesting basements. But, since the first consistent specification language for this activity, LePus, was introduced very recently, it is a domain which remains in its very early stages. Therefore, it remains a large amount of work to do before being able to use it as a support for the other design patterns related activities.

2.3.3. Case Tools for Design patterns

Design patterns has been rapidly of a great impact on the software community. Their benefits were rapidly well-known and designers used them more and more. To assist this increasing utilization, the software industry is calling for efficient CASE tools for design patterns.

In this part, we will make the point on the research into the recognition of Design Patterns, then into their automatic application and finally into the automatic suggestion of their use.

2.3.3.1. Recognition of Design Patterns in legacy systems

Automatic detection of design patterns in arbitrary source code is very difficult. It deals with what the pattern community generally calls the tracing problem: more Design Patterns are applied, the more difficult it will be to reorganize the structure of the participating design patterns. This phenomenon appears specially because the Design Patterns can be applied at different level of granularity. Some works attended to pass besides this appearing difficulties and furnished first results in different programming languages.

We notice two interesting works of the early stages of this domain: the one of Krämer and Prechelt [KrPr96] and the one of Brown [Brow95].
Krämer and Prechelt produced the PAT system. This is a tool that detects some structural patterns of the GoF in C++ source code. They also provide an output in OMT models and an evaluation of the results by metrics. In the same period, Brown worked on a version for Smalltalk.

Those two works highlighted the fact that working with the structure of programs were not sufficient enough to detect design patterns (particularly for non-structural Design Patterns!). However they demonstrated that working with the behavior of systems rapidly becomes a very hard task. Indeed the information about the behavior is not entirely contained in the languages. For example, the simple fact that Smalltalk has no type make difficult the analyze of the interactions between objects. For those reasons, to analyze the behavior it is necessary to test the system and to deduce the behavior from the results of these tests. Such operations involves a mass of information too excessive to be practically treated. And the detection of Design Patterns needs to go further than the conventional reverse engineering works and to use some sort of Artificial Intelligence strategies.

The work of Pinali [Pina99] give some interesti ng basements in this direction with a code analyzer for Java. But the research still needs to work in this domain to be able to furnish a reliable CASE tool for detection.

2.3.3.2. Insertion of Design Patterns in new systems

The insertion of the Design Patterns is a developed research domain and numerous works attempt to that goal. We will briefly describe here the capital ones.

The first noticeable contribution was the work of Budinsky and his colleagues who presented a tool that supports the application of design patterns by generating code in C++. But that tool was limited to a hard generation of code.
On this point, the work of Florijn, Meijers and van Minsen [FMPW97] (see § 2.3.2.3.1) goes further. Indeed, they furnish an extension of Smalltalk in which the design patterns are considered as abstraction of higher-level: the *fragments*. Their platform assists developer in applying Design Patterns in three ways:

- generating program elements for a new instance of a pattern,
- integrating pattern occurrences with the rest of the program by binding program elements to a role in a pattern,
- checking whether occurrences of patterns still meet the invariants governing the patterns and repairing the program in case of problems.

This last work was the major and more interesting contribution in this domain.

It may be also noticed an interesting commercial tool TogetherSoft [Toge00] for Java/UML. It is an integrated CASE tool which gather design and development phases. They help the user both to apply the design pattern and furnish a trace of the application in the software itself.

2.3.3.3. Automatic suggestion for using Design Patterns

In regard to our investigations in the literature, the work in the automatic suggestion of Design Patterns is much more modest than in the previous areas. It seems that there is no tool which assists really the user. Actually, the research seems to be limited to the analyze of legacy system and the detection of situations which should involve the use of a design pattern.

On this last point we notice the recent work of Correa and Werner [Corr99] which provides a tool for analyzing and detecting defects in the design of system. The tool, “OODPTool”, works on UML code. About the suggestion of Design Patterns they took two directions:

- *Detection of general design defaults that design patterns should solve.* They referred here to the chapter of the GoF's book [GoFo95] "Design for change" where some general design defaults are described with a set of design patterns susceptible to ameliorate the situation. Nevertheless OODPTool does not succeed in suggesting the proper pattern and satisfies in just displaying the suggestions of the book. Therefore, this system do not help in the use of the design patterns but just attempt to detect some design defect which should involve them.
- *Detection of anti-patterns.* The anti-pattern are patterns of bad design. However their number is very small compared to the design patterns' catalog and only few of them provide a concrete solution expressed as a design pattern. Practically, OODPTool could recognize the use of not more than three design patterns using the anti-patterns.

This two directions did not succeed in suggesting efficiently a design pattern as a solution to a detected problem.

Actually, it seems that the research does not, at present, provide satisfying results in the field of the automatic suggestion of Design Patterns.

To conclude this study of the CASE tools for design patterns, it appears that this research field is today well established. Recognition and Insertion of Design Patterns arrived to interesting practical results. However, about an automatic suggestion of design patterns, the results are still unsatisfying. Taking position on the use of the Design Patterns in the next chapter, we will see why we think this domain is limited by the definition of the Design Patterns itself.

2.3.4. Conclusion

The formalization of the Design Patterns made a recent improvement with the introduction of a rigorous specification language : LePus. Except for the intent of the Design Patterns which remains the burning part of them, we believe that this language should serve as a basement towards a general specification of the Design Patterns and their relationships and should rapidly improve this research area.

Similarly, the research in Case tools for the insertion and the recognition should consider this work as a good support to arrive finally to consistent and more generic results. However, like the formalization of the intent, the automation of the suggestion of Design Patterns remains a burning and unsatisfying domain.

2.4. Position on the use of Design patterns

In this chapter, we will clear up our position on the use of the design pattern. In a first part we will expose the difference between design patterns and frameworks. Then we will explain why we think that design patterns cannot be considered as clip-arts. Finally we will discuss the communication purpose of the design patterns.

2.4.1. Design Patterns vs. Framework

Design patterns are description of design strategies which solve some general design problem. Like frameworks, they encapsulate design decisions in reuse purpose. But over this similarity, the GoF book highlighted the differences between the Design Patterns and the frameworks in three major way [GoFo95]:

- *Design Patterns are more abstract than frameworks.*
- *Design Patterns are smaller architectural elements than frameworks.*
- *Design Patterns are less specialized than frameworks.*

In contrast with framework, the design patterns are not executed and reused directly but have to be implemented each time they are used. Since design patterns encapsulate a way of designing, the designer has a lot to learn from them. A Design Pattern is a solution to a general problem and the designer will certainly use it in a specific context related to the problem solved by the pattern. But their use could be still more useful if we consider them also as abstraction mechanisms that should inspire the user.

2.4.2. Design Patterns are not clip-arts

We think that design patterns are not only like clip-arts as some researchers affirm. If we stay in the domain of the painting, a better metaphor to design patterns than clip-arts should be the painting techniques themselves. Indeed, the first time a painting technique is used and encapsulated, it is because it solves a known problem. But does it mean that the painters over the centuries never went out of those fixed purposes? What make innovation and creativity was to apply those techniques for new purposes and to combine them with other painting techniques.

It is clear that the first time a designer will use a design pattern, he will use it for the general goals described in the pattern itself.
As we explain in the part "Knowing how to use design patterns" (see § 2.2.2.3), all the times that a designer apply a design pattern are occasions for learning and understanding more about this pattern. After a certain amount of uses, the user will really understand the abstraction mechanism. Then he will be able to excel the common use of the design pattern to new goals and associations with other design patterns. As Chris Pehura [Pehu00] said " *the best design should be the*

realization of the combination of at least three design patterns ". But such combinations will deeply depend on the context and will certainly ask for innovative uses of the design patterns.

It may be considered that the intent is one of the major benefits that design patterns offered to the software industry: patterns becomes solution to general problems. However we believe that this intent should remain a support and not limit the design patterns to a fixed field of uses.

2.4.3. Design Patterns as communication means

The work of Ellen Agerbo and Aino Cornils, "How to preserve the benefits of Design Patterns" [AgCo98], attempts to insure the three general benefits of the design patterns:

- they encapsulate experience,
- they provide a common vocabulary for computer scientists across domain barriers,
- they enhance the documentation.

On this purpose they restrict the set of the design patterns by defining some criteria towards a quality of the design patterns. They also work on a library of design patterns. About that, they declared that " *design patterns will be fixed, in the sense that it will not be possible to adapt it in other ways than were foreseen when making the LDP [a Library Design Pattern is a design pattern which suits with the restricted criteria they defined]* ".

To establish limits on the set of design patterns may be beneficial but it is contradictory with the fact that design patterns cannot be fixed: they are not frameworks. The general benefits of the design patterns should not be limited to enhancing the communication between designers but be also considered as a way of communicating an abstraction mechanism.

Design Patterns are useful for communication purposes for design. However, they are not clip-arts: their use should not be limited to a fixed intent. Design patterns should help the designer and influence his style. But they cannot dictate the design and limit their use to rigid purposes.

Therefore, the automatic suggestion of design patterns is a delicate task. Indeed, how to automate the suggestion of design patterns, without fixing their intent. However, an help system for assisting the designer in choosing the design patterns may be interesting. But this assistant should insist on presenting the design patterns also as communication tools of abstraction mechanisms.

2.5. Conclusion

Since their introduction, Design Patterns have known a real success. The industry is generally enthusiast by their benefits and tends to use them more and more. But their use and understanding is still a complex task for the designer not comfortable with the set of design patterns.

Consequently, several works were done in the direction of CASE tools for the application and recognition of design patterns. These research area knew recently the introduction of formal basements which are certainly going to improve the consistence of the results.

Nevertheless, if we want to take entirely benefits of the design patterns, their use still remains a subtle task which seems to be far from being systematize. The design patterns appear as very useful tools of communication for design particularly if they are not dictating the design with a fixed use. It is why the task of helping the user in using the set of design patterns is hard.

In the next chapter, we will discuss which directions we must take to help the user without limiting the benefits of the design patterns.

3. HELPING THE DESIGNER TO KNOW AND USE DESIGN PATTERNS

3.1. Introduction

We described in §2.2.2 the way by which design patterns should be used. We retain that a reliable use of the design patterns results from two directions:

- to use them *properly,*
- to use them *correctly.*

We saw that to achieve those purposes, the user needs

- to have a synoptic knowledge of the set of design patterns
- to understand really the abstraction mechanism encapsulated by the design pattern.

In fact, a reliable use of the design patterns is closely related to a good understanding of the design patterns' set. Therefore helping the designer to use the design patterns should mean also to facilitate the understanding and learning of the design patterns' set.
However an help system should not be limited to that aim and should also assist the user in choosing the design pattern appropriate to his problem. This system should help both the novice which does not own a synoptic view of the set, and the expert which wants to see as clearly as possible if a design pattern should fit to his purposes.

This chapter highlights the criteria we think necessary to assist the designer in using Design Patterns. In a first part, we discuss the features enhancing the learning and understanding process of a design patterns. Then we will study how to make a user choosing a design pattern in the entire set.

3.2. Learning Design Patterns

This part highlights the mechanisms and features that make the user learn about design patterns.

As we saw in § 2.2.2.4, the existing Design Patterns tutorials are all based on the same principle: to use accurately the design patterns the designer needs a good understanding of them and a synoptic knowledge of the entire set. As such, those tutorials can also be seen as learning methods.
Here we are first going to review which are the interesting topics of the available learning methods. Then we will discuss other points that, on our opinion, are also necessary to reach an efficient learning and good understanding of the design patterns.

3.2.1. Interesting topics of available learning methods

The existing learning methods are based mainly on the induction. They are based on the principle that the best way to understand an abstraction mechanism is to experiment it in a concrete situation.
Indeed, as explained in § 2.2.2.3, a first example presents the abstraction in a concrete way. Then the practice helps to certify the understanding.

Both example and practice are useful because they present the design patterns in a context well-known by the designers: some practical pieces of design or code. In such a familiar context the designer is more able to see and recognize the design strategies belonging to the design patterns and their advantages.

3.2.1.1. Highlighting the advantages

In a world where the notion of time becomes a predominant factor, we generally like to know the reasons for which we are spending our time. Therefore, before learning a new thing, we need to know what should be the benefit to spend time on it. We believe from our own experience that the same phenomenon appears when using the design patterns.

Of course, some people simply base their trust in design patterns' celebrity, but we believe that to present clearly the advantages of design patterns will stimulate the attention of the majority. Indeed, we begin to be really motivated to learn a design pattern when we have the sensation that it will bring us real benefits.

Example and practice put the design patterns in a practical context with which the designers are familiar. In such a context, they are more able to see the advantages because they can compare the results with their experience. The benefits of design patterns being more explicitly exposed, the attention and the interest of the designer are motivated.

The available tutorials utilize for this purpose the examples in the design patterns' description. But they also discuss distinctly the advantages of the design patterns all along the description of the patterns and in some more specific sections like "Consequences" in the GoF's book.

3.2.1.2. Highlighting the drawbacks

It is really important to highlight distinctly the advantages of a design pattern to motivate the learning of the design mechanism, but it is as much important to show its drawbacks. Just to know about the advantage of a tool is like to drive a race car without knowing there is no brakes: you will go ahead enjoying the power of the engine but how you will look like if you encounter an obstacle which should force you to stop.
It is always significant to allow the user to have a critic look at the tool to be able to use it correctly and properly.

As they do it with the advantages, the tutorials described also the drawbacks of the design patterns. They expose them with some examples and exercises and more distinctly all along the description of the Design Patterns as well as in the "Consequence" section of the GoF's book [GoFo95].

From our analyze of the Design Patterns tutorials, we retain the necessity to highlight both advantages and drawbacks for making understand completely the design patterns.

The consequences of the design patterns are really a key point of their understanding. The opening questions suggested in the courses about Design Patterns [Keri00], serve in showing the effects, positive or negative, of their application under different viewpoints.

3.2.2. Additional necessary items for enhancing the learning

3.2.2.1. Introduction

"When we wrote our book [GoFo95], we were trying to hide something. We were trying to avoid talking about one pattern being a specialization of another, or one pattern containing another as a component. We wanted to avoid going meta and just wanted to talk about patterns." E. Gamma.

In all books and tutorials, design patterns are presented as patterns. However, if the pattern presentation presents numerous advantages (known approach, oriented for reuse...), it has also the effect to show the design patterns as blocks. Therefore the design patterns became for the majority of the users revered design black boxes. The progresses in tools for automating the use of design patterns will certainly contribute to increase this phenomenon.

The formalization of the patterns attempts to the necessary work of demystifying the design patterns. To enhance the comprehension of design patterns, it may be essential to finally expose their cores.

3.2.2.2. Highlighting the design principles behind the design patterns.

The abstraction mechanism encapsulated by a design pattern is the composition of basics design rules. To understand an abstraction, we have two possibility:

- to reason about particular facts, the induction. This is the approach we analyzed previously.
- to reason directly about the mechanism itself.

If we take the example of a mathematical theorem, we have the same two directions to understand it. It is possible to learn it in applying it in different problems, then you will know by the practice the scope of the theorem.
Another method should be to show the demonstration of the theorem. Once you understand the demonstration, you can really grasp the scope of the theorem. Revealing its origin, the theorem is demystified. It can no more frighten because it appears as it is really: a simple combination of small rules you had already integrated. Then the theorem becomes a part of you and ameliorates your knowledge without limiting you to some fixed uses. Now the theorem could increase your creativity.

Of course, design patterns are not mathematical theorems and we are not going to demonstrate them. But we believe that design patterns should be shown also as the combination of other design principles as inheritance, composition, interfacing and so on.
We do not mean that patterns are the design principles themselves: they are the results of applying them in practice to specific real world problems. The patterns teach us more that the deep workings of those design principles. That is what make their singularity.
But we believe that if we allow the user not only to understand the uses of a design pattern but its depths, we offer him the possibility to integrate it and so let him increase his creativity more than limit him to fixed applications.

3.2.2.3. Highlighting the relationships between design patterns.

Going more deeply inside design patterns, as said in §2.3.2.3.2, it appears obvious relationships between the design patterns themselves. The GoF book presents them informally in the “related pattern” at the end of the pattern.

The relationships are very useful in a learning purpose. When two tools are very similar, to see the difference between the two is a way to highlight their subtleties. For example, we take two wines from the same vine but not the same year. We know the wine coming from this vine since a long time and the

differences between the two are very small. But, it is certain that if we get the subtle differences between them, we will know more about each of them; we will recognize their own personality.
To confront two Design Pattern very closed is a way to see new subtleties in the two.

In this sense, the relationships between design patterns must be clearly highlighted. Each relationship is an opening question for the designer: Why this two design patterns are related? What make their difference? Do I really understand the mechanism of the two?

To conclude, we claim that the design patterns tend to appear like magic black boxes because of the format in which they are described in the available tutorials. This phenomenon is very often leading to confusions in their use. On the opposite, we believe that highlighting the design principles constituting the design patterns and their relationships is an interesting way towards a complete understanding of the design patterns.

3.2.3. Conclusion

In order to enhance the comprehension of design patterns, it may be necessary to show clearly:
- their consequences,
- the design principles which compose them.

Moreover it may be considered that the relationships with other design patterns constitute an efficient support for assuring a reliable comprehension.

This part defines some important features we should highlight for attempting to a better understanding of the design patterns. We want also to help the designer to use the design patterns, that is to say to choose a design pattern or a set of design patterns which could fit to its problem. Thus, in the next part, we will discuss the necessary topics to consider for choosing properly the design patterns.

3.3. Choosing appropriate Design Patterns

Solving a design problem with a design pattern is
- to identify the appropriate design pattern in he design patterns' set
- to apply it.

It is easy to identify the already-known design patterns. Indeed, it can be evaluated rapidly if they are appropriate or not. Unhappily there are not so many people who know the twenty-three design patterns defined in the GoF's book. Consequently, it may be interesting to help the designers to choose a design pattern in the whole set.

Choosing a design pattern is to find it inside the set and to evaluate it to assure its suitability. Therefore to help a designer to choose a design pattern means both:

- to help him to evaluate the design patterns according to his specific problem
- to help him to head for a proper solution (a design pattern or a set of Design Patterns)

3.3.1. Evaluating a Design Pattern

3.3.1.1. Introduction

"Patterns have costs (indirection, complexity) therefore [one should] design to be as flexible as needed, not as flexible as possible." E. Gamma

As discussed in § 2.2.2.1, to apply a design pattern not properly can be harmful. Under this viewpoint, the designer must take care of evaluating the design pattern he is planning to use, and so to assure that it will bring him more benefits that inconveniences.

3.3.1.2. Definition of a proper pattern

A proper design pattern offers interesting benefits without creating undesired situations. Therefore to determine if a design pattern is proper, is to know if you are ready to accept its drawbacks for taking benefits of its advantages.

Consequently a proper pattern should both:

- offer interesting benefits
- not result in undesired situations.

3.3.1.3. Consequences and design principles

All the abstraction mechanisms described in design patterns are based on the OO paradigm: they are combinations of basic principles like *inheritance*, *composition* and so on. We notice that those principles carry their own effects. And that those effects will influence directly the consequences of the design patterns.

For example, one of the advantages of *Composite* [GoFo95] is to allow the addition of new features dynamically. However it may be noticed that this feature is not particularly belonging to the design pattern itself, but by the *composition* principle which is used in the design of *Composite*. In the same way, you should critic the fact that *Composite* is heavy because is leading to create a lot of small objects. But similarly, this drawback comes from the *composition* principle.

The design patterns inherit the effects of the design principles which compose them.

Design Patterns are, however, much more than the design principles they use: they are a specific organization of design principles. Consequently, their effects are more that the sum of the effects of the design principles themselves. For example, the fact that *Composite* allows *to treat composite structures and individual objects uniformly* is inherent in *Composite* itself.

The design patterns results in

- some consequences coming from the design principles which compose it,
- some consequences belonging to the design patterns as particular organizations of design principles.

It may be considered that for choosing properly a design pattern, it is necessary to highlight this phenomenon. Indeed to apply *Composite* only for getting the effect of a consequence directly related to the composition principle, results in implementing an useless heavy structure. To avoid undesired complexity, the designer should know what are the consequences really owned by the design pattern itself.

3.3.1.4. Evaluation of a design pattern

Evaluating a design pattern means to estimate which will be the benefits and the negative side effects of the design pattern when applying it in the specific problem of the designer. This estimation is closely related to the user's context. The user must be able to express his needs: which are the advantages he requires and which are the drawbacks he really cannot accept.

On that purpose, a help system must clearly the advantages and the drawbacks of design patterns. On this base, the user will be able to identify which advantages he needs and which he does not, which drawbacks he does not want and which he is ready to accept. This identification will allow him to evaluate the design patterns as appropriate or not appropriate.
For example, a user considers that he needs all the advantages of a design pattern. Moreover, looking at the drawbacks of this design pattern, he does not identify any one which could prejudice the main specifications of his system. Therefore, the user can certainly claim that this design pattern is proper to solve his specific problem.

The check of the consequences of design patterns,

- advantages needed / not needed,
- drawbacks not accepted / accepted,

participates in their evaluation.

In conclusion, the examination of the design patterns' consequences participates in their evaluation. But, in order not to misuse them, the designer should always know which consequences are related to the design patterns itself and which ones are related to the design principles which compose it.

3.3.2. Towards a proper solution

3.3.2.1. Introduction

It is necessary to close the scope of solutions to offer concrete results to the user. Evaluating the design patterns, the designer can distinguish which are the appropriate design patterns for his problem. The evaluation participates in closing the scope of solutions towards the proper one.

But this closing movement is likely not sufficient to help the designer to choose a design pattern. He also requires to be lead towards the proper design patterns. Thus, we must not only close the scope of solutions but also open it.

3.3.2.2. Navigating through the knowledge

Certainly because of the dependency on the time, most of the people have this bad reflex of fixing the scope of their researches towards solutions in their already-acquired knowledge. The help system must consider this factor and open its entire knowledge to the user without limiting him to it. The idea is to meld the knowledge of the design patterns' set with the user's own experience.

For accessing the knowledge, the user needs some gates. One very consistent entrance is the knowledge itself: the navigation through the knowledge allows to learn, to evaluate and to discover it in the same time. In our case, navigating through the knowledge means to navigate through the design patterns, to go from one to another towards a solution.

3.3.2.3. Relationships between Design Patterns

Design Patterns offer a mean to navigate between them: their relationships. One design pattern can be reached from another one with a relationship. For example, *Strategy* can be see as an alternative of *Decorator*. This relationship could lead to explore *Decorator* whereas we were studying *Strategy.*
With the relationships, it is possible to open the scope of solutions, that is to say to offer new suggestions of design patterns to the designer. Then each design pattern becomes in the same time a possible solution and a portal towards new potential solutions.

Each design patterns has a set of relationships with other ones. In the context of the navigation through the knowledge, these relationships may be seen as gates towards other design patterns. However, the purpose is not to make the user navigating through the whole set, but to show new directions which are susceptible to interest him.

Consequently, we must give the user the potentiality to differentiate the directions which are susceptible to lead him towards a solution from the others. It is the nature of those relationships which will allow the user to evaluate the usefulness of the directions. For example, the user evaluates a design pattern and identifies that he needs some of its advantages but cannot accept some of its drawbacks. In this case, a relationship susceptible to help him, would be certainly an *alternative* to the undesired drawback more than, for example, an *association* with another design pattern.
Therefore the relationships between the design patterns can be considered as a useful way of navigating towards a solution if their nature is explicitly exposed.

It may be considered that the navigation through the knowledge could efficiently help the designer of all levels to head towards a suitable solution for his problem. On this purpose, it is necessary to present explicitly the relationships between the design patterns.

In conclusion, for helping a designer to choose a appropriate design pattern to his specific problem, a help system must support the evaluation of design patterns and the navigation through them towards possible solutions. Then the designer could find a satisfying solution considering a positive evaluation and an absence of other possibilities.

3.4. Conclusion

For assisting the designer in using the design patterns, we must help him both to learn and understand consistently the design patterns and to find a solution in the design patterns' set.

In order to enhance the comprehension of design patterns, it appears necessary to express clearly their consequences and the design principles that compose them.

We saw also, in the last part, that helping the designer to find an appropriate solution in the design patterns' set results from a double movement:
- closing the solutions' scope in evaluating the design patterns,
- opening the solutions' scope in directing the user towards other design patterns susceptible to be suitable.

For making possible this process, it is essential to expose explicitly the consequences of design patterns and the nature of their relationships.

This analyze concludes that for enhancing the learning and understanding of design patterns and the research of solution in the design patterns' set, the following features of design patterns must be highlighted:
- their advantages and drawbacks,
- the design principles which compose them,

- the relationships with others.

In this sense, it may be interesting to reorient the classical viewpoint on the design patterns in the direction of those new features. Therefore we should exchange the classical format used for describing the design patterns for a new presentation highlighting efficiently those necessary but disregarded topics.
On that purpose, in the next chapter, we will introduce a reoriented presentation of some design patterns of the GoF’s set.

4. A REORIENTED PRESENTATION OF THE DESIGN PATTERNS

The presentation of the GoF's design patterns set is becoming a standard in this field of research. However, as we claimed in the early chapter, in order to help consistently the designer in using design patterns it may be necessary to reorganize the design patterns in function of new criteria.

This chapter introduces a new presentation of design patterns issuing from a reorganization of the common ones. A first part presents some basements for our work, then a second part discusses practically the new necessary criteria highlighted in the third chapter. The third part explains how we must reorganize the other information to conserve a complete presentation of the patterns. Finally, we expose the general aspect of this new presentation.

4.1. Introduction

Before all, it may be considered that our work will not use the specification language LePus [Eden98b]. It is certain that it should give us some consistent basements, but LePus is too young and using it, we should stand surely in front of enormous studies[2] out of the scope of this thesis. By consequence, our work will not take advantages of this formalism.

Whatever, our purpose is to verify the validity of our previous analyze. We aimed to develop a prototype to examine if enhancing the learning of design patterns and navigating through the knowledge should be some reliable basements for a help system to assist the designer in using design patterns.

[2] We think particularly about the specification of the relationships which just begin to be studied by Vlissides and his colleges (see §2.3.2.3.2: Specification of the relationships).

To obey the restricted timetable of this thesis, we did not use the whole set of design patterns but a restriction of the set shared by the GoF's book. But, we try to make this restriction to be as much as possible representative of the entire set of design patterns.
On that purpose, we based our choice on two taxonomies of the design patterns, the classic one suggested by the GoF [GoFo95] and the classification of Kardell [Kard97]. These taxonomies define some strict categories in the design patterns' set. We selected a restricted set of design patterns in attempting to encounter the most various categories as possible.
We were also influenced by the course about the design patterns of Kerievsky from Industrial Logic [Keri00]. He define an order in which learning design patterns. This sequence is based

- on the importance of the design patterns: it begins by the more fundamental ones.
- on the relationships between them: the relationships offer logical links for restructuring coherently the course.

Based of those criteria, we groped for a permissible restriction and arrived on the following selection:

- *Decorator*
- *Strategy*
- *Template Method*
- *Composite*
- *Chain of responsibility*
- *Factory Method*

Finally, we notice the fact that this new presentation of design patterns will be based on an electronic support. As such, it will not be limited to a linear description like in the book tutorials. We will see in this chapter how we took benefits of this significant advantage.

4.2. Highlighting new features

In the third chapter, we drew out some design patterns' features necessary to highlight for enhancing their learning and facilitating their use. In this part, we will expose those features in each design pattern of our restricted set.

On this purpose, we will first study the recurring design principles below the design patterns, and then their consequences. Finally, we will characterize precisely the relationships between the design patterns.

4.2.1. Recurring principles below the design patterns

In this part, we will study the design principles which compose the design patterns. First we will analyze the style guidelines of the GoF's book which describe the main recurring design principles of design patterns. Then we will identify concretely the design principles used in our set of Design Patterns.

4.2.1.1. The style recommendations of the Gof catalog

In the section "How to solve design problems" of the GoF's book [GoFo95], they introduce two design rules:

- *Program to an interface, not an implementation.*
- *Favor object composition over class inheritance.*

Design patterns were introduced for giving design solutions to general problems of flexibility. The style guidelines of the GoF's book recommend flexible design methods over the common but more rigid ones.

For example, programming to an *interface* will give more flexibility to the code. Indeed, the clients manipulating *interfaces* are not identifying a specific object but a type of object. Similarly, using the *composition* for adding responsibilities to an object is more flexible than to use the *inheritance*. *Composition* can be dynamic when *inheritance* works at compile-time. However, the use of *composition* complicates the code in increasing the number of objects and their interactions. On the opposite, the *inheritance* is much easier to understand in a legacy system. The book discusses more in details the advantages and drawbacks of those guidelines.

Those style recommendations have for main goal to produce flexible code. However, the flexibility results usually in an increase of the complexity. Thus, those design rules are not imperative and like the design patterns, they cannot be applied in a cookbook fashion.

We still notice that the design patterns targeting flexibility, those guidelines appear generally as a leitmotiv in their design.

4.2.1.2. A set of Design Principles

On the base of the previous style recommendations and of our analyze of the design patterns' set, we identified some recurring design principles. We listed five general ones:

- **Composition**:
  Assembling or composing objects to get more complex behavior.

- **Inheritance**:
  A relationship that defines one entity in terms of another. Class inheritance defines a new class in terms of one or more parent classes. The new class inherits its interface and implementation from its parents.

- **Interface**:
  The set of all signatures defined by an object's operations. The interface describes the set of requests to which an object can respond.

- **Abstract class**:
  A class whose primary purpose is to define an interface. An abstract class defers some (in opposition to an Interface) or all (like an Interface) of its implementation to subclasses. An abstract class cannot be instantiated.

- **Direct implementation**:
  In opposition to Interface and Abstract class which defer the implementation to other classes.

Those design principles influence the characteristics of the design patterns they compose. For example, looking at *Composite* (Annex 1), we can identify that it uses in its solution the *composition*: the object Composite aggregates its children as objects conforming to the *interface* Component. Component is also the parent *abstract class* of the classes Composite and Leaf (object without child). Finally, *Composite* uses in its solution *composition*, *interface* and *abstract class* . *Composite* is a combination, a specific arrangement of those design principles. Consequently these last ones affect largely the results of *Composite* on a design. The dynamic aspect of *Composite* is due to the dynamic aspect of the *composition.* Similarly, thanks to the *interface* principle, *Composite* can encapsulate the children as black-boxes and so can treat similarly the Composite objects and the Leaf objects.

The design patterns results are definitely influenced by the design principles which compose them. In the design patterns of our restricted set, we identify the following combinations:

X uses Y in its solution	**Composition**	**Inheritance**	**Interface**	**Abstract Class**
Decorator	X		X	
Strategy	X		X	
Factory Method		X	X	
Template Method		X		X
Composite	X		X	X
Chain of Responsibility	X		X	

Table 1: Design principles in design patterns

4.2.2. Extracting the consequences of the design patterns

In this section, we specify the consequences of each design pattern of our set. For achieving this purpose, we first studied the different kinds of consequences belonging to the design patterns and their organization.

Consequently, in a first part, we will discuss the consequences of design principles and their influence on the design patterns. Then, we will explain our position on the use of the consequences and their organization. Next, we will highlight the relations between the consequences seen as advantages and the ones seen as drawbacks. Finally, we will refine each design pattern of the set to reveal clearly their consequences.

4.2.2.1. Consequences of design principles

As we discussed it in § 3.3.1.3, the design principles have discernible consequences. For example, we already saw that *composition* allows to add dynamically responsibilities to an object but that it also results in some complex interactions between the objects.

We explained in the previous part that the design patterns are largely influenced by the design principle. This influence can be interpreted as an inheritance of the consequences of design principles in the result of the design pattern. For example, the consequences of *composition* will be inherited by *Composite*: as *composition, Composite* will add dynamically responsibilities (in this case some component) to an object (in this case Composite).
The consequences of design principles are inherited by the higher-level structures: the design patterns, and so become also consequences of design patterns.

In this discussion about the consequences, both design patterns and design principles will be now named **design abstractions**. Moreover we will distinguish **consequences** and **inherited consequences**. A design pattern owns

- some **inherited consequences**, inherited from the design principles used in its solution,
- and some **consequences** which belong to the specific combinations.

4.2.2.2. About the use of the consequences

The consequences of a Design Pattern are its general effects on the code and design in which it will be used. The intent is the description of the particular design issue or problem that the design pattern addresses. As such, the intent is closely related to the consequences: the intent is achieved thanks to the consequences of the design pattern on the system in which it is applied.

Therefore there is some consequences of the design pattern which solve specifically the particular problem described in the intent. But the intent remains in a higher level than the consequences since it relies on them.

The consequences are the reflects of human experiences and as such are very important benefits of the design patterns. But like the intent, the consequences depend on a context. The most obvious evidence is the fact that we are able to distinguish advantages and drawbacks. For example, it is commonly accepted as positive that the *Composition* principle leads to more flexibility: it is easier to customized. However to increase the flexibility will let a more abstract implementation which will be more difficult to understand and learn. Then a same consequence can be seen as an advantage or a drawback depending on the point of view of the user.
In §2.3.2.2, we explained that the intent cannot be fixed and should stay open to variation for letting the possibility of innovative uses. Here we affirm that for the same reasons, the set of consequences of a design pattern cannot be exhaustive and could be extended.

Finally, the consequences are very useful tools to evaluate and understand a design pattern but, as the intent, they must not be fixed and the set of consequences of design patterns should stay open to extensions.

4.2.2.3. Lists vs. detailed descriptions

The consequences in the available tutorials, GoF or Java, are informal and detailed. They are spread all along the description of the Design Patterns, even if a special chapter is generally dedicated to them, like the chapter "Consequences" in the GoF's book.
In an article about a method for enhancing the creativity (TRIZ), Rantanen [Rant99] discusses about the notion of list. He claimed that *"lists are more psychological or pedagogical than rigid formulations "*. It is indeed much easier to look at a list than to a formulated explanations. Lists give a more synoptic view of the situation thus, they are easier to retain and synthesize.
From our experience, we observe it is common to abandon the study of the different consequences described as soon as we get a first interesting one: "Why to spend my time in going on searching further: I've already got what I need". Doing that, we miss some other advantages and specially some other drawbacks.

We believe that for a more reliable use of design patterns, their consequences should be presented in lists. However, it is noticeable that because in lists, the consequences' descriptions become smaller, they could be also less explicit. Consequently, it may be necessary to conserve the possibility to display more complete formulations of the formulation. Thanks to the electronic support, this option can be easily included in the lists as hyperlinks between the condensed descriptions and some more complete ones.

4.2.2.4. Related consequences

If we take for example the *Chain of responsibility* design pattern, we found from the tutorials description the following consequences:

- *give more than one object a chance to handle the request.*
- *make vary object that can fulfil a request*
- *avoid coupling the sender of a request to its receiver*
- *reduce coupling*
- *receipt is not guaranteed*

These consequences are just a listing of what is described more rigidly in the tutorials. We observe that all this consequences are related: they are all effect of a same mechanism seen under different points of view.

The consequences are the reflects of human experiences with the design patterns. The communication of this experience is significant for the designers. But, we need to give a minimum of directions to our work to avoid getting long lists of consequences full of recurrences.

It is interesting to expose several consequences as soon as they relate different point of view on the effect of design patterns. However, the lists do not have to contain useless redundancies.
Concretely, in the previous example, it may be noticed a repetition between: "*avoid coupling the sender of a request to its receiver*" and "*reduce coupling*". Those two consequences describe a same viewpoint: the coupling. The difference is that one is more general than the other. It seems more useful to keep the more specific since it is closer from the design abstraction itself.
On the opposite, the consequences "*give more than one object a chance to handle the request*" and "*make vary object that can fulfil a request*" may not be considered as redundant. There are very closed since they deal both with the handling of the request by different objects and so the flexibility. But, we still believe that the similarity is not enough obvious to make this redundancy useless. Indeed, the user should take benefits of the two explanations. Maybe to see "*make vary object that can fulfil a request"* will not be retain the attention of the designer who is planning to distribute the handling of messages when the other will certainly do it. Similarly, "*give more than one object a chance to handle the request*" could be too specific to the designer who looks for a variation possibility.

Finally, we retain that because they results from the same mechanism, all the consequences are often akin. But, the diversity may be substantial to conserve the real benefits of the consequences. Then we will not try to limit the set by some drastic criteria but just with the common sense.

4.2.2.5. Advantages and Drawbacks

A consequence can be seen as an advantage or a drawback depending on the situation. Because we want to help the user to evaluate the design abstractions, we must present him as much as possible the two sides of the consequences.

For example, here is the list of the consequences of the *composition* principle we were able to extract from the GoF book:

- *add features dynamically*
- *add flexibility*
- *lots of small objects are created*
- *reuse in terms of new components*
- *"black-box" reuse*

We analyzed each item of this list to define if it is an advantage or a drawback. We arrive to the following conclusions:

- *Add features dynamically:* This consequence is an explicit advantage as soon as we do not prefer to add the features statically.
- *Add flexibility:* It can be seen both as an advantage and a drawback. We can consider it as an advantage since it makes easier the customization, but also as a drawback since it makes the structure more complex and so harder to learn.
- *Lots of small objects are created:* Similarly, it may be considered as an advantage since it makes lights hierarchies but also as a drawback since the behavior depends on interrelationships.
- *Reuse in terms of new components:* We believe that this feature is not a consequence of the composition but more the description of the mechanism itself.
- *"Black box" reuse:* It can be seen as an advantage since it keeps encapsulation, but also as a drawback since it requires well-defined interfaces.

It appears that all consequences can be considered as an advantage and as a drawback. Therefore it is necessary to expose the two sides when they are not explicit. For example, for *flexibility*, *small objects* and *"black box" reuse*, we must present them both as advantage and as drawback. On the opposite, in some other consequences like *add features dynamically*, the advantage side as the drawback are explicit*:* the advantage is to be dynamic and the drawback is obviously to be not static. We will not express clearly the other side of such consequences since it results from the common sense.

4.2.2.6. Consequences in the Design Patterns' set

Finally, it may be necessary for each design abstraction to specify explicitly their consequences as lists. It should appear clearly in those consequences both the advantageous and disadvantageous sides.

In our set of Design Principles, the following results occur:

Design Principle	**Consequence**	**Advantage / Drawback**
Composition	add features dynamically	*Explicit advantage*
Composition	flexibility	easy to customized / hard to learn
Composition	Lots of small objects are created	light hierarchies / behavior depends on interrelationships
Composition	"black box" reuse	keeps encapsulation / requires well-defined interface
Interface	The client is not aware of the specific type of the objects	more transparency / objects are constrained to the interface
Interface	The client is not aware of the specific implementation of the objects	more transparency / objects are constrained to the interface
Inheritance	add features at compile-time	easy to customize and to modify the implementation / not dynamic
Inheritance	"white-box" reuse	keeps benefit of the features of the super class / inherits an implementation maybe not adapted to the new problem domain
Abstract class	provide only the generic implementation	direct the behavior of the subclasses / could be not adapt the new problem
Direct Implementation	easy to learn	*Explicit advantage*
Direct Implementation	Unflexibility	*Explicit drawback*

Table 2: Consequences of design principles

Similarly here are the consequences of the design patterns of our restricted set. We notice that the consequences inherited from the design principles are not included in this list since they were described in the previous table.

Design Pattern	**Consequence**	**Advantage / Drawback**
Composite	compose objects into tree structures to represent part-whole hiearchies	*Explicit Advantage*
Composite	lets client treat individual objects and compositions of objects uniformly	*Explicit Advantage*
Composite	make vary the structure and composition of object	*Explicit Advantage*
Composite	difficult to restrict the nature of the compos in function of the parent	*Explicit Drawback*
Composite	composed children do not know their parent	*Explicit Drawback*
Composite	children are not ordered	*Explicit Drawback*
Chain of Responsabilities	give more than one object a chance to har the request.	*Explicit Advantage*
Chain of Responsabilities	make vary object that can fulfil a request	*Explicit Advantage*
Chain of Responsabilities	avoid coupling the sender of a request to its receiver	*Explicit Advantage*
Chain of Responsabilities	Receipt is not guaranteed	*Explicit Drawback*

Design Pattern	Consequence	Advantage / Drawback
Decorator	change skin of an object and not the guts	*Explicit Advantage*
Decorator	add responsabilities	*Explicit Advantage*
Decorator	make vary object responsabilities	*Explicit Advantage*
Strategy	change guts of an object and not the skin	*Explicit Advantage*
Strategy	eliminate conditional statements	*Explicit Advantage*
Strategy	lets the algorithm vary independently from clients that use it	*Explicit Advantage*
Strategy	communication overhead if different granularity of algorithm (parameters)	*Explicit Drawback*
Factory Method	create objects knowing only when but not what	*Explicit Advantage*
Factory Method	lets a class defer instantiation to subclasses	*Explicit Advantage*
Template Method	defers steps of algorithms to subclasses	*Explicit Advantage*
Template Method	requires to know which method to override and which not	*Explicit Drawback*

Table 3: Consequences of design patterns

4.2.3. Characterizing the relationships between Design Patterns

In § 3.3.2, we explained that the navigation through the design patterns set should help the designer to find solutions in the set. On that purpose, the relationships between the design patterns appear as efficient gates towards the solution provided that their nature is explicitly exposed.

This part characterizes the most clearly as possible the relationships between design patterns. First, we will review the available works of this field. Then we will explain that the relationships could also be extended to the design principles and not only limited to the design patterns. Finally, we will introduce a new taxonomy of the relationships between design patterns.

4.2.3.1. Existing works

As we discuss it in § 2.3.2.2, the relationships between the design patterns remain a fuzzy domain. The research in this activity have just begun to have consistent basements thanks to the recent specification language LePus [Eden98b]. But, this activity is still too undeveloped to enter the scope of this thesis. Therefore, we will run on the informal works done in this domain. We will, however, go ahead in clearing up as far as possible these relationships.

It is interesting to notice on this domain, the taxonomy introduced by Zimmer [Zimm96]. He highlighted three types of relationships between the design patterns:

- X uses Y in its solution
- X is similar to Y
- X can be combined with Y.

We notice also that the GoF's book introduces the notion of alternatives. This other relationship may be significant for navigating through the design patterns in the research for solutions. For example, in the "Related Patterns" part of the *Decorator* description, we find *Strategy* as an alternative design pattern.
This alternative is described relatively to a specific consequence of the design pattern. In our example, this consequence is that *Decorator* ***changes the skin of an object and not the guts*** when *Strategy* ***changes the guts of an object and not the skin***.

4.2.3.2. Relationships between design principles

We believe also that relationships could be extended to the design principles. For example, the alternative relationships is obvious between *composition* and *inheritance*: the two of them are dealing with specialization and reuse.
Moreover, by definition of the design principles, it may be considered relations like "*Composite* uses *composition* in its solution" as explained in § 4.2.1. As a matter of fact, the relationships "*X uses Y*" can be applied between design patterns and design principles.

Therefore the relationships could be define for the design abstractions in general, that is to say indistinctly design patterns and design principles.

4.2.3.3. A new taxonomy

Finally our analyze leads us to extend the Zimmer's taxonomy

- to the relationships between design abstractions and not only design patterns,
- in including the notion of consequences.

We obtained the following statements:

- ***X uses Y*** *means that X will share the consequences of Y.*

For example, in the case of " *Composite* uses *composition* in its solution" , *Composite* will inherit the consequences of *composition*. The "use of" relationships between the design patterns of our restricted set and the design principles we identified were already presented in table 1 (§4.2.1.2).

- ***Y is an alternative to X*** *means that Y owns a consequence which is an alternative to a particular consequence of X.*

We added this relationship since it is very useful to the navigation: when a consequence is not satisfying, the user have the possibility to study another alternative design pattern and to compare the two.

- ***X is similar to Y*** *means that X is very closed to Y but that it defers sufficiently not to be the same one.*

We believe that this category is very closed to the next one. A similar design pattern can also be considered as an alternative, but a specific kind of alternative. This relationship is also very important for the navigation. As we claim it in § 3.2.2.3: " Highlighting the relationships between design patterns. a similarity between two design patterns is an opportunity to refine the comprehension of each ones.

- ***X can be associated with Y*** *means that it is common to use the feature of one to improve the other*

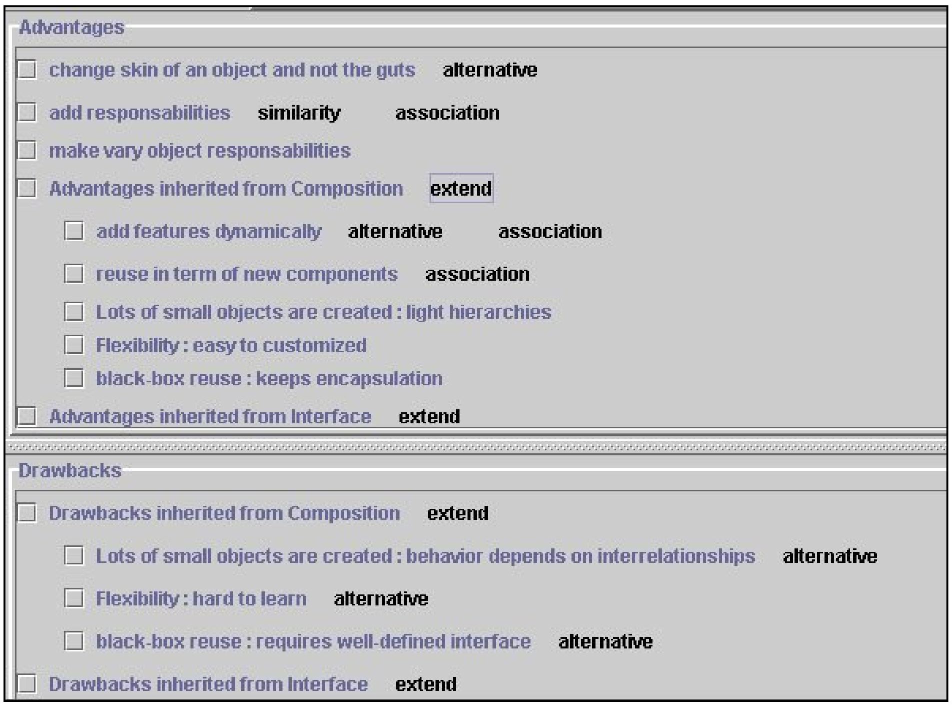
This relationship describes some common association between design patterns.

We applied these new taxonomy on the set of design patterns and the set of design principles. The detailed results are available in Annex 2 and Annex 3.
It may be noticeable that the relationships are associated to some consequences. For example, *Decorator* changes the skin of an object when *Strategy* changes the guts. Then this *alternative* relationships will be associated to the consequence of *Decorator* "changes skin of an object and not the guts". In the same way, the *similarity* between *Decorator* and *Composite* could be associated with the consequence "add responsibilities" since the two of them allow to add responsibilities but for distinct purposes.

4.2.4. Conclusion

Our presentation should separate the advantages from the drawbacks and organize those consequences in list. Moreover, the relationships between design patterns can be associated with one of their consequences and so displayed in the list.

Consequently we arrive to the following presentation (here with *Decorator*):



Advantages

- ☐ change skin of an object and not the guts alternative
- ☐ add responsabilities similarity association
- ☐ make vary object responsabilities
- ☐ Advantages inherited from Composition extend
  - ☐ add features dynamically alternative association
  - ☐ reuse in term of new components association
  - ☐ Lots of small objects are created : light hierarchies
  - ☐ Flexibility : easy to customized
  - ☐ black-box reuse : keeps encapsulation
- ☐ Advantages inherited from Interface extend

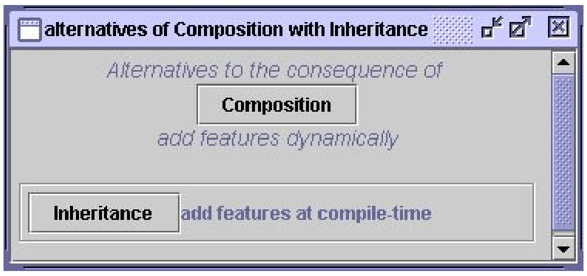
Drawbacks

- ☐ Drawbacks inherited from Composition extend
  - ☐ Lots of small objects are created : behavior depends on interrelationships alternative
  - ☐ Flexibility : hard to learn alternative
  - ☐ black-box reuse : requires well-defined interface alternative
- ☐ Drawbacks inherited from Interface extend

Figure 4: List of consequences

The consequences of the design principles which compose the design pattern are displayed as *Advantage inherited from X* or *Drawbacks inherited from Y* which can be optionally extending.

In taking advantage of the electronic support, this presentation offers a synthetic description of the advantages and drawbacks of a design pattern. In addition to that, it highlights explicitly the difference between the *inherited consequences* and the *consequences* of design patterns. The relationships between the design abstractions are presented as hyperlinks related to a consequence. Those hyperlinks lead to brief descriptions of the related relationships. For example, for the alternatives related to the “add feature dynamically” consequence of *composition*, we get:



alternatives of Composition with Inheritance

Alternatives to the consequence of

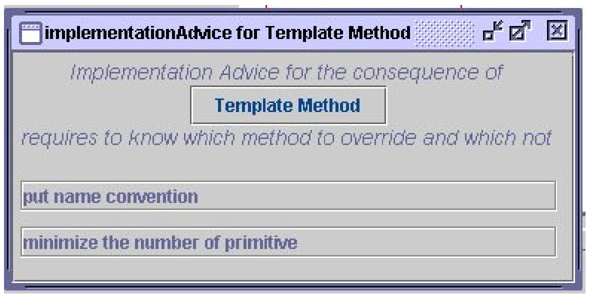
Composition

add features dynamically

Inheritance add features at compile-time

Figure 5: Relationship description window

This presentation exposes also the implementation advice related to some consequences. Like the relationships, they are accessible with an hyperlink which leads to their description. For example for *Template Method*:


implementationAdvice for Template Method

Implementation Advice for the consequence of

Template Method

requires to know which method to override and which not

put name convention

minimize the number of primitive

Figure 6: Implementation advice description window

4.3. Towards a complete description

In regard to the conclusions of §3, the previous part introduces a new presentation of design patterns' consequences.

Towards a complete description of design patterns, this part exposes the way by which our presentation includes some other essential items.

First, we present a new structure for the intent. Then we discuss the insertion of the diagram, the participants description and the examples in our design patterns' presentation.

4.3.1. About the intent of the design pattern

4.3.1.1. Introduction

In §2.4, "Position on the use of Design patterns", we concluded that the design patterns are the reflects of human experiences: they are communication tools. As such, their *intent* must be consider as a significant part. But, the design patterns are first communication tools of a design strategy, of an abstract mechanism. So to take entirely benefits of them, the intent must not be fixed and has to stay open to variations.

The GoF's book define the intent section of a design pattern as:

A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

The intent as it is defined in the GoF's bo ok is essential in the description of design patterns as communication tools of human experiences. However, design patterns have to be first considered for the design mechanism they encapsulate. Consequently, the intent cannot be the center of the description of the design patterns.

As for the consequences, the intent is described in a rigid formulation in the GoF's book. We think necessary to clarify this description and to present it in a more structured way.

First, we study how some existing works could help us in structuring the intent. Then we propose a new consistent format for its description.

4.3.1.2. Existing works

4.3.1.2.1. A more formal structure of the intent

Gustavsson and Ersson [GuEr99] have worked on the formalization of the intent of design patterns. From their work we retain some formal structures they introduced for describing the intent of the design patterns.

For *creational patterns*, they highlighted the following sections:

Create	Entity	Action	Force	Supplementary information

For *structural patterns:*

Action to apply	Entity	Outcome	Result

For *behavioral patterns*:

Objective	Course of action or proceedings	Result

This study presents the interest to give a first approach of how to structure the intent of a design pattern. However we believe it is insufficient to clarify really the intent. Specially, it should be necessary to get a sole structure, even more general, for all the design patterns. Indeed, since the designer needs to be accustomed with the structure of the intent for taking really benefits of it, this structure must be consistent from one design pattern to another.

4.3.1.2.2. A taxonomy of the Design Patterns

To helping the designers to he use of design patterns, Magnus Kardell introduced a new taxonomy [Kard97]. His work is interesting because it should give some direction to classify and clarify the intent of the design patterns.

He arrived to four general fields of classification in which he analyzed some subcategories:

-	***applies to:***	object / object families / related object families
-	***purpose:***	instantiation /functionality / interface / communication / physicality / access / state
-	***scope:***	static / dynamic
-	***application time:***	building /reusing

We believe in the usefulness of such a classification compared with the more classic but less complete one of the GoF's book: creational / behavioral / structural.

4.3.1.2.3. The variation purpose of the Design Patterns

"Each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change." [GoFo95]

The GoF's book highlights concretely this common feature with a list of associations between design patterns and their variation purposes. We believe that the variation purpose of a design pattern is an important part of its intent and that it should clearly highlighted in our description.

4.3.1.3. Reformulating the intent

We wanted to get a structure of the design pattern which could be related closely to the taxonomy of Kardell. Influenced by the formalization of Gustavsson and Ersson, we arrive to the following structure:

- applied to (entity nature)
- purpose
- mean (collaboration)
- result
- scope
- variation

For example, here is the intent of *Decorator* as described in the GoF's book:

"Attach additional responsibilities to an object d ynamically. Decorators provide a flexible alternative to subclassing for extending functionality."

With our structure, we obtain for *Decorator*:

Applied to		
Category	**Description**	
Object	an object	
Purpose		
Category	**Description**	
Functionnality	add responsabilities	
Scope	Dynamic	
Mean (collaboration)	by forwarding requests to the composant	
Result	delivers a flexible extending mechanism	
Variation	make vary object responsabilities	

Table 4: Intent structure

This structure of the intent includes all the information of the original intent excepted the idea of *flexible alternative to subclassing*. However, this last point is an *alternative* relationship and is not really essential in the intent description.

This presentation adds the *Mean* by which the design pattern attempt to its purpose. The *Mean* deals more with the description of the abstract mechanism itself than the intent. However it is sometimes present in the intent description of the GoF's book. Indeed, it gives some useful information for understanding the design patterns' use. Consequently, it may be interesting to include it in our intent structure.

We notice also in our structure the *Variation* item. Its purpose is to offer another viewpoint on the design patterns' intent: the one of the variation. We believe that to see the intent from two viewpoints increases the understanding and goes ahead with the idea that the intent should not be fixed.

The Annex 4 presents the description of all design patterns of our set accorded with our new format.

4.3.2. The structure

4.3.2.1. The diagram

In the available tutorials, the structure of the abstract mechanism encapsulated in the design patterns is presented with a diagram and explained with an informal description of collaborations.

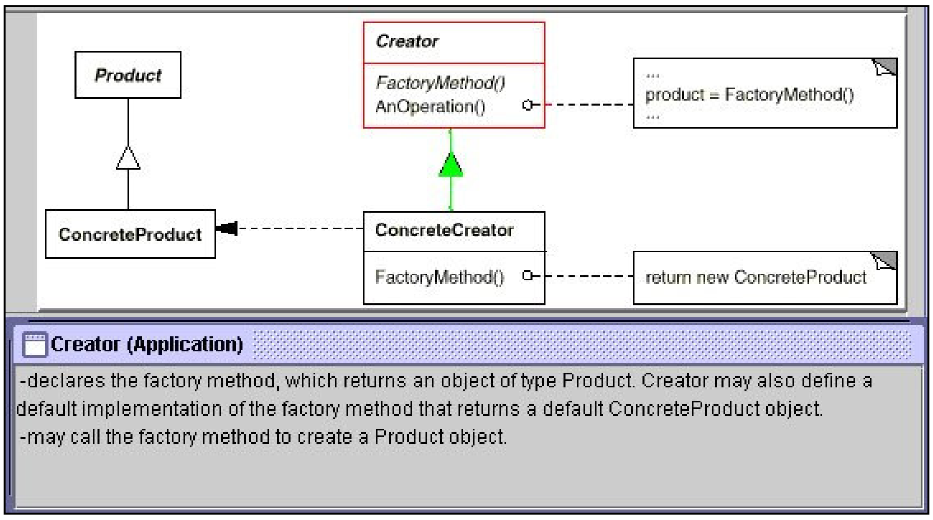
The diagram is done with the OMT/UML notations which is a widespread vocabulary. A detailed description of the role of the participants and their collaborations is necessary to complete the information offered by the diagram. However, the graphical explanation has the advantage to be expressive and synthetic and remains the center of the description of the structure. Similarly, in our presentation, it may be essential to represent the structure with such a diagram.

4.3.2.2. The description of the participants

It may be necessary to provide a weightless presentation of design patterns, more attractive and easier to understand in a glance. However, for remaining consistent and not disregarding some information, we must take benefit of the electronic support.

The diagram synthesizes the structure of the design strategies encapsulated in design patterns. The descriptions of the participants of the structure are additional explanations of the diagram. As such, they are directly related to the diagrams and are meaningless when standing alone. Indeed, it is the study of the structure of the diagram which brings to require more details about the roles of the participants.

Consequently, it may be interesting to use the diagram itself as a portal towards the participants' descriptions. Practically, we use hyperlinks towards the descriptions on the diagram. This method is largely used on the internet and we think that the people are generally comfortable with it. Thus, a mouse click on one of the object pictured in the diagram will display its description. For example, here in *Factory Method*, to click on the actor *Creator* results in the display of a description window.



Creator (Application)

-declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
-may call the factory method to create a Product object.

Figure 7: Participants description window

4.3.2.3. The design principles

We claimed, in § 3, the necessity to highlight the design principles composing design patterns for enhancing their understandability. We highlighted this composition in the description of the consequences but we believe that it should also appear clearly in the structure itself. The diagram should show the design principles.
Practically we used colors to highlight the design principles in the structure diagram of a design pattern. Here in *Strategy*, we obtain:

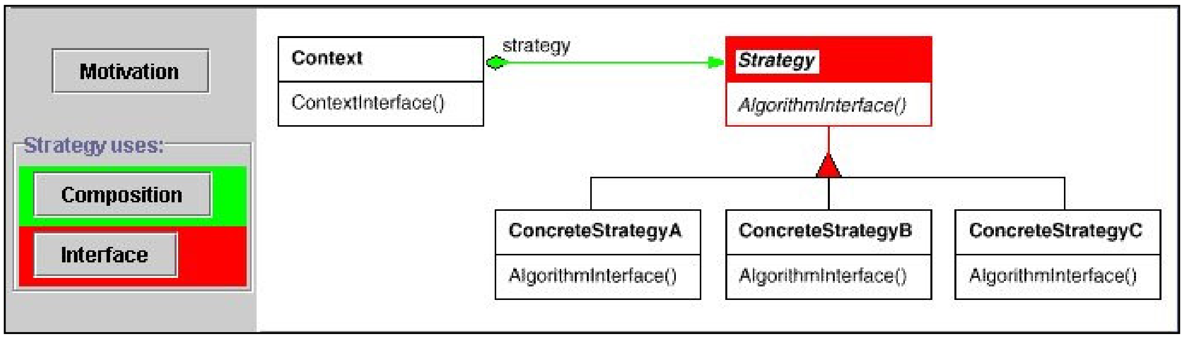
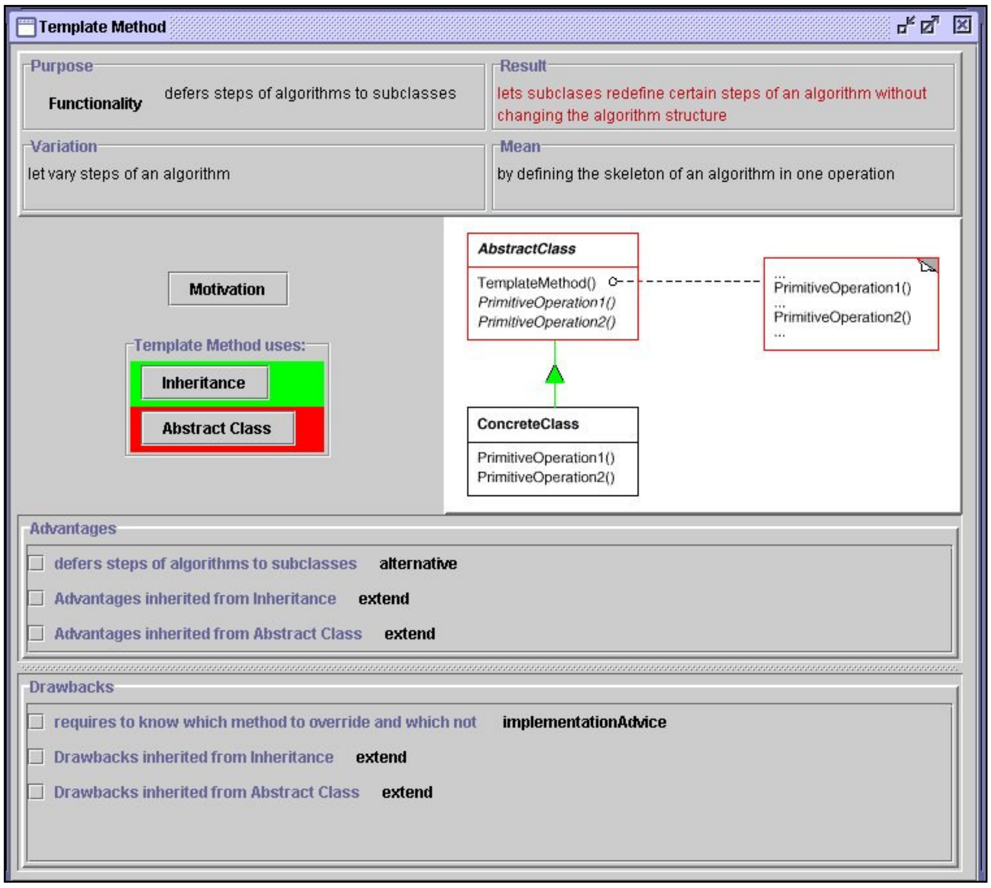


Figure 8: Highlighting of design principles in the diagram

4.4. General appearance of the new presentation

From our experience, the graphical representation of a design pattern is generally the first think that a designer will look at. The designers work very often with such diagrams. Therefore they are generally more comfortable with diagrams that with more weighing textual presentations. Of course, the diagram might not be sufficient to understand in details a design pattern. But it should stand as the center of the tool.

A design pattern will therefore have the following general appearance, here for *Template Method*:



Purpose
Functionality defers steps of algorithms to subclasses

Result
lets subclases redefine certain steps of an algorithm without changing the algorithm structure

Variation
let vary steps of an algorithm

Mean
by defining the skeleton of an algorithm in one operation

Advantages
- ☐ defers steps of algorithms to subclasses alternative
- ☐ Advantages inherited from Inheritance extend
- ☐ Advantages inherited from Abstract Class extend

Drawbacks
- ☐ requires to know which method to override and which not implementationAdvice
- ☐ Drawbacks inherited from Inheritance extend
- ☐ Drawbacks inherited from Abstract Class extend

Figure 9: Design pattern description window

We have already discussed the benefits of an example for understanding an abstraction and it may be significant to provide such a textual support in our presentation. Therefore, we provide an hyperlink towards an example with the button *motivation*.

The comprehensibility of our presentation is based on the separation of the different parts of the design patterns description. In dividing the information, the different features of design patterns become more easily accessible. Then the presentation can offer a more synoptic view of the whole.

The common descriptions take care of describing the design patterns in a logical and structured way detailing the connections between each items. Our presentation takes benefits of the electronic support and skips those heavy connections existing between the different parts of the description.

It may be considered that our new presentation could be beneficial for the users. It displays clearly the substance of design patterns in both a consistent form and a restricted field. However, we know that a textual support could be reassuring for the users: in such descriptions, the connections between the different parts are explicitly done and the users have just to let themselves be carried by the speech. Our presentation furnishes an access to such a textual description with the *Motivation* button.

Some shot-screens of this new presentation are available for all the design patterns and design principles of our set in Annex 5.

4.5. Conclusion

We have introduced in this chapter a new presentation of design patterns.

Since the research field of the design patterns' specification is just beginning to present interesting results, our presentation could not have been based on a strict formalism. However, we proceeded available works about the design patterns description and their relationships towards a new coherent format of design patterns.

This new presentation attempts to reorient the design patterns' description in the direction of essential but too disregarded features: the design principles which compose design patterns, the consequences of design patterns and their relationships. The result is the introduction of an explicit list of consequences for each design pattern. This list make a clear separation between the advantages and the drawbacks of the design pattern. It discerns explecitly the consequences of the design principles which compose it, from its own consequences. Finally, it presents, associated to the consequences, some relationships with other design patterns.

For being consistent, this presentation includes also the description of the intent and the structure. The intent is reformulated according to a complete and coherent pattern. The structure is displayed as a diagram which becomes the center of the presentation. All this new presentation is the portal towards complementary information like the description of the participants in the structure or a textual description of an example.

In taking benefits of the electronic support, the new presentation skips some nonessential parts of the description: the informal transitions surrounding each items. Consequently, it furnishes a compact outlook of the design pattern.

Finally, we succeeded in producing a compact but coherent and comprehensible original display of design patterns' descriptions. Moreover it puts forward the features identified, in the third chapter, as necessary for enhancing the design patterns' use. So, it may be considered as a consistent base for a help system.

In regard to our new presentation, the next chapter proposes a help system for assisting the designer in using design patterns.

5. A HELP SYSTEM

5.1. Introduction

In the second and third chapters, we highlighted some characteristics of design patterns which could help the designer to use them. We arrived to the necessity to enhance the comprehension and the learning of the design patterns as well as the navigation through the set of design patterns.

After this analyze, we worked on how to reformat the description of design patterns according to these new considerations. We obtained an electronic presentation of design patterns.

Our purpose now is to introduce a system to help practically the designer to find a solution to its problem in the set of design patterns. This system will support the navigation through the knowledge in taking benefits of the new presentation we defined.

The first part, presents an overview of our help system. Then a second and third detail how this help system integrates the two principles we introduced previously: the evaluation process and the navigation. Afterwards a fourth part discusses the advantages and drawbacks of our open assistant. Finally, a last part presents some conceptual features which enhance the efficiency of our help system.

5.2. Overview of the help system

Our help system attends to assist the designer for using design patterns. As an assistant, it should be the most transparent as possible and the design patterns should remain the center of the system. Typically, the system must be a support for displaying judiciously design patterns.

On this purpose, we constructed a system with two windows. The main one displays the design patterns, the design principles, the entry points, the relationships descriptions, i.e. all the knowledge of the system. It is in this window that the user will evaluate the design abstractions.
The second window will display the state of the navigation. As such, it can let continuously the designer know about the results of his evaluations and the trace of the knowledge he already explored.

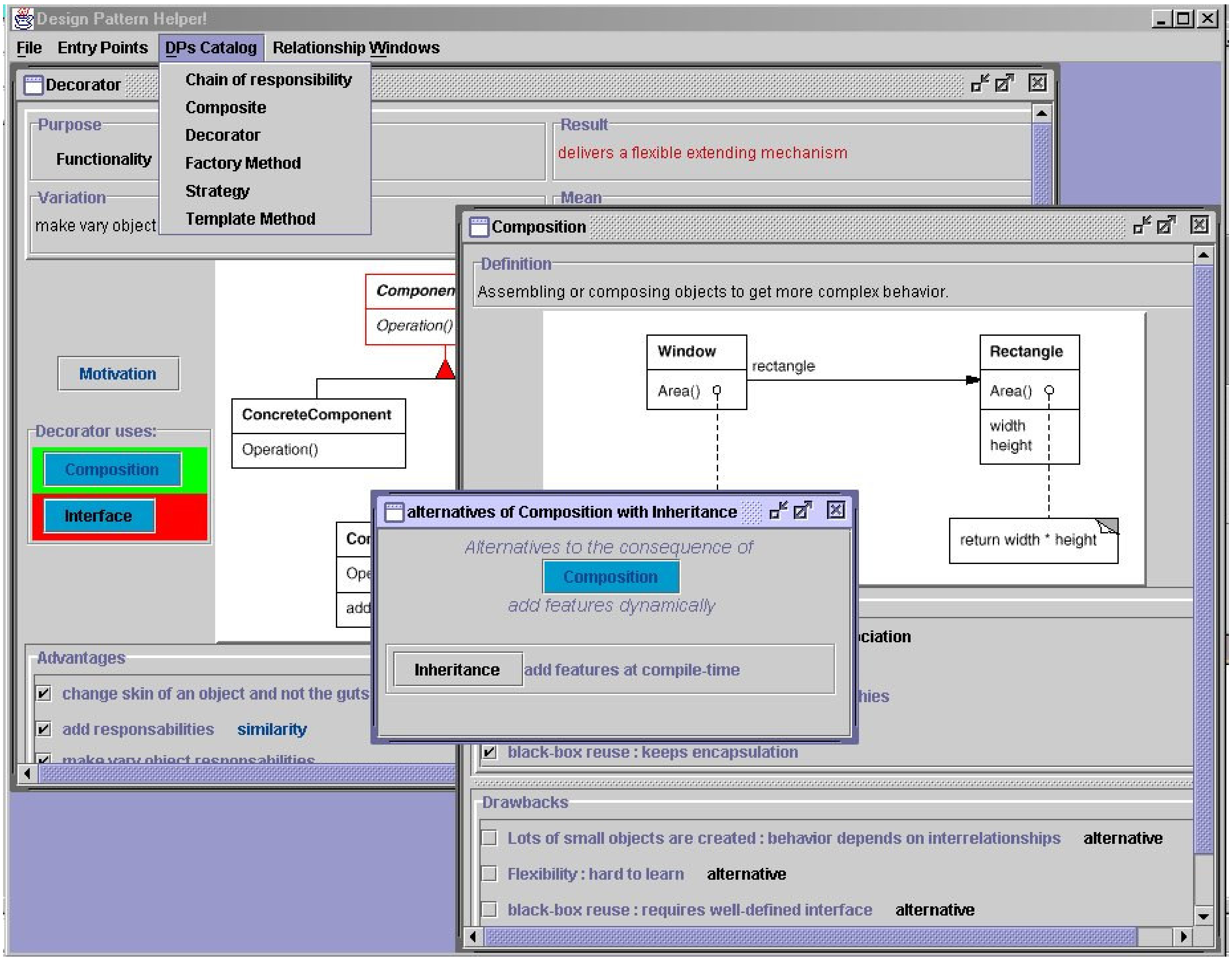


Figure 10: Help system's main window

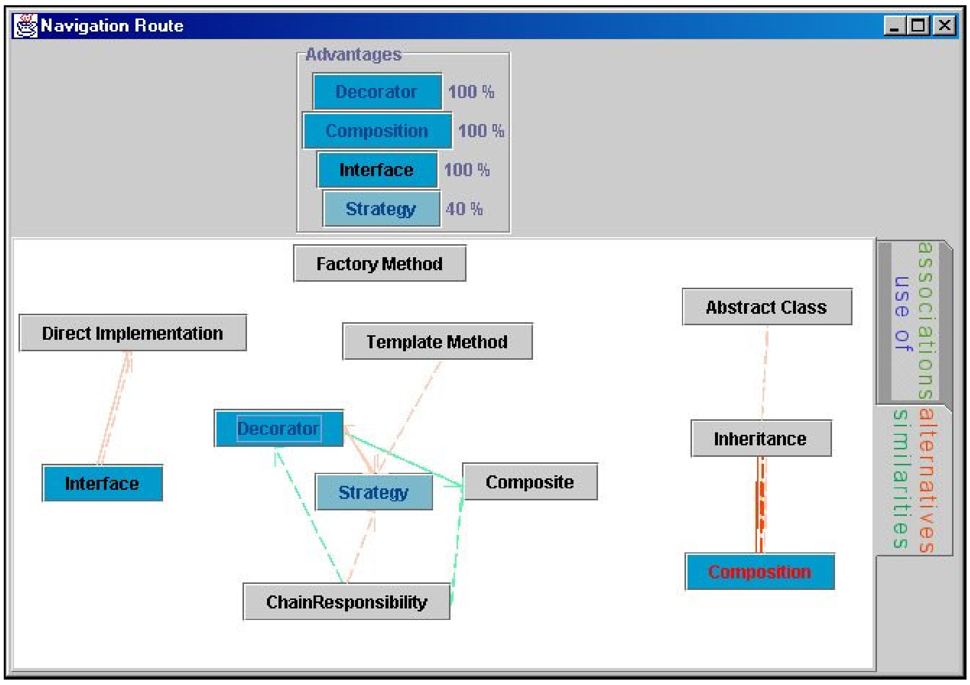


Figure 11: Help system's navigation board window

5.3. Design Patterns' evaluation

In § 3.3.1, we concluded that the design patterns' evaluation must be based on the check of their consequences:

- advantages needed / not needed
- drawbacks not accepted / accepted.

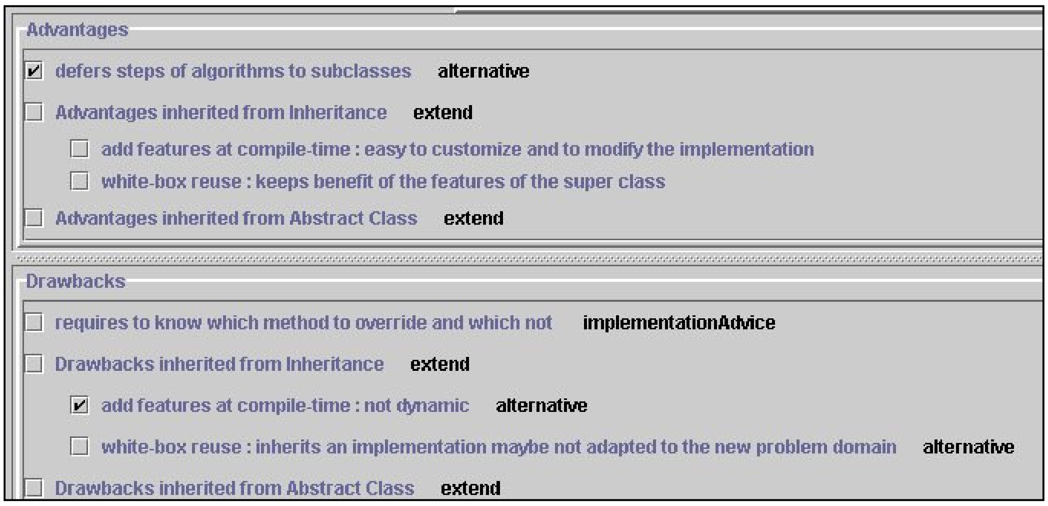
This part discusses the integration of this principle in the help system. First, it explains how to consider the consequences as check lists. Then, it presents how the help system traces those evaluations.

5.3.1. Consequences as Check Lists

The consequences are a mean of evaluating design patterns. Our help system allows the users to evaluate each consequence in presenting the consequences' sets as check lists.

We distinguished the consequences considered as advantages from the ones considered as drawbacks. The evaluation is based on the following rules:

- checking an advantage means to need effectively this feature for solving the problem,
- checking a drawback means to reject this inconvenient in the particular context of the problem.



Advantages

☑ defers steps of algorithms to subclasses alternative

☐ Advantages inherited from Inheritance extend

  ☐ add features at compile-time : easy to customize and to modify the implementation

  ☐ white-box reuse : keeps benefit of the features of the super class

☐ Advantages inherited from Abstract Class extend

Drawbacks

☐ requires to know which method to override and which not implementationAdvice

☐ Drawbacks inherited from Inheritance extend

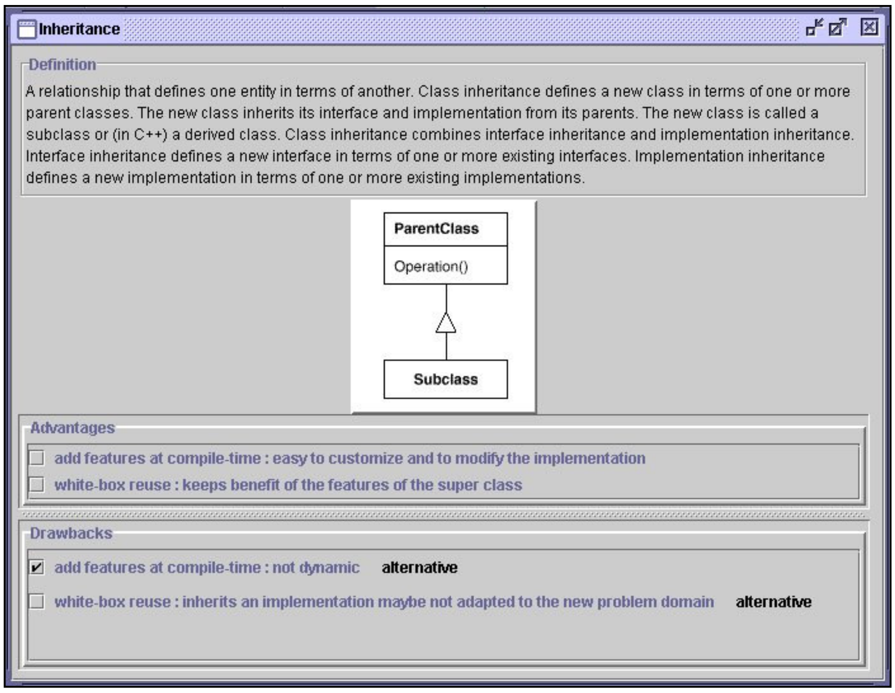
  ☑ add features at compile-time : not dynamic alternative

  ☐ white-box reuse : inherits an implementation maybe not adapted to the new problem domain alternative

☐ Drawbacks inherited from Abstract Class extend

Figure 12: Checking of consequences

For example, the previous figure is showing the advantages and drawbacks of *Template Method* . The designer selected the advantage named *defer steps of algorithm to subclass*. Then, he expressed his needs for making vary some parts of an algorithm. He checked also the drawback named *add feature at compile time: not dynamic.* Doing that, he expressed the fact that he is not ready to accept a solution which is *not dynamic*. This drawback belongs to the *inheritance* principle as shown on the list. Actually, to reject this drawback means to refuse the use of *inheritance* which is static. The help system is consistent and then each time that the user will encounter the consequences of *inheritance* (in another design pattern or in the description of this design principle itself), he will retrieve the rejected drawback.


Inheritance

Definition

A relationship that defines one entity in terms of another. Class inheritance defines a new class in terms of one or more parent classes. The new class inherits its interface and implementation from its parents. The new class is called a subclass or (in C++) a derived class. Class inheritance combines interface inheritance and implementation inheritance. Interface inheritance defines a new interface in terms of one or more existing interfaces. Implementation inheritance defines a new implementation in terms of one or more existing implementations.

Advantages

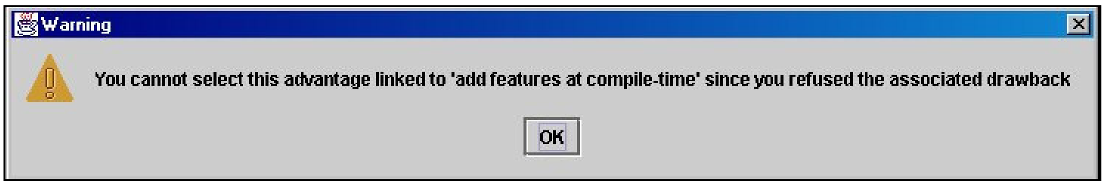
- [ ] add features at compile-time : easy to customize and to modify the implementation
- [ ] white-box reuse : keeps benefit of the features of the super class

Drawbacks

- [✓] add features at compile-time : not dynamic alternative
- [ ] white-box reuse : inherits an implementation maybe not adapted to the new problem domain alternative

Figure 13: Consistence of the checking of consequences

Not dynamic is the drawback side of the consequence *add features at compile time*. This consequence of *inheritance* can also be seen as an advantage since it involves that in the system it will be *easy to customize and to modify the implementation*. It is important to notice that the user cannot take benefit of a consequence as an advantage if he cannot accept it as a drawback. To avoid it, if he attends to check both the advantage and drawback sides of a same consequence, the system will warn him.


Warning

You cannot select this advantage linked to 'add features at compile-time' since you refused the associated drawback

Figure 14: Warning window of contradictory checking

Finally, the check of the design patterns' consequences enables their evaluation. But the help system must be consistent. Then if, for example, one advantage of *composition* is checked, this advantage must be selected in all the design patterns in which this design principle participates.
Moreover, it is necessary to prevent the user from making some contradictory checks. Thus, the system refuses to check in the same time a drawback and an advantage related to the same consequence.

5.3.2. Tracing the evaluation

It may be considered to trace the results of the evaluations that the user did during his navigation. On that purpose, we use a visual indicator: some colors. For example, if the user is checking some advantages of *Decorator,* the background color of all the hyperlinks towards this design abstraction will turn in blue. Practically, all the buttons *Decorator* in the map, in the relationships' descriptions, in the design abstractions' descriptions and in the entry points to the navigation will be blue. This color will have a saturation depending on the rate of advantages checked: it will be more saturated if more advantages are selected. Similarly, to refuse some drawbacks will color the background of the related design abstractions in yellow.

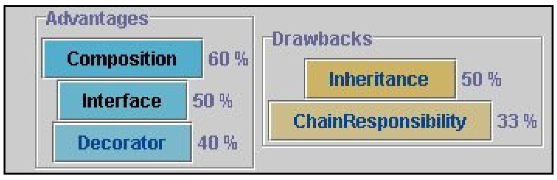


Figure 15: Board of evaluations' results

In addition to those indicators, it may be interesting to display a summary of the evaluations' results which could serve as a reference board for the user. We included this board in the navigation window. This table orders the results in two columns: one for the advantages and one for the drawbacks. For each design pattern displayed, it indicates the rate of advantages or drawbacks checked.

5.4. The Navigation

This part presents the principal feature of our system to help the user to discover a solution: the navigation through the knowledge.

A first part introduces the devices of our help system which allow to begin the navigation. Then a second part explains the significant advantages of supporting the navigation with a map.

5.4.1. Beginning the navigation

Our system attends to help the designer to use the design patterns. We can imagine two main use cases of the help system:

- *the user wants to apply a design patterns and needs to verify his choice* . It is the most common use of the design patterns. The designer is in a situation that reminds him a design pattern he has already used or heard about. He will look at the description of this design pattern for verifying if it is really proper and for studying the abstraction mechanism to be able to apply it.

- *The user has a problem and imagine that a design pattern could help him .* For example, the designer is in front of a design problem which involves the flexibility of the code or a separation of concern. Since he learnt from his experience that design patterns could solve these kinds of situations, he will try to find a solution in the set of design patterns.

In the first case, the user already knows the design pattern and just requires for an easy and fast access to this pattern. On that sense, the help system holds a menu containing the design patterns' list.
In the second case, the situation is more problematic: the user does not yet know which design pattern to use for solving his problem.

5.4.1.1. Automatic suggestion of a design pattern

For leading the designer towards a solution when he only knows the fact that he has a problem, we should, at first glance,

- capture and analyze his problem
- try to find an appropriate design pattern.

Unfortunately that puts us in front of two hard obstacles:

- the main difficulties we have to capture the problem of the user,
- the necessity to let open the design patterns' use.

5.4.1.1.1. Main difficulties to capture the specific user's problem

First we need to know how to capture the specific problem or the attention of the designer. We saw in § 2.3.3.3 that the research into the design patterns suggestion from a problem recognition in legacy systems gives really unsatisfying results.

A recent work [Corr99] is attempting to recognize common design defaults in legacy systems. But the design patterns cannot be used in a cookbook fashion and this work is not sufficient to suggest efficiently a solution in terms of design patterns.
Moreover, the use of design patterns does not always make good design. The design patterns can also lead to useless complexions of systems. And the design patterns' application must take into account some requirements which are not included in the design itself like " *How the design should be reuse?*' or " *Which are the entities which should remains flexible?*'.

We claim that a defaults' detection in legacy systems like Correa did [Corr99], could be much more efficient if it was not limited to the analyze of the design or code but if it was also asking precise questions to the user. On that purpose, this system should certainly use some artificial intelligence techniques.

Such a system is applicable only at refactoring time. Thus, it should be also interesting to find a way to help the user to communicate his specific problem even without concrete support like a code or a design. As for the refactoring, the system should ask precise questions to the designer to try to help him to define and model his problem.

Those systems are particularly complicate to implement because as we explained in § 2.2.2.2, the design patterns can be applied at different levels of granularity. Then the system should be able to detect a situation involving the use of design patterns without any indications of the level of granularity at which the problem resides.
In the case of the refactoring, a possible way may be to analyze systematically all the possibilities until finding situations susceptible to be a problem. But this calculation should run very high techniques of artificial intelligence to keep a reasonable use of resources.
In the other case, the solution is more complicate. First the system must know if the user is describing the problem at the right level of granularity. Indeed, maybe the problem is hanging out at a higher level that the one visualized by the designer. To solve such a situation, the system should be sufficiently intelligent to deduce from its interactions with the user if the problem resides at another granularity or not.

Those two directions involve the state-of-art of artificial intelligence. As such, they constitute interesting research directions but could not be defended in the limited scope of this thesis. Moreover, as we will explain in the next part, these directions should not be sufficient to help really the designer in using design patterns.

5.4.1.1.2. Necessity to let open the use of the design patterns.

In the previous paragraph, we proposed some directions to automate the suggestion of design patterns. The principle is to detect a problem that is solvable with a design pattern. Such problems enter, by definition, the scope of the *intent* of design patterns: the intent is the **only** concrete relation between the abstraction mechanism encapsulated in the design patterns and a context of application. It is why the intent is one of the main benefits of the design patterns.

However as we claimed it in § 2.4, design patterns are also abstract mechanisms and should not be limited to the use described by the intent. A design pattern should be able to solve a problem which is not identified by its intent. Moreover, as Chris Pehura [Pehu00] said "*the best design should be the realization of the combination of at least three design patterns*". But the use of a combination of design patterns is already out of the scope of the intents of all those design patterns.

Therefore, to conserve all their benefits, the design patterns cannot be limited to the use described in their intent. But since the intent is the only mean to associate design patterns to some practical problems, an automatic suggestion will definitely restraint the interest of the design patterns. Thus, a consistent help system to assist the designer in using design patterns cannot be restricted to an automatic suggestion tool.

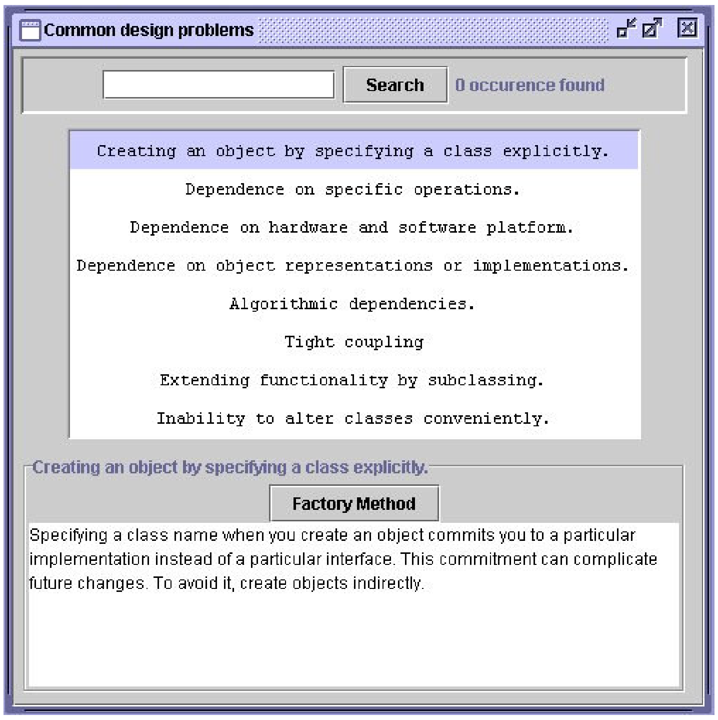
5.4.1.2. Other entry points

Our previous conclusions lead us to the necessity to find other entry points in the navigation. These entry points are not conducting obligatory to an appropriate solution but they will certainly suggest a first design pattern, from which the designer could begin his navigation towards a solution.

5.4.1.2.1. Design defaults

The GoF's book furnished a list of generally encountered design defaults. A design default should be for example " *dependence on specific operations*" or " *creating an object by specifying a class explicitly*". The list suggests for each design default a set of design patterns susceptible to solve the problem. It may be considered that the designer could find, in this list, some situations related to his problem and so could begin the navigation.
Moreover, by definition, this list gives some very useful style recommendations. It provides also an interesting overview of the design patterns' scope of action since it described some concrete problems they solve. Thanks to that, the designer should be more aware of why and when to use the design patterns.
In checking the list, the designer will learn that some design habits could be considered as defaults like " *creating an object by specifying a class explicitly*". Even if this point does not correspond exactly to his problem, the next time he will need to create a class in his design, he will certainly think about that and look for using a design pattern.

Finally, this list is very advantageous in the context of an help system: it could serve at an efficient entry point in the navigation and furnishes some essential style advice for OO designers. Therefore, we will integrate this list in our help system. A system of research by keywords was implemented to facilitate the use of the list.



Creating an object by specifying a class explicitly.
Dependence on specific operations.
Dependence on hardware and software platform.
Dependence on object representations or implementations.
Algorithmic dependencies.
Tight coupling
Extending functionality by subclassing.
Inability to alter classes conveniently.

Creating an object by specifying a class explicitly.

Specifying a class name when you create an object commits you to a particular implementation instead of a particular interface. This commitment can complicate future changes. To avoid it, create objects indirectly.

Figure 16: Common design problems description window

5.4.1.2.2. Design Patterns' taxonomy

Some taxonomies were defined for helping the designer to use the design patterns. They classify the patterns in explicit categories which reflect their application scope. On this subject, it is interesting to notice two major works.

The first taxonomy was introduced in the same time that the design patterns by the GoF [GoFo95]. It remains at present the most known classification of the design patterns. They defined two fields divided in few categories:

- ***Purpose:*** structural / behavioral / creational

- ***Scope***: class / object

With this classification, they could highlight some families of design patterns and so give a general view of their domain of application.

This classification could be very useful as an entry point in our system. However we prefer use the more recent work of Kardell [Kard97] who made a interesting taxonomy.

He presented four general fields of classification in which he analyzed some subcategories:

- ***applies to:*** object / object families / related object families

- ***purpose:*** instantiation /functionality / interface / communication physicality / access / state

- ***scope:*** static / dynamic

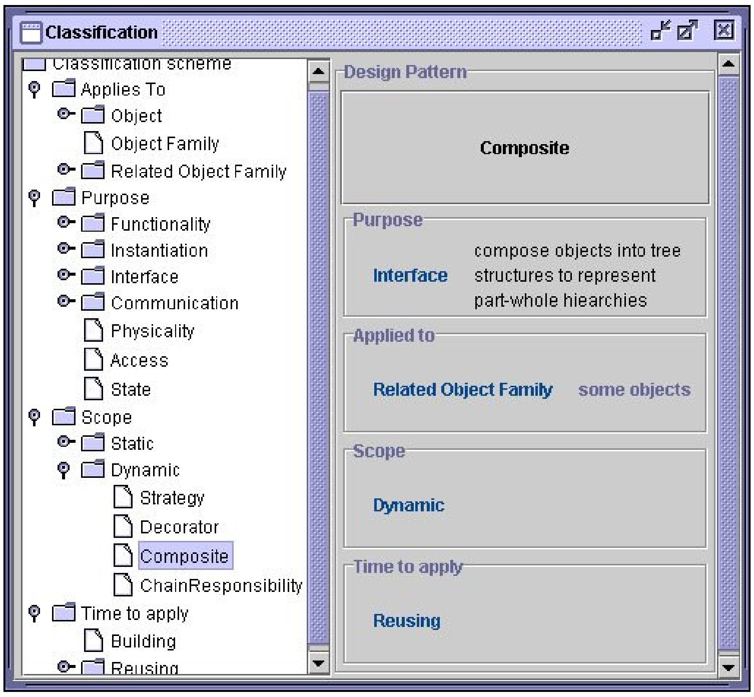
- ***application time:*** building /reusing

The subcategories are exclusive: in a same field a design pattern cannot belong to two categories. This feature is fundamental because it approves the consistency of the classification. If a design pattern could be found in two categories of a same field, it will signify that the categories are not strict enough to classify the design patterns but are just indications.

Such a classification can be very useful in our system. It defines a rigorous organization of the design patterns' set which could help the user to begin the navigation. For example, a user wants to be able to change an object at run-time. Then, he will look at the category *scope – dynamic* and get a first selection of design patterns.

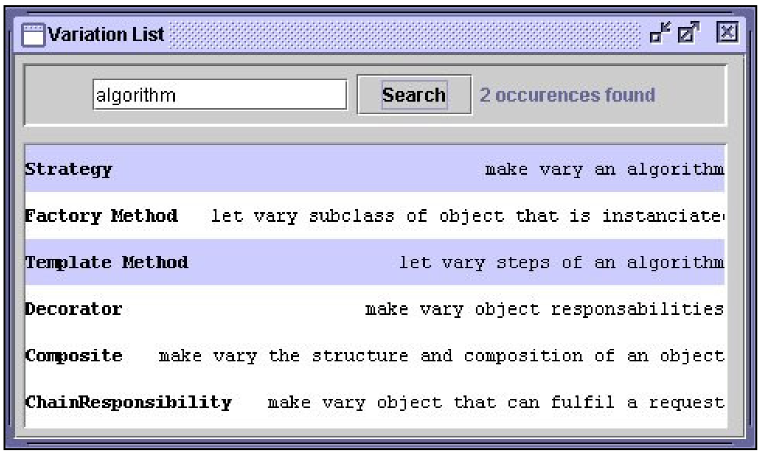
Then he will certainly look at one of these design patterns. This design pattern will be exhibited as belonging to one specific category for each field of the classification as shown in the previous figure. We think that such a presentation should question him on the signification of the other categories. So, for example, he would try to know with which type of entities his specific problem deals (*applied on one object? a family of objects?*) or which is its general purpose (*to create an interface? To instantiate?*). Those steps will certainly lead to a first design pattern susceptible to interest the designer. From this first entry, the user will be able to start a navigation towards an appropriate solution.

Figure 17: Design patterns classification window



5.4.1.2.3. Design Patterns for variation purposes

According to their definition, all the design patterns can be considered design strategy with a variation purpose. The GoF's book provides a list associating design patterns and their variation purposes. We also believe that this list could be an efficient entry point in the design patterns set. Indeed, if the user analyzes his problem under the viewpoint of the variation (for ex. " *I want this object being able to vary*"), he will certainly find a solution in this list.



Strategy	make vary an algorithm
Factory Method	let vary subclass of object that is instanciate
Template Method	let vary steps of an algorithm
Decorator	make vary object responsabilities
Composite	make vary the structure and composition of an object
ChainResponsibility	make vary object that can fulfil a request

Figure 18: Variation purposes window

5.4.1.3. Conclusion

It may be considered that an automatic suggestion of design patterns requires to limit the use of design patterns to fixed situations. Moreover, it should certainly involve some artificial intelligence practices that should lead us far out from the scope of this thesis. Therefore, such a solution cannot practically replace the work we did in this thesis.

For beginning the navigation, our help system provides some classical but significant entry points:

- the design patterns list,
- a taxonomy,
- a list of variation possibilities offered by the design patterns
- a list of design defaults that the design patterns should help to solve.

5.4.2. A navigation map

For navigating, a map is an efficient guide. Indeed, it offers a synoptic view of the set of entities present in the environment and the way they are connected. In our case, we are going to navigate in an environment of entities much more abstract than cities and roads. But we believe that a navigation map could be an interesting practical support to guide the designer in the design patterns' set.
A first part introduces our navigation map where the relationships are shown as routes towards other design patterns. Then we discuss how to take benefit of this map to trace the path of the user.

5.4.2.1. Showing all the relationships

Our purpose with a map is to offer a synoptic view of the design patterns' set. It should show all the design patterns and their relationships.
We analyzed four different types of relationships between the design patterns:

- *use of*
- *alternative*
- *similarity*
- *association.*

A map becomes unreadable if we show all the relationships in once. But since our support is electronic, the map can be displayed under different angles with a simple switch. Then, it may be possible to define a clean map for each relationships. But the practice showed that using different colors, two different types of relationships in once still let a map comprehensible.
Therefore our help system provides a navigation map of the design patterns' set accessible from two different points of view:

- *use of* & *associations*
- *alternative* & *similarities*

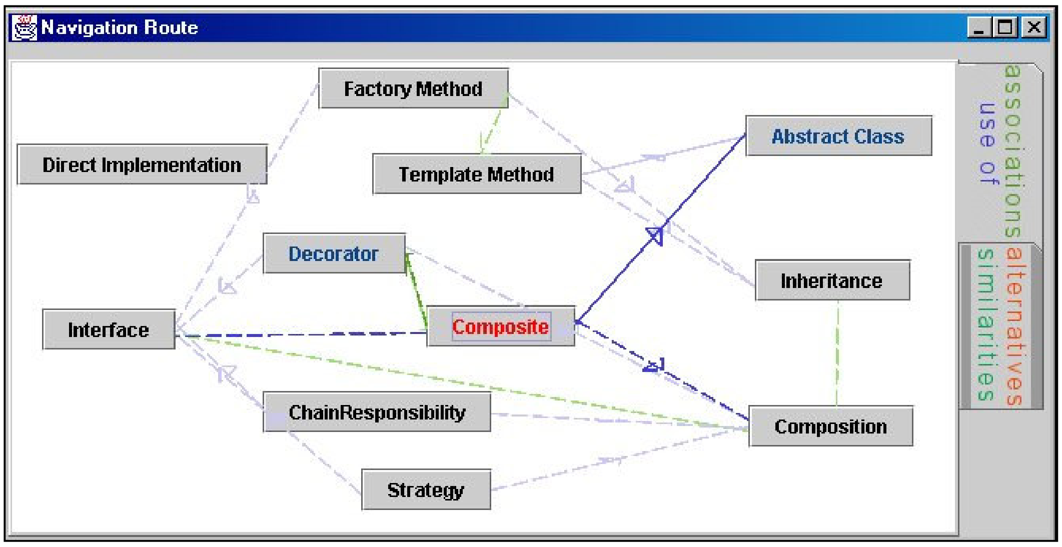


Figure 19: Navigation map

In such a navigation map, if the user looks at the relationships " *use of*", he can easily identify the design principles composing each design pattern. In the example of the figure, *Composite* is selected (it signifies that the "reoriented description" of *Composite* is displayed in the same time). It may be interesting to notice that the map highlights the relationships related to the design abstraction selected. It appears clearly on the map that *Composite* uses *interface, composition* and *abstract class* in its solution. There is also displayed an association with *Decorator*. Similarly if we display the map from the other angle, we will see the alternatives and similarities of *Composite.*

5.4.2.2. Displaying the state of the navigation

It may be pertinent to show also in our navigation map the state of the navigation. We mean by *state of navigation* the path already explored by the user and where he is at the moment.

As explained in § 5.4.1, the most current use of design patterns is to recognize directly during the design a situation which involves a well-known design pattern. For example, most of the designers knows the MVC design pattern. Consequently, the majority have the tendency to apply it when constructing an interface. That is the same process with the other design patterns: the best way to use a design pattern is still to recognize its usefulness directly at design time. Therefore a contextual help system for the use of design patterns must provide a friendly way for the designer in order to increase the number of design patterns he acquired.

In this sense, it may be essential to assist the user in his navigation allowing him to explore freely the design patterns. We mean by freely that the user should not have the apprehension of loosing his first goal.

It may be considered to keep trace of what the user did: the relationships and the design abstractions he explored, the design abstractions he evaluated. All of this should be explicitly displayed in our navigation map. Then only he will be tempted to explore new directions.

For example, a designer can evaluate *Composite* as proper for his situation. There is a common association between *Composite* and *Decorator*. He explores also this way for the case it could be useful for its situation. In looking at the description of *Decorator,* he discover that *Strategy* is an alternative of *Decorator*. Then only by curiosity, he could give a rapid glance at *Strategy.* Thus, he goes back easily to *Decorator* since it was traced on the map.
All this process could have been done in very few time since he has only proceeded five hyperlinks between *Composite* and *Strategy*. In this case, this navigation gave no additional benefits to the research of the solution, but now the designer knows more about *Decorator* and *Strategy*. At first glance, this navigation could seem useless but it allowed the user to extend his knowledge of the design patterns' set.

Therefore, the map will let appear clearly the path explored by the designer and the results of his evaluations. Practically, all the hyperlinks towards design abstractions will indicate with a particular color of the text if

- it was already explored,
- it is at present displayed and activated in the main windows,
- it was not yet explored.

On the same purpose, the relationships will be stipple as long as they will not be explored.

It may be considered that to display the path of the user allows him not only to keep trace of its analyze but also to see clearly what he did not yet explore.
For example, the designer could evaluate *Decorator* and consider it as appropriate. Looking at the map of the ***alternatives and similarities***, he notices that he **did not yet explore** a *similarity* with *Decorator*: the design pattern *Composite*. He could look at this relationship and try to understand the reasons that make *Composite* and *Decorator* similar. Doing that, he will certainly discover subtleties in the two of them [3] and so improve his global knowledge of the set. And we should even imagine that he will discover that actually *Composite* is more appropriate to his problem.

This example illustrates that it could be useful for the user to see the path he did not do. And that this feature is as necessary for assisting the navigation that to display the path he did.

[3] We discuss this phenomenon in §3.3.2.3: Relationships between Design Patterns.

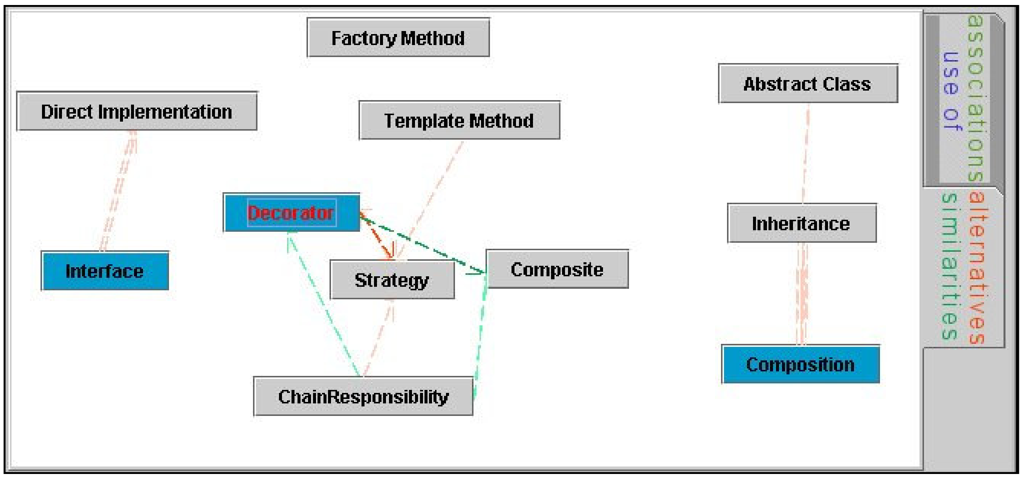


Figure 20: Tracing of the user route on the navigation map

Finally, the navigation map increases the visibility of the design patterns' set. It highlights the route of the user giving a trace of his activity: what he did and by the same way what he did not. Consequently it invites the designer to learn new design patterns and participates actively to the search for solution. For those reasons, we believe that the map enhances considerably the navigation.

5.5. An open assistant versus an intelligent tutoring system.

Our help system can be qualified of open assistant since it assists the user to find a solution more than it gives a solution. At first glance, it always seems better to have a system which provides practically some solutions. The user will be reassured: he does not have to make the effort to learn the design patterns. The project manager will just have to give his trust in a stable automatic tool and will not have anymore to repeat weekly that the use of the design patterns should be increased. This solution is effectively better at short term.

However, we believe that design patterns are design experiences that the designer should not only reuse as clip-arts but that they must influence his own style. Considering that the research into the automatic application of design patterns is already offering satisfying results, with a strict problem solver, the style belonging to the design patterns will progressively be skipped for the profit of concrete and immediate results.
But the design patterns participate in the formation of the designer. They increase his knowledge and enhance his technique. At long term, the designer will be better and consequently more efficient.

Moreover, to use a strict problem solver as an Intelligent Tutoring System could be, requires to determine explicitly the design patterns use cases. Indeed, such a system should know for each design pattern some appropriate uses (see

§1.3.2.1). However, the design patterns are design abstractions. As such, they could be used in situations that might not have been imagined by their creators. An help system should let some place to innovative uses of the design patterns.

The openness of our system is a strength since it lets some room for the user to imagine innovative uses of design patterns. As explained in [ZlZu91] a scientific paper considering creativity and pedagogy, Boris Zlotine defines some necessary conditions for developing the creativity:

- presence of uncertainty : a problem that can no be solved by known methods
- the freedom to work without instruction as how the work should be done in what order, etc.
- the dependence of the results on the particular individual: his experience, intuition, will power, etc.

A rigid system is going against those three principles when our help system is entering those criteria.

- *presence of uncertainty* : the help system is just a guide and do not give solution.
- *the freedom to work without instructions* : The help system do not direct by force in any directions. It is just assisting the communication and the comprehension of the knowledge.
- *the dependence of the results on the particular individual:* The solution remains the choice of the user; the help system guides, gives some advice but does not impose anything.

The help system presents a positive environment for enhancing the creativity of the designer: it just guides him and never commands him.

It should serve to meld the knowledge of the design patterns' set with the experience of the user. Actually, we can consider our help system as a bridge between the subjective style of a designer and the approved style of design patterns.

Our help system offers the possibility of a double movement. On one hand, the designer's techniques will be enhanced by the influence of the design patterns. On the other hand, the design patterns' use should be excelled by the creativity of the designer. We believe that such a system should make

- better designers, with more skills
- and better solutions, more adapted to the specific problems and not only classical applications of the design patterns.

5.6. Towards a contextual help

We believe in the fact that making our help system contextual should improve its benefits. Indeed, our help system can assist (without forcing) the designer in fighting against natural reflexes which limit his knowledge of the design patterns' set. In a first part we are going to analyze those reflexes. Then we will see how a contextual help system could correct such a behavior.

5.6.1. Common behavior of the user

The time pressure leads most of the people to be satisfied with their acquired knowledge. Before good solutions, people look for rapid solutions which could let them more time for solving other problems.

For example, we can imagine that a designer finds that *Decorator* could more or less suit to his present problem. He is not needing all the advantages of this design pattern for his particular problem but modifying a little bit the structure he should get an acceptable and light solution. The designer likes particularly *Decorator* because he have already used it several times and he is now very comfortable on it. He is so comfortable that he begins to make large variations on its structure to fit to his new problems, letting his creativity adapting the thinks. It was one of our purpose to let the creativity of the user: this designer really integrated *Decorator*. However, he is so comfortable on it that he is not looking for other solutions: "why to spend more time on this problem?". But the fact is that in his particular problem *Composite* was much more adapted to his problem. The role of the help system should be to suggest him to have a look to this other design pattern similar to *Decorator*.

It is important to remind that one of the benefits of the design patterns is to provide a common vocabulary between designers. As we discussed in § 2.4, we believe that this purpose should not limit the use of the design patterns to the one described in the intent. However, we believe that if a solution was already listed in the design patterns' set, this solution should be used because the design patterns enhance the communication and comprehension of the design.

Consequently, it may be considered that a help system should encourage the designer to learn always a little bit more about the design patterns' set and not to stay satisfied with his acquired knowledge.

5.6.2. Contextual assistance of the help system

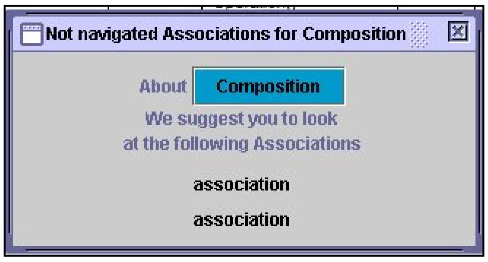
In reaction to the common behavior of the users, a help system should invite him to learn each time a little bit more about the design patterns' set. This part presents that this purpose is achieved thanks to the navigation map, some additional suggestions and a special sorting of the lists. Finally, we discuss how to trace the level of the user to make possible such features in our system.

5.6.2.1. Navigation map

We have already discussed the benefits of the navigation map in § 5.4.2. It increases the visibility of both the set and the route of the user. Doing that, it invites to learn new design patterns and so participates actively in our aim of fighting against the reflex of being contented by one's present skills for solving problems.

5.6.2.2. Additional suggestions

It may be necessary to suggest, from time to time, the user to explore some relationships. For example, after a certain rate of acceptation of the advantages of a design pattern, the system could highlight some *associations* the user did not explore yet.



Not navigated Associations for Composition

About Composition

We suggest you to look at the following Associations

association

association

Figure 21: Suggestion window

Similarly, our help system will suggest some unexplored *similarities*. It will do the same with some unexplored *alternatives* after a certain rate of drawbacks.

Nevertheless, we think that those suggestions cannot be systematic or the system will very quickly begin to be heavy. The user must be assisted in finding a solution and in knowing more the set but he must never be forced in any directions. Of course, it was only suggestions, but a suggestion which is systematic appear like a constraint for the user.

Therefore we believe in the necessity to make those suggestions randomly and especially depending of the knowledge level of the user. Indeed, if the user already knows some relationships, it is maybe not useful to remind him their existence by a suggestion, even if he did not yet explored them in this session. At least, in such a situation, the random rate of suggestions should be decreased.

5.6.2.3. Sorting of the lists

It may be considered that to change the appearance of design patterns' descriptions could entice the designer into looking at unknown features.

The cognitive domain gives us a good example to illustrate this phenomenon: the restaurants. In most of the restaurants, the people are always doing the same kind of actions: they look to a waiter, indicate the number of persons and then they sit. It is a so common habit that we applied it in all the restaurant without really paying attention to the type of restaurant it is. Actually, we will very often begin to look at the restaurant after those formalities. However if something special happen in the restaurant when you entered, a chicken flying or the cook hunting a cat, we will have immediately a different look at this restaurant and will remark some details of the atmosphere (the ceiling, the color of the ground...) we should not have observed in a classical context.

In this sense, it may be necessary to change the lists' ranking of our help system depending of the trace of the user. The elements (consequence, entry point description) already evaluated or explored by the user should be relegate to the end of the list. Similarly, unknown elements should be first in the list. It may be considered that such small modifications should be interesting for encouraging the user to look at new knowledge.
For example, a designer used several times *Strategy*. Each time, the order of the lists of advantages and drawbacks were the same. When inspecting it once again, the user will certainly only look at the first elements of the lists which will remind him the use of *Strategy* as he understood it previously. However, if this order is changed, he will encountered elements that he was certainly aware of but that he has maybe never really took into account.
Showing design patterns under different fluctuating angles, a help system could entice the designer into reconsidering always his knowledge of the design patterns' set.

5.6.2.4. Aiming to trace the level of the user 's knowledge

To make such a contextual help, the system needs to know the level of the user's knowledge.

Each session of our system should be dedicated to a specific problem. Indeed, the navigation map and the evaluations become useless if the system is used for solving two problems in the same time. For example, a question could be how to recognize evaluations corresponding to one specific problem. Therefore, for each problem, a new session should be initialized.

However, the level of the user changes from a session to another: at each session the designer will acquire a little bit more of knowledge about the design pattern' set. Consequently, the level of the user will depend also of his experiences with the help system.
Then, it should be necessary to trace the history of the user i.e. to record the design abstractions and the relationships he already explored, with the number of times, if they were evaluated and when was the last time. This data will be useful to try to adapt the system to the user's characteristics.
For example, a user is evaluating positively *Decorator* but the system knows, from the user's history file, that until now he never looked at *Composite* nor at the

similarity between *Composite* and *Decorator*. In this case, it will be very interesting to suggest this relationship which he is certainly not aware of. Similarly, the log data could indicate that the user have already evaluated *Composite* several times and that he explored the relationships between the two design patterns few time ago. Then the suggestion could be really superficial.

Some history file about each user are useful for knowing his level and than being able to give a proper help. However, it may be difficult to know about the level of the user's knowledge when he has never used the system. Here we believe that a list of judicious questions could give an initial overview of his knowledge. It could begin by some general questions on his knowledge of the set. For example, we should ask for each design pattern of the set if he

- has never heard about,
- knows the existence,
- has already used it,
- perfectly dominates it.

Then some more subtle questions could give information about his knowledge of the relationships. For example, " *Do you know that there are some relationships between the design patterns?*". Then we should attend to see if he is aware of the main relationships between the design patterns he has already used.

Finally, to trace the level of the user's knowledge seems technically attainable. However, in the scope of this thesis, we did not find the time to include it in our prototype. But it may be a significant feature which should lead to interesting studies about artificial intelligence techniques to be able to guide the user without making a rigid system.

5.7. Conclusion

In this chapter, we presented a prototype of a help system. Our system appears as an advanced support for the displaying the design patterns' catalog since it furnishes a consistent help to the research of solution.

First, the system perm its a coherent evaluation of design patterns by checking the consequences. It keeps trace of the evaluations in changing the colors of design patterns and with a visual summary of the results.

Moreover, the help system supports efficiently the navigatio n through the design patterns' set. It provides various and effective entry points in the set. A navigation map helps to increase the visibility of the set and highlight the route of the user. Doing that, it keeps up the constant learning of design patterns and facilitates the research for solution.

Finally it was discussed the fact that an open help system, as we did, gives substantial benefits compared with a strict problem solver into the context of this application. It improves the designer style and knowledge and increases the suitability of the solutions.

Nevertheless we give contextual directions that may be considered to encourage the user to learn each time a bit more about the design patterns' set.

6. CONCLUSION AND FUTURE WORKS

In this chapter, we discuss the results of our research work. In order to be more explicit in this conclusion part, we reformulate our aims and motivations. Afterwards, we summarize our approach and finally, present the perspective works for this thesis.

6.1. Motivations and initial goal

The first steps at the origin of this thesis were in the direction of guiding the design patterns' use in legacy systems. Design patterns are beneficial tools but it is important to note that they have a complexity cost. So, their use cannot be easily systematize. Some initial investigations in the literature exposed rapidly that design patterns' related activities are numerous and various. However, the design patterns' use appears as being a burning point.

- First, we noticed a general confusion about the "right" way to use design patterns,
- secondly, it seems that there is no available tool to assist effectively the designer in using design patterns.

Therefore, this first analyze directed this thesis towards the study and development of a prototype of Intelligent Tutoring System (ITS) for teaching design patterns' use.

For achieving this purpose, it was first necessary to emerge of this general literature disorientation about design patterns' use. Then we defined our position on this topic: the design patterns are significant communication tools but **their use that cannot be fixed.**
Because of design patterns particular characteristics, it was essential to find some technical ideas in order to develop the tutoring system. On this purpose, we continued our explorations in the literature.

This analyze demonstrated that an ITS was a hard goal to attain in the context of the design patterns' use. Indeed, we consider that the design patterns' use cannot be fixed. Indeed, it is difficult to determine an "accurate" use of design patterns on which base a practical ITS. Consequently, it was more realistic to work in the direction of a contextual help system to assist the designer in using design patterns. Moreover, with regard to our investigations in the literature, it appeared that such a work could be effectively relevant for our research field.

6.2. Summary and conclusion

This thesis report relates the results of our researches about the design patterns' use and learning by OO software designers. On those basements, it presents also our prototype of a contextual help system for assisting the OO designer in using correctly design patterns.

The first chapter explains our approach, the evolution of our objectives and presents the structure of report.

In the second chapter, we presented that the design patterns are very beneficial tools but their use is not simple ([GoFo95]). Indeed, their application has a complexity cost and the designers cannot apply them in a systematic manner. This difficulty involves the fact that designer needs a good knowledge of the whole design patterns' set in order to use design patterns correctly.
But for taking some benefits of design patterns, it is required to exceed this appearing difficulty. Numerous design patterns related activities work in this direction. We distinguish two main fields in this research area: the specification of design patterns and the tools for assisting the design patterns' use.
The specification is a recent activity but some reliable formal basements for specifying design patterns were recently introduced with LePus [Eden98b].
About the tools for assisting the designer in using design patterns, it appears some consistent works available for implementing the design patterns ([FMPW97], [Toge00]), and for detecting their utilization in legacy systems ([KrPr96], [Brow95], [Pina99]). However, it seems that there is no satisfying tool available to assist the suggestion of design patterns ([Corr99]).

One of the reasons for this deficit seems to be the general confusion in the positions about the design patterns' use ([GiLo97], [Pehu00]). To emerge from this situation, it was necessary to refine our position on this topic: in this work, the design patterns are considered as significant communication tools but their use cannot be fixed**.**

On the base of this position, we have identified and shown in the third chapter which topics could help the designers to use design patterns. For using correctly the design patterns, it is necessary to have a reliable knowledge and understanding of the design patterns' set. Moreover, to assist in using design

patterns for solving a problem, a help system must support the research of the solution. On that purpose, it seems interesting to navigate towards this solution and to evaluate the suitability of propositions.
This analyze results practically in the necessity to highlight some substantial but disregarded features of design patterns: their advantages and drawbacks, the design principles which compose them and their relationships with others.

To satisfy those requirements, we have introduced in the fourth chapter, a new presentation of the design patterns' description. In the scope of this thesis, this work could not have been based on LePus, the specification language we presented in the second chapter. However, we proceeded available works about the design patterns' description ([GoFo95], [GuEr99], [Kard97]) and their relationships ([Zimm96]) to define a new coherent and structured format.
It results in a presentation centered on the design structures encapsulated in design patterns which clears up the design principles which compose them. It presents also the improvement to expose explicitly advantages and drawbacks of design patterns in separated lists. Moreover, we reformulated the intent with a complete and coherent pattern. Finally, this presentation evinces the nature of the relationships between design patterns.

In addition to be compact and comprehensible, our presentation of design patterns puts forward the features necessary for enhancing the design patterns' use that were defined in the third chapter. As such, it provides a consistent base for the development of a help system.

Therefore, in the fifth chapter, we were able to introduced a help system for using design patterns. Supported by our new presentation of design patterns, our help system assists the designer in finding some solutions to his design problems in the design patterns' set. First, it favors efficiently the navigation through the design patterns' set towards a solution with a navigation map. Secondly, it supports a coherent evaluation of design patterns by checking the pertinence of their consequences and tracing the results.
Moreover, our prototype integrates some contextual features which invite the user to learn always more about the design patterns' set.

Finally, we discussed the results of our help system and highlighted two directions which may be considered:

- it improves the designer style and knowledge,
- it increases the pertinence of the solution.

In this thesis, our analyze shows the clear necessity to elucidate the design patterns substance to enhance their learning. Moreover, our prototype of help system illustrates the interest of the navigation through the knowledge for assisting the research in a knowledge base like design patterns' set.

In conclusion, this research project demonstrates that in order to enhance the design patterns' use, it may be considered to work on the comprehension of design patterns as much than on their communication and application as most of design patterns related activities do.

6.3. Future works

Our research results in a substantial and original solution to assist the designers in using design patterns. But, in the scope of this thesis, our system could not reach further than the stage of a prototype. In the perspective of a more consistent tool, the following directions should be considered:

- *Concrete experimentation of the prototype*:
  Our propositions need to be verified in a practical context. It may be significant to test the prototype with some designers of different levels. Then the analyze of the various reactions should highlight the main defaults and main qualities of our system. This study is fundamental to give reliable basements for any further works.

- *Use of a strong formalism*:
  It may be considered to use a strong formalism as LePus to refine our structure of the design patterns' description. Such formalism could specify explicitly the relationships between design patterns and so enhance the navigation towards a solution. Moreover, it could strength the relations between the structure of a design pattern and its consequences. This work should help to elucidate the design strategies encapsulated in design patterns.

- *Extension to the entire design patterns' set*:
  Our work was based of a restricted set of design patterns. A functional help system should consider the entire set of the GoF's book. It may be also significant to extend this set with some other design patterns.

- *Amelioration of the contextual aspect:*
  The contextual aspect of a help system should take into account the directions we give in the last chapter. As explained in §5.6.2, the user's level and experience should be traced in order to offer some substantial contextual features. Moreover, it may be considered to integrate some open questions to challenge the comprehension of the user ([Kuro00]).

- *Design problems detection*:
  The detection of design problems could serve as an interesting entry point in a help system. However, as presented in § 5.4.1.1.1, it may be necessary to enhance the available works with an stronger interaction with the user and some artificial intelligence techniques.

- *Association with other CASE tools for design patterns*:

A help system for assisting the design patterns’ use should also consider the available CASE tools. Typically, it may be interesting to offer an access to a tool automating the application of design patterns when a solution is found.

7. INDUSTRIAL AND ECONOMIC IMPACT OF OUR RESEARCH

7.1. Design Patterns and software industry

7.1.1. Software Reuse

Reuse is becoming a revered commandment in the software industry. In every domain, this notion appears. A lot of efforts were done into better programming languages, tools, and techniques: COBOL, structured analysis and design, then OO languages and OOA&D processes. This enthusiasm is easily understandable when look at the **benefits of reuse** on the software industry.

On this subject, Martin Griss from HP and Ivar Jocobson from Rational presented during the international conference OOPSLA'97 (Conference on Object-Oriented Programming, Systems, Languages, and Applications) [Oops97] a concrete analysis of those benefits. They worked in collaboration with the major software companies of the market like AT&T, Brooklyn Union Gas, Ericsson AXS, HP, IBM, NETRON, REBOOT, and Microsoft. They concluded that reuse provides statically the following benefits:

Time To Market Reductions	1.5 to 2 times
Quality Improvements	5 to 10 times
Maintenance Cost Reductions	2 to 5 times
Development Cost Reductions	15% to 30%

Table 5: Software reuse benefits

Considering this significant feedback, the reuse is rightly considered as an essential principle in the software industry.

7.1.2. Design Patterns: tools of reuse

Software developers always had a strong tendency to **reuse designs** that have worked well for them in the past. Then, as they gain more experience, their repertoire of design experience grows and they become more proficient. Unfortunately, this design reuse is usually restricted to personal experience.

However, considering the significant benefits of reuse, it was necessary to share design knowledge among developers. On that purpose, software patterns appeared in the software community and with them **design patterns**. A design pattern encapsulates a experimented design strategy solving general problem in a particular context.

The availability of a catalog of design patterns can help both the experienced and the novice designer to recognize situations in which design reuse could or should occur. Such a collection is time-consuming to create, but as explained in the following parts, the invested effort pays off.

7.1.3. Design reviews about the design patterns' usage

Since their introduction, the design patterns' use increased rapidly in the software community. To answer the demand for concrete feedback, a group of industrial worked on reviewing the design patterns' use ([CoBe96], [Co&al97]). They observed the following proceeds:

Patterns ...	FCS	AT&T	Motorola	BNR	Siemens	IBM
are a good communications medium	✓	✓	✓	✓	✓	✓
are extracted from working designs	✓	✓	✓	✓	✓	✓
capture design essentials	✓	✓	✓	✓	✓	✓
enable sharing of "best practices"	✓		✓	✓	✓	✓
are not necessarily object-oriented		✓	✓	✓	✓	
should be introduced through mentoring	✓				✓	✓
are difficult/time-consuming to write			✓	✓	✓	
require practice to write					✓	✓

Table 6: Design reviews results about design patterns

These results show clearly the position of the software industry on the design patterns: the use of design patterns can have a significant impact on the way a team develops software. The improved communication through patterns alone is a valuable asset. Giving novices the opportunity to learn from positive examples which already form the basis of a shared team vocabulary can help speed their contribution to the team. However, it is noticeable that half the companies claim that a pattern mentor is necessary to support the design patterns' use in the work teams.

7.2. Our help system

The software industry is aware of the substantial benefits of design patterns and pretends to take dividends of their use. However, design patterns have a complexity cost and their use appears a delicate task which could be rapidly more harmful than advantageous in the software life cycle. Therefore the software industry may be concerned by a software assisting their use. With regard to our investigations in the literature, it appeared relevant to study such a tool in this thesis project.

Our research analyses results in a prototype of a help system to assist the designers in using design patterns.

Our work targets two main goals:

- to enhance the learning of the design patterns' set,
- to support the research of design solutions in terms of design patterns.

Practically, we suggest a electronic support for using the design patterns' set. With a contextual aspect, our prototype works on improving regularly the knowledge and the understanding of the user. Moreover, some navigation and evaluation devices assist the user to find a solution to his design problem.

It may be considered that our help system results in:

- improving the designer style and knowledge
- increasing the pertinence of the solutions.

7.3. Conclusion

Considering the results of this work, it may be considered that an help system to assist the designers in using design patterns could have a significant interest for the software industry.

First, because it improves the learning of the design patterns, such a system could tend to replace the necessity of a pattern mentor and the costly courses and seminars about design patterns.

Secondly, the help system attempts to facilitate the design patterns' use. Therefore, it may be considered that such a tool will increase the general usage of design patterns and so ameliorate globally the practice of software reuse and its benefits.

8. References

[Alex77] Christopher Alexander , *A pattern language* , New York Oxford University Press, 1977.

[Alex79] Christopher Alexander , *A timeless Way of Building* , New York Oxford University Press, 1979.

[AgCo98] E. Argebo and A. Cornils , *How to preserve the benefits of design pattens*, Computer science department, University of Aahus, Denmark, 1998.

[Coop99] James W. Cooper, *The design patterns: Java Tutorial,* Addison-Wesley Pub Co 1999.

[CoSc95] J.Coplien and D.Schmidt, eds. , *Pattern languages of Program Design*, Addison-Wesley, 1995.

[CoBe96] J.Coplien, K. Beck and al., *Industrial experience with design patterns*, in 18 th Instl. Conf. On Software Engineering IEEE CS Press, March 1996.

[Co&al97] J.Coplien and al., *Reviews of Industrial experience with design patterns,* http://www1.bell-labs.com/user/cope/Patterns/ICSE96/icse.html

[Corr99] A. Correa and M.L. Werner , *Identification of Problematic Constructions in OO Applications : an approach based on heuristics, design patterns and anti-patterns*, thesis report COPPE Universidade Federal de Rio de Janeiro, 1999.

[DpLi00] Mailing lists about design patterns :
- gang-of-four-patterns: http://hillside.net/patterns/Lists.html#gang-of-4
- pattern-discussion: http://hillside.net/patterns/Lists.html#patterns-discussion

[Eden98a] Ammon H. Eden, *Giving "the quality" a name*, 1998, http://www.math.tau.ac.il/~eden/bibliography.html

[Eden98b] Ammon H. Eden, *LePus – Symbolic logic modeling of object oriented architectures: a case study.*, 1998, http://www.math.tau.ac.il/~eden/bibliography.html

[Eden99] Ammon H.Eden, *Precise specification of Design Patterns and tools support in their application,* Phd. Dissertation, September 1999.

[FMPW97] G. Floriji n, M. Meijers, P. van Winsen, *Tool support for object oriented pattern.,* Proceedings of ECOOP' 97.

[Ga&al96] D. Garlan and al., *Stylized architecture, Design Patterns, and Objects,* National Science Foundation CCR-9357792, September 1996.

[GiLo98] Joseph Gil and Davi H. Lorentz *Design Pattern and Language Design,* in Computer Vol. 31, No.3, March 1998.

[GoFo95] E. Gamma and al. *Design Pattern, Elements of Reusable object-Oriented software,* Addison Weisley 1995.

[GuEr99] A. Gustavsson and M. Ersson *Formalizing the intent of a design pattern,* for the course of Amnon H. Eden about object architecture, 1999.

[HHGa90] R.Helm, J.M. Holland, D. Gangopadhyay, *Contracts: Specifying compositions in Object Oriented system*, Proceedings of OOPSLA 1990.

[Kard97] Magnus Kardell, *A classification of object oriented design patterns,* master's thesis, Umea University, Denmark, 1997.

[Brow95] Kyle Brown, *Design reverse engineering and automated design pattern detection in STK,* Master thesis, 1995 (http://www2.ncsu.edu/eos/info/tasug/kbrown/thesis2.htm)

[KiBe96] Kim J. and K. M. Benner (1996). *Implementation Patterns for the Observer Pattern*, [VCKe96].

[Keri00] Joshua Kerievsky, *A Learning Guide To Design Patterns,* Industrial Logic Inc., http://www.industriallogic.com/papers/learning.html.

[KrPr96] Christian Krämer and Lutz Prechelt, *Design Recovery by automated search of structural design patterns in OO software.,* in Proc. of Working Conference on Reverse Engineering, 1996.

[Kuro00] Brian T. Kurotsuchi, *Wonderful World of Design Patterns* , http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/patterns/

[Lima92] J. C. M Lima. *Towards an intellignet system for the building of diagnostics in tutorials.* Phd. Dissertation, University of Paris VI, Octobre 1992.

[Lima95] J. C. M Lima. *Knowledge representation in software packages aimed to know about their users.* In "Advances in Database and Expert Systems". IIAS Editions, ISBN 0921836228, Windsor, Canada, pp 110-114, 1995.

[Lima97] J. C. M. Lima. *Teaching intelligently by computers: a formal model based on an object notation* . Universidade Estadual do Norte Fluminenese, Centro de Ciência e Tecnologia, Research report 12. 97, Campos, Brasil, 1997.

[Oops97] OOPSLA http://oopsla.acm.org

[Pehu00] Chris Pehura site: www.pehura.com

[Pina99] Osvaldo Pinali Doederlein, *Design Pattern extraction for software documentation*, master thesis EMOOSE'99.

[PrUn98] Lutz Prechelt, Barbara Unger, *A series of controlled experiments on Design Patterns: Methodology and Results*, in Proc. Softwaretechnik '98, 1998.

[Rant99] Kalevi Rantanen, *Brain, Computer and the ideal final result,* 1999, http://www.ideationtriz.com

[RLKS98] A. Romanczuck-Requile, J. C. M. Lima, C. Kaestner, E. Scalabrin. *A Contextual Help System Based on Intelligent Diagnosis Processing Aiming to Design and Maintain Object-Oriented Packages.* In "Lecture Notes in Computer Science", ISBN 3-540-65460-7 1543, Springer-Verlag, pp. 64-65, 1998.

[Rohn96] Rohnert H., *The Proxy Design Pattern Revisited*, in [VCKe96].

[Toge00] TogetherSoft (2000) www.together.com

[VCKe96] Vlissides J. M., J. O. Coplien, and N. L. Kerth, *Pattern Languages in Program Design 2.*, Addison-Wesley 1996

[Wiki00] Wiki Pages about patterns: http://c2.com/cgi/wiki?WikiPagesAboutWhatArePatterns

[Zimm96] Walter Zimmer, *Relationships between design patterns,* In J.O. Coplien and D.C. Schmidts (eds.), *Patterns languages of programming design*, Addison-Wesley 1996, pp. 345-364.

[ZlZu91] Boris Zlotin and Alla Zusman, *TRIZ and Pedagogy,* 1991, http://www.ideationtriz.com

9. FIGURES INDEX

10. TABLES INDEX

11. ANNEXES

ANNEX 1

EXAMPLE OF OO DESIGN PATTERN: COMPOSITE.

Name

Composite

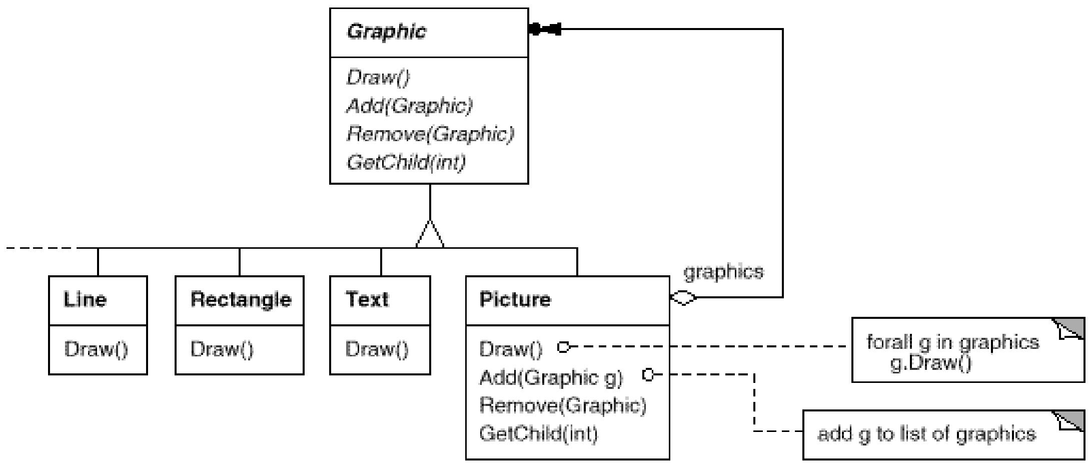
Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Motivation

Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components. The user can group components to form larger components, which in turn can be grouped to form still larger components. A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.

But there's a problem with this approach: Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically. Having to distinguish these objects makes the application more complex. The Composite pattern describes how to use recursive composition so that clients don't have to make this distinction.

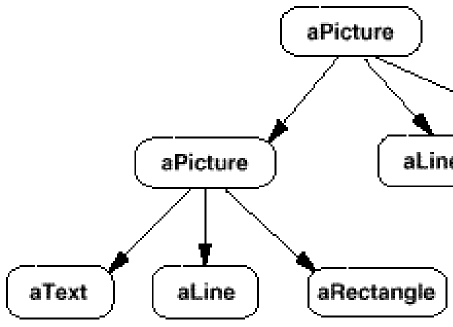


The key to the Composite pattern is an abstract class that represents both primitives and their containers. For the graphics system, this class is Graphic. Graphic declares operations like Draw that are specific to graphical objects. It also declares operations that all composite objects share, such as operations for accessing and managing its children.

The subclasses Line, Rectangle, and Text (see preceding class diagram) define primitive graphical objects. These classes implement Draw to draw lines, rectangles, and text, respectively. Since primitive graphics have no child graphics, none of these subclasses implements child-related operations.

The Picture class defines an aggregate of Graphic objects. Picture implements Draw to call Draw on its children, and it implements child-related operations accordingly. Because the Picture interface conforms to the Graphic interface, Picture objects can compose other Pictures recursively.

The following diagram shows a typical composite object structure of recursively composed Graphic objects:



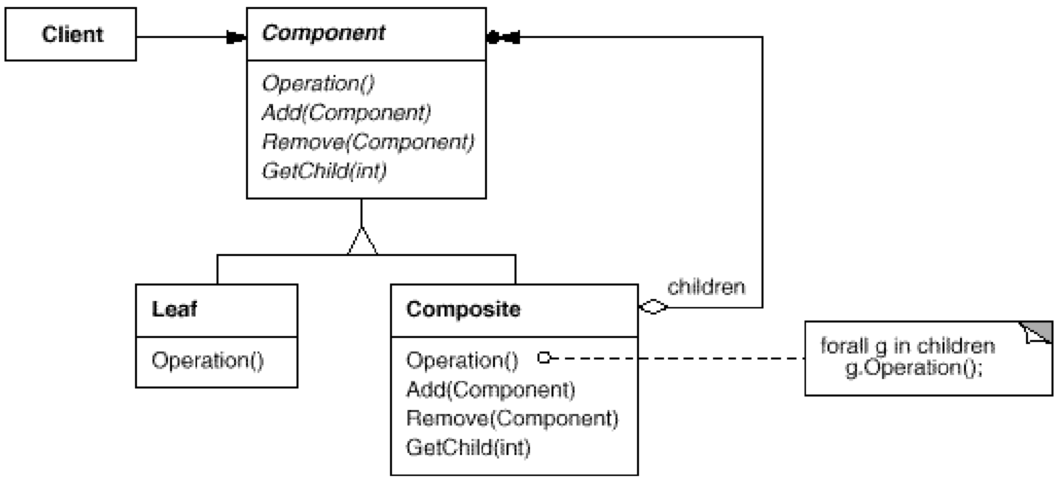
Applicability

Use the Composite pattern when

- you want to represent part-whole hierarchies of objects.
- you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Structure

A typical Composite object structure might look like this:



Participants

Component (Graphic)

- declares the interface for objects in the composition.
- implements default behavior for the interface common to all classes, as appropriate.
- declares an interface for accessing and managing its child components.
- (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.

Leaf (Rectangle, Line, Text, etc.)

- represents leaf objects in the composition. A leaf has no children.
- defines behavior for primitive objects in the composition.

Composite (Picture)

- defines behavior for components having children.
- stores child components.
- implements child-related operations in the Component interface.

Client

- manipulates objects in the composition through the Component interface.

Collaborations

Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Consequences

The Composite pattern

- defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Wherever client code expects a primitive object, it can also take a composite object.

- makes the client simple. Clients can treat composite structures and individual objects uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component. This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition.

- makes it easier to add new kinds of components. Newly defined Composite or Leaf subclasses work automatically with existing structures and client code. Clients don't have to be changed for new Component classes.

- can make your design overly general. The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

Known Uses

Examples of the Composite pattern can be found in almost all object-oriented systems. The original View class of Smalltalk Model/View/Controller [KP88] was a Composite, and nearly every user interface toolkit or framework has followed in its steps, including ET++ (with its VObjects [WGM88]) and InterViews (Styles [LCI+92], Graphics [VL88], and Glyphs [CL90]). It's interesting to note that the original View of Model/View/Controller had a set of subviews; in other words, View was both the Component class and the Composite class. Release 4.0 of Smalltalk-80 revised Model/View/Controller with a VisualComponent class that has subclasses View and CompositeView.

The RTL Smalltalk compiler framework [JML92] uses the Composite pattern extensively. RTLExpression is a Component class for parse trees. It has subclasses, such as BinaryExpression, that contain child RTLExpression objects. These classes define a composite structure for parse trees.
RegisterTransfer is the Component class for a program's intermediate Single Static Assignment (SSA) form. Leaf subclasses of RegisterTransfer define different static assignments such as primitive assignments that perform an operation on two registers and assign the result to a third; an assignment with a source register but no destination register, which indicates that the register is used after a routine returns; and an assignment with a destination register but no source, which indicates that the register is assigned before the routine starts.

Another subclass, RegisterTransferSet, is a Composite class for representing assignments that change several registers at once.

Another example of this pattern occurs in the financial domain, where a portfolio aggregates individual assets. You can support complex aggregations of assets by implementing a portfolio as a Composite that conforms to the interface of an individual asset [BE93].

The *Command* (233) pattern describes how Command objects can be composed and sequenced with a MacroCommand Composite class.

Related Patterns

Often the component-parent link is used for a *Chain of Responsibility* (223).

Decorator (175) is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.

Flyweight (195) lets you share components, but they can no longer refer to their parents.

Iterator (257) can be used to traverse composites.

Visitor (331) localizes operations and behavior that would otherwise be distributed across Composite and Leaf classes.

ANNEX 2

CONSEQUENCES AND RELATIONSHIPS OF DESIGN PRINCIPLES.

Design Principle	Consequence	Advantage / Drawback	Alternative		Association		
			Design Pattern or principle	Affected consequence	Design Pattern or principle	Added consequence	Use intent
Composition	add features dynamically	A	Inheritance	add features at compile-time	Interface	The client is not aware of the specific type of the objects	to make dynamicity available
Composition	flexibility	easy to customized / hard to learn	Inheritance	"white-box" reuse			
Composition	Lots of small objects are created	light hierarchies / behavior depends on interrelationships	Inheritance	add features at compile-time	Inheritance	"white-box" reuse	to react to the lack of available components
Composition	"black box" reuse	keeps encapsulation / requires well-defined interface	Inheritance	"white-box" reuse			
Interface	The client is not aware of the specific type of the objects	more transparency / objects are constrained to the interface	Direct Implementation	Limit the use to the object and its subclasses			
Interface	The client is not aware of the specific implementation of the objects	more transparency / objects are constrained to the interface	Direct Implementation	Limit the use to the object and its subclasses			
Inheritance	add features at compile-time	keeps benefit of the features of the super class / not dynamic	Composition	add features dynamically			
Inheritance	"white-box" reuse	keeps benefit of the features of the super class / inherits an implementation maybe not adapted to the new problem domain	Composition	"black box" reuse			
			Abstract class	provide only the generic implementation			
Abstract class	provide only the generic implementation	direct the behavior of the subclasses / could be not adapted to the new problem			Interface	The client is not aware of the specific implementation of the objects	An abstract class can be used as an interface

ANNEX 3

CONSEQUENCES AND RELATIONSHIPS OF DESIGN PATTERNS.

Design Pattern	Consequence	Advantage / Drawback	Alternative			Association		
			Similarity?	Design Pattern or principle	Affected consequence	Design Pattern or principle	Added consequence	Use intent
Decorator	change skin of an object and not the guts	A		Strategy	change guts of an object and not the skin			
Decorator	add responsabilities	A	X	Composite	add composition	Composite	add composition	To add responsibilty to a composite object
			X	Proxy	control access to an object			
Decorator	make vary object responsabilities	A	X	Adapter	make vary an interface			
Strategy	lets the algorithm vary independently from clients that use it.	A	X	Flyweight	manage shared objects			
			X	State	make vary a state of an object			
				Visitor	make vary operation that can be applied to object			
Strategy	eliminate conditional statements (practical...?)	A						
Strategy	communication overhead if different granularity of algorithm (composition ?)	D		Implementation advice	pass the context as parameter			
Strategy	change guts of an object and not the skin	A		Decorator	change skin of an object and not the guts			
Factory Method	create objects knowing only when but not what	A				Template Method	lets subclasses redefine steps of an algorithm	Factory method are often call by template method
Factory Method	lets class defer instantiation to subclasses	A						

Design Pattern	Consequence	Advantage / Drawback	Alternative			Association		
			Similarity?	Design Pattern or principle	Affected consequence	Design Pattern or principle	Added consequence	Use intent
Template Method	defers steps of algorithms to subclasses	A		Strategy	make vary an algorithm			
Template Method	requires to know which method to override and which not	D		Implementation advice	put name convention			
				Implementation advice	minimize the number of primitive			
Composite	compose objects into tree structures to represent part-whole hiearchies	A						
Composite	make vary the structure and the composition of an object	A						
Composite	lets client treats individual objects and compositions of objects uniformly	A						
Composite	difficult to restrict the nature of the composant in function of the parent	D		Implementation advice	run-time checking			
Composite	composed children do not know their parent	D		Implementation advice	Pass the parent as parameter to the child			
Composite	children are not ordered	D				Iterator	make you vary how an aggregate's elements are accessed, traversed	to traverse Composite
						Decorator	add responsabilities	To add responsibilty to a composite object
Chain of Responsabilities	give more than one object a chance to handle the request.	A						
Chain of Responsabilities	avoid coupling the sender of a request and the receiver	A	X	Decorator	Make vary object responsabilities			
Chain of Responsabilities	Receipt is not guaranteed	D						
Chain of Responsabilities	make vary object that can fulfil a request		X	Composite	treat composite structure and individual objects uniformly			

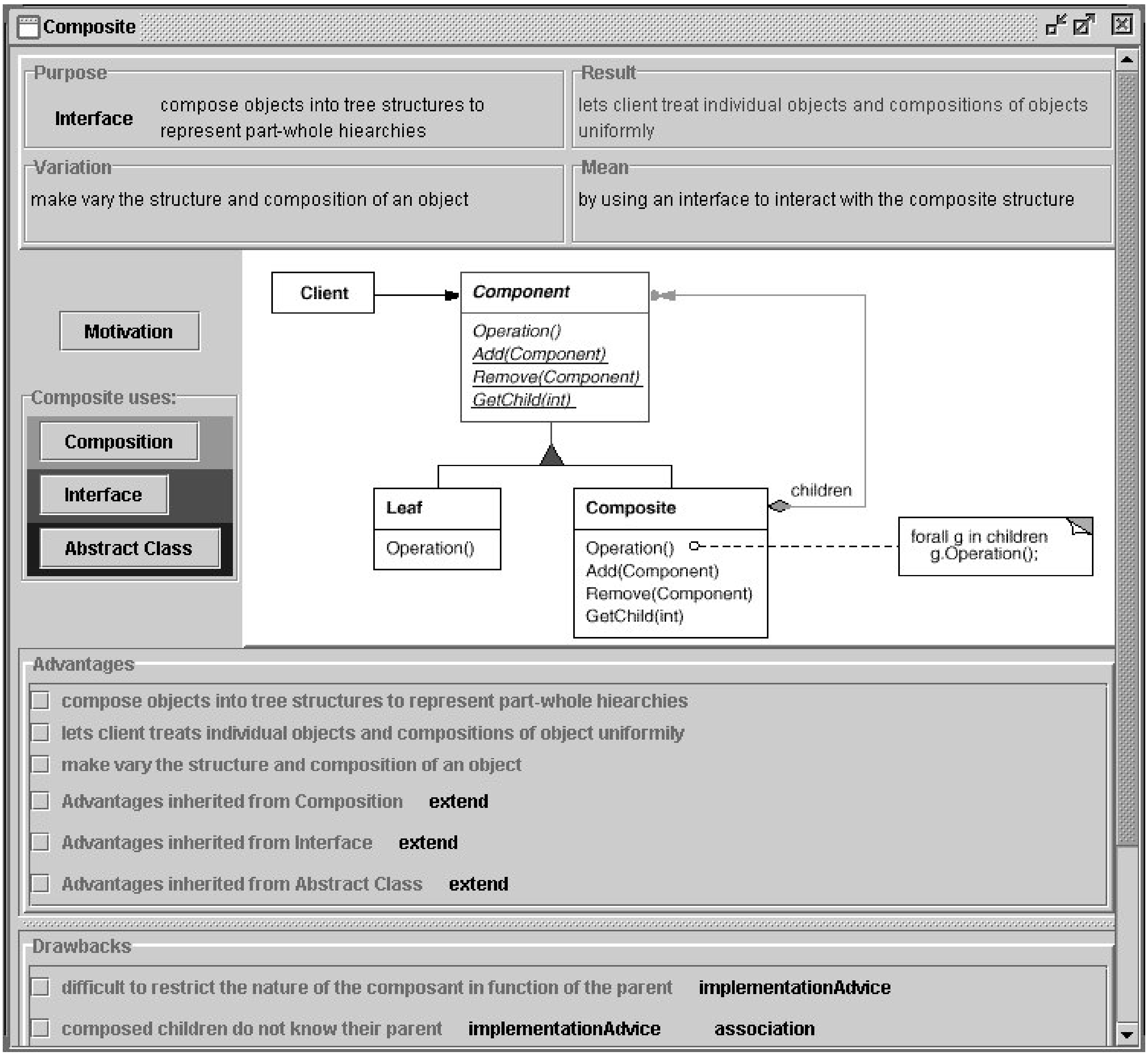
ANNEX 4

INTENT DESCRIPTION OF DESIGN PATTERNS.

	Applied to		Purpose		Scope	Mean (collaboration)	Result	Variation
	Category	Description	Category	Description				
Decorator	Object	an object	Functionnality	add responsabilities	Dynamic	by forwarding requests to the composant	delivers a flexible extending mechanism	make vary object responsabilities
Strategy	Object	an object	Functionnality	make vary an algorithm	Dynamic	by encapsulating each algorithm	lets the algorithm vary independently from clients that use it	make vary an algorithm
Factory Method	Object	some objects	Instantiation	create objects knowing only when but not what	Static	by defining an interface that lets subclasse decide which object to instantiate	lets a class defer instantiation to subclasses	let vary subclass of object that is instanciated
Template Method	Related objects family	a class	Functionnality	defers steps of algorithms to subclasses	Static	by defining the skeleton of an algorithm in one operation	lets subclases redefine certain steps of an algorithm without changing the algorithm structure	let vary steps of an algorithm
Composite	Related objects family	some objects	Interface	compose objects into tree structures to represent part-whole hiearchies	Dynamic	by using an interface to interact with the composite structure	lets client treat individual objects and compositions of objects uniformly	make vary the structure and composition of an object
Chain of Responsibility	Related objects family	some objects	Communication	give more than one object a chance to handle the request.	Dynamic	by chaining the receiving objects and passing the request along the chain until an object handles it	avoid coupling the sender of a request to its receiver	make vary object that can fulfil a request

ANNEX 5

SHOT SCREENS OF DESIGN PATTERNS AND DESIGN PRINCIPLES.



Purpose

Interface compose objects into tree structures to represent part-whole hiearchies

Result

lets client treat individual objects and compositions of objects uniformly

Variation

make vary the structure and composition of an object

Mean

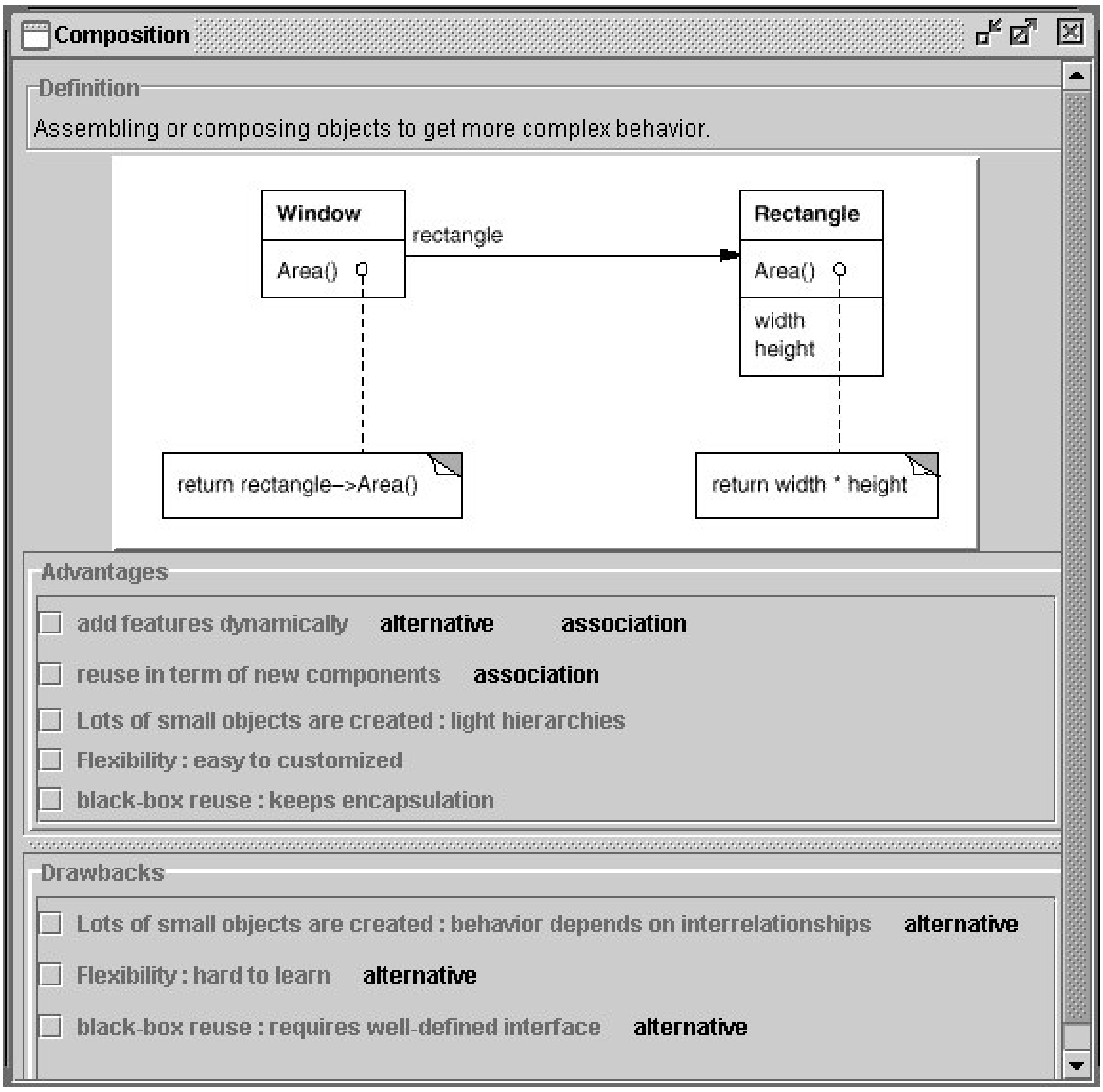
by using an interface to interact with the composite structure

Advantages

- ☐ compose objects into tree structures to represent part-whole hiearchies
- ☐ lets client treats individual objects and compositions of object uniformily
- ☐ make vary the structure and composition of an object
- ☐ Advantages inherited from Composition **extend**
- ☐ Advantages inherited from Interface **extend**
- ☐ Advantages inherited from Abstract Class **extend**

Drawbacks

- ☐ difficult to restrict the nature of the composant in function of the parent **implementationAdvice**
- ☐ composed children do not know their parent **implementationAdvice** **association**



Composition

Definition

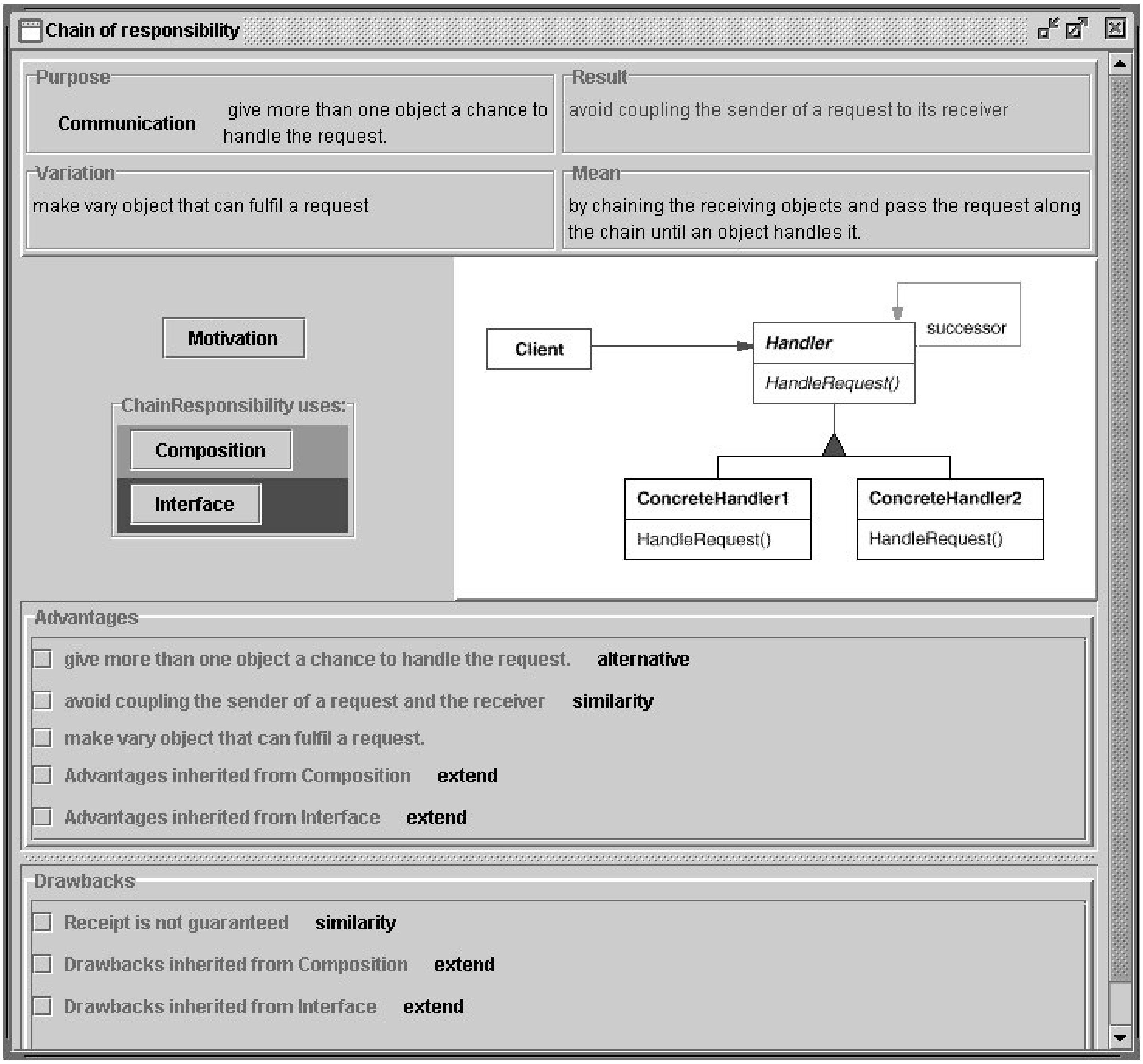
Assembling or composing objects to get more complex behavior.

Advantages

- ☐ add features dynamically **alternative** **association**
- ☐ reuse in term of new components **association**
- ☐ Lots of small objects are created : light hierarchies
- ☐ Flexibility : easy to customized
- ☐ black-box reuse : keeps encapsulation

Drawbacks

- ☐ Lots of small objects are created : behavior depends on interrelationships **alternative**
- ☐ Flexibility : hard to learn **alternative**
- ☐ black-box reuse : requires well-defined interface **alternative**



Chain of responsibility

Purpose

Communication give more than one object a chance to handle the request.

Result

avoid coupling the sender of a request to its receiver

Variation

make vary object that can fulfil a request

Mean

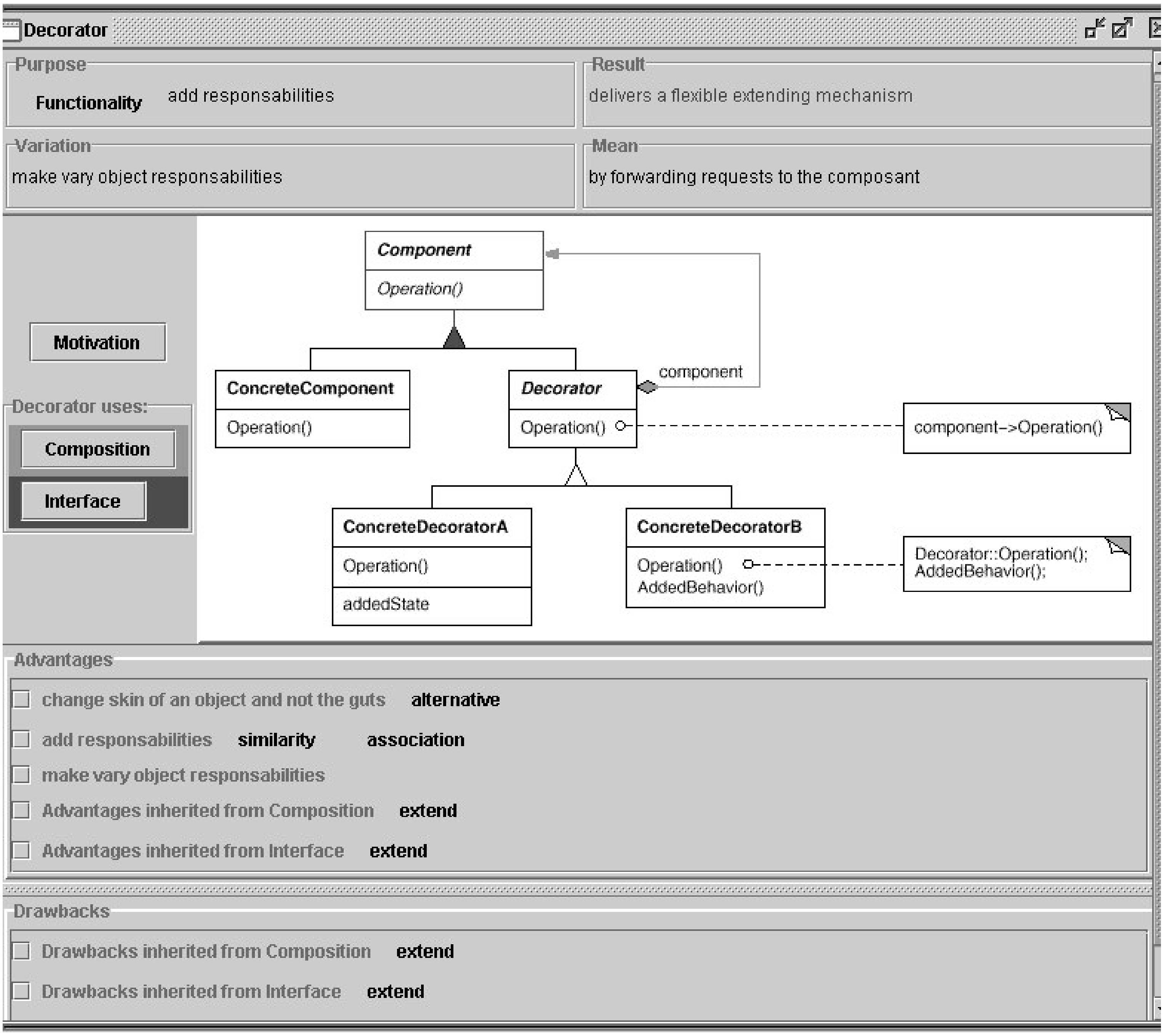
by chaining the receiving objects and pass the request along the chain until an object handles it.

Advantages

- [ ] give more than one object a chance to handle the request. **alternative**
- [ ] avoid coupling the sender of a request and the receiver **similarity**
- [ ] make vary object that can fulfil a request.
- [ ] Advantages inherited from Composition **extend**
- [ ] Advantages inherited from Interface **extend**

Drawbacks

- [ ] Receipt is not guaranteed **similarity**
- [ ] Drawbacks inherited from Composition **extend**
- [ ] Drawbacks inherited from Interface **extend**



Decorator

Purpose

Functionality add responsabilities

Result

delivers a flexible extending mechanism

Variation

make vary object responsabilities

Mean

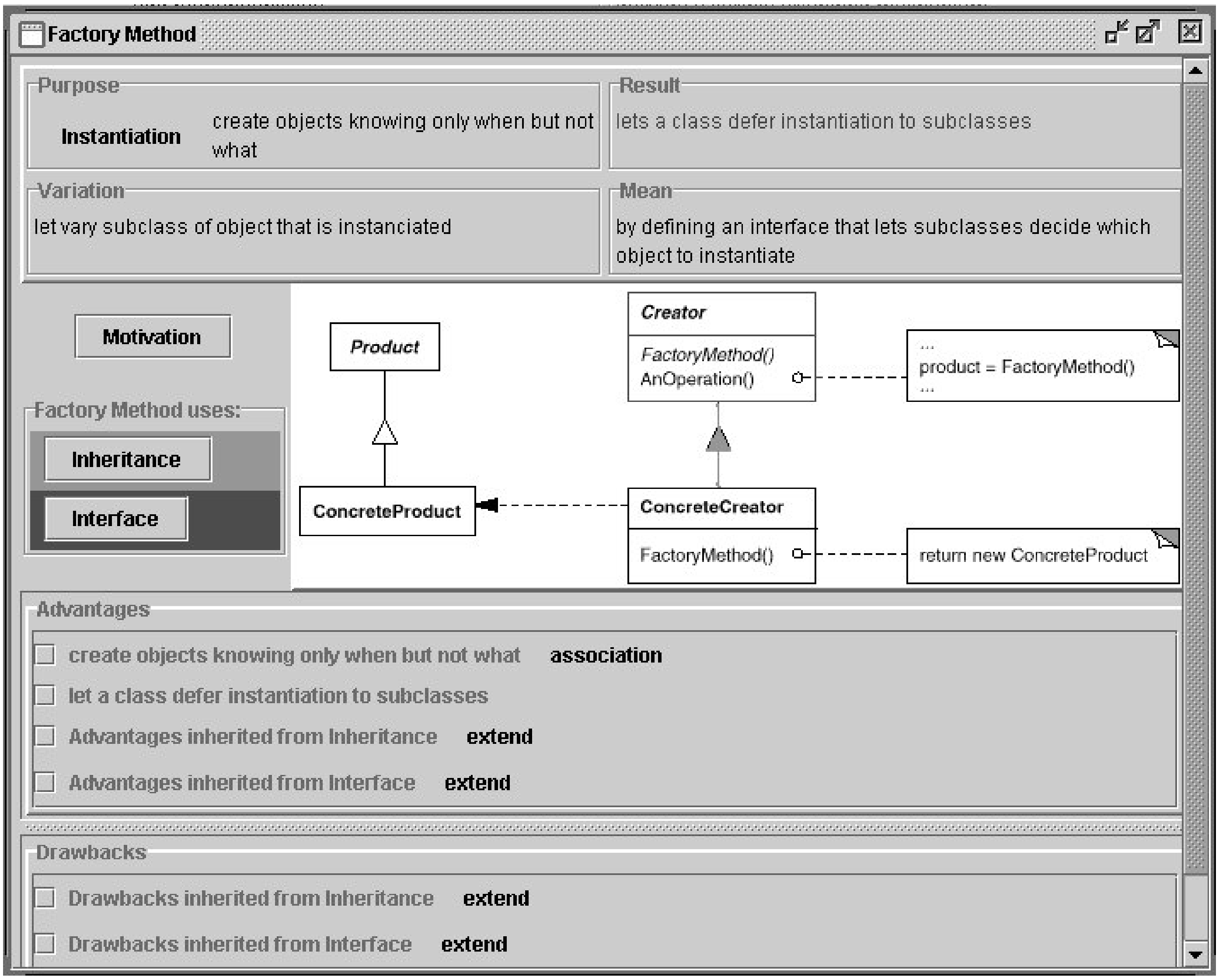
by forwarding requests to the composant

Advantages

- ☐ change skin of an object and not the guts **alternative**
- ☐ add responsabilities **similarity** **association**
- ☐ make vary object responsabilities
- ☐ Advantages inherited from Composition **extend**
- ☐ Advantages inherited from Interface **extend**

Drawbacks

- ☐ Drawbacks inherited from Composition **extend**
- ☐ Drawbacks inherited from Interface **extend**


Factory Method

Purpose
Instantiation create objects knowing only when but not what

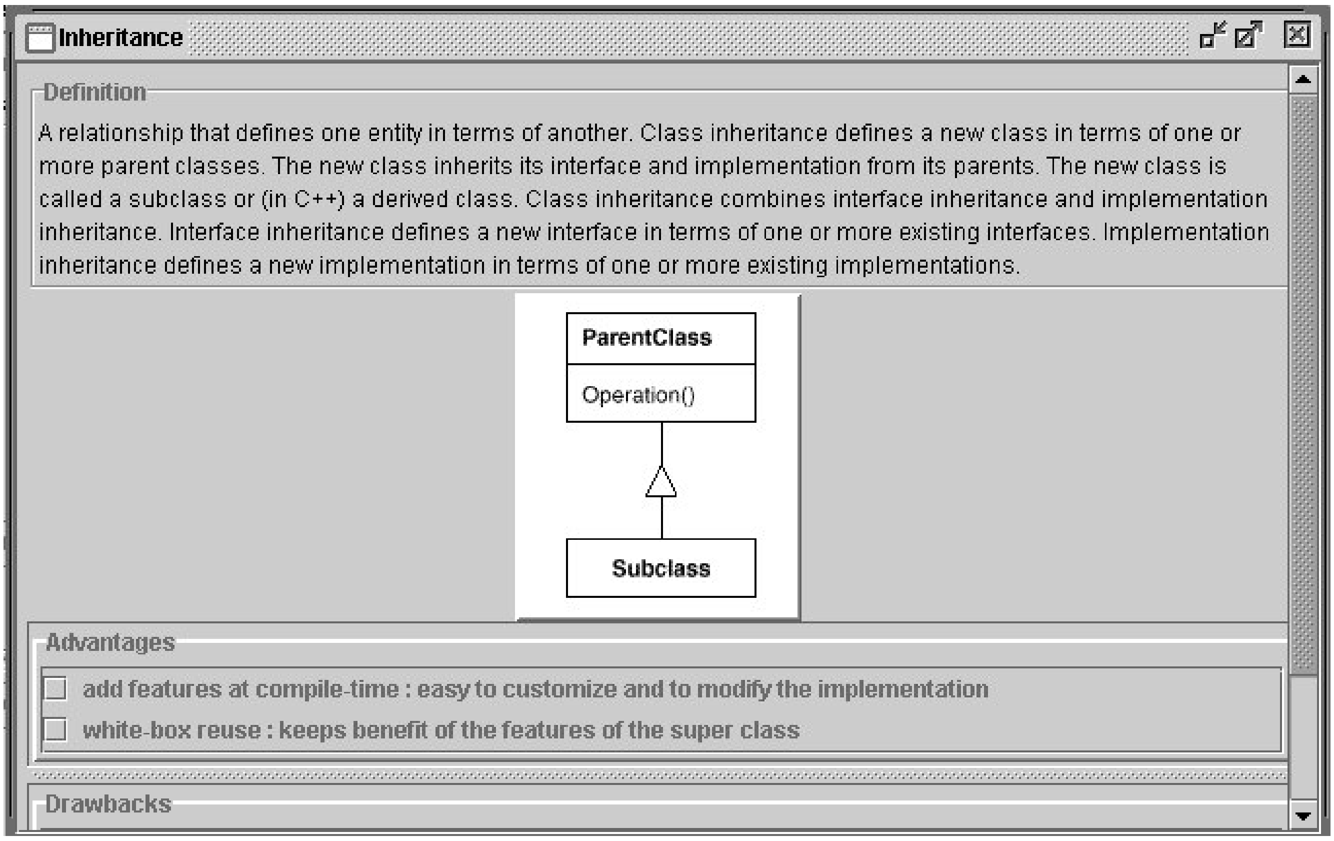
Result
lets a class defer instantiation to subclasses

Variation
let vary subclass of object that is instanciated

Mean
by defining an interface that lets subclasses decide which object to instantiate

Advantages
- ☐ create objects knowing only when but not what **association**
- ☐ let a class defer instantiation to subclasses
- ☐ Advantages inherited from Inheritance **extend**
- ☐ Advantages inherited from Interface **extend**

Drawbacks
- ☐ Drawbacks inherited from Inheritance **extend**
- ☐ Drawbacks inherited from Interface **extend**



Inheritance

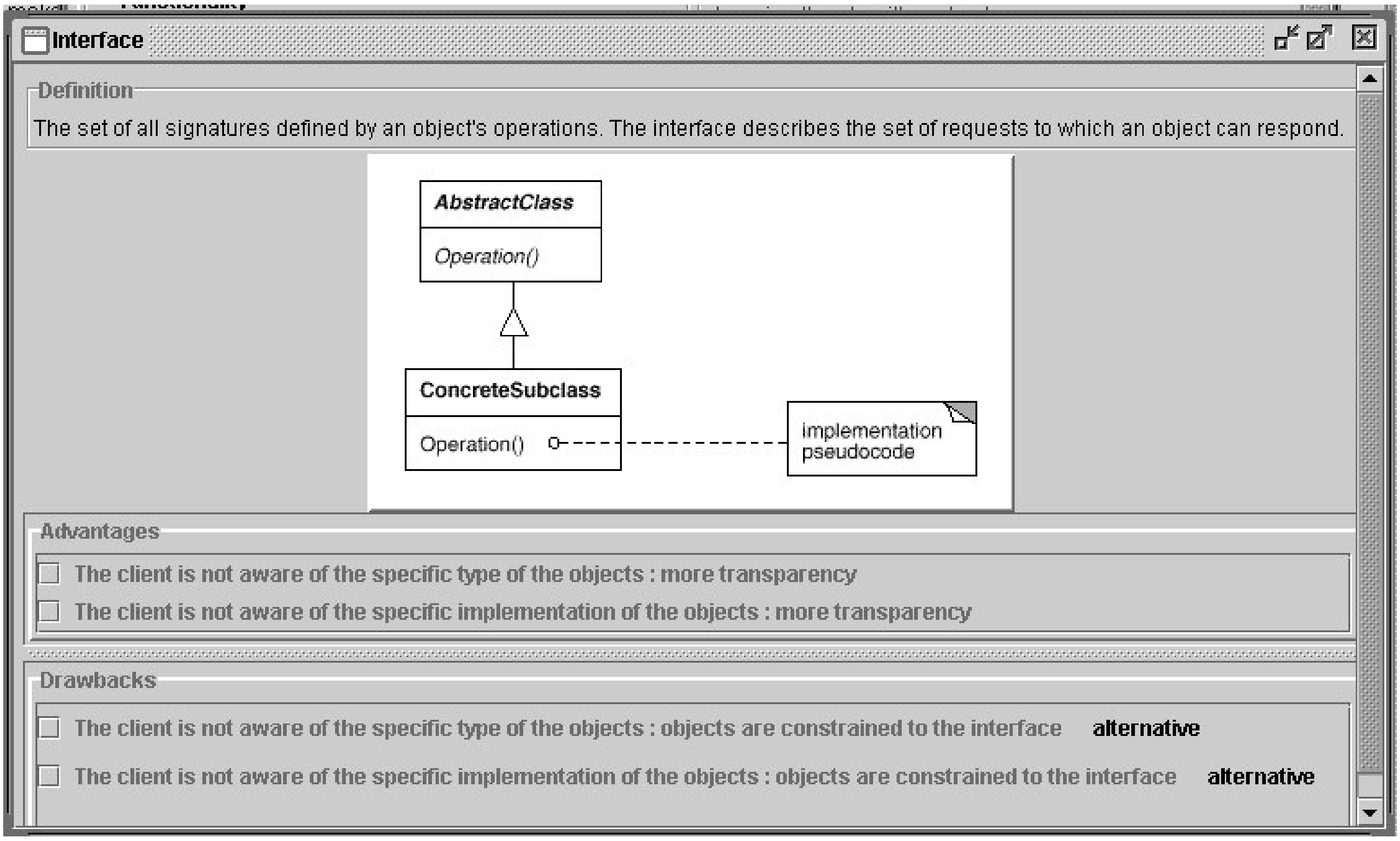
Definition

A relationship that defines one entity in terms of another. Class inheritance defines a new class in terms of one or more parent classes. The new class inherits its interface and implementation from its parents. The new class is called a subclass or (in C++) a derived class. Class inheritance combines interface inheritance and implementation inheritance. Interface inheritance defines a new interface in terms of one or more existing interfaces. Implementation inheritance defines a new implementation in terms of one or more existing implementations.

Advantages

- [ ] add features at compile-time : easy to customize and to modify the implementation
- [ ] white-box reuse : keeps benefit of the features of the super class

Drawbacks



Definition

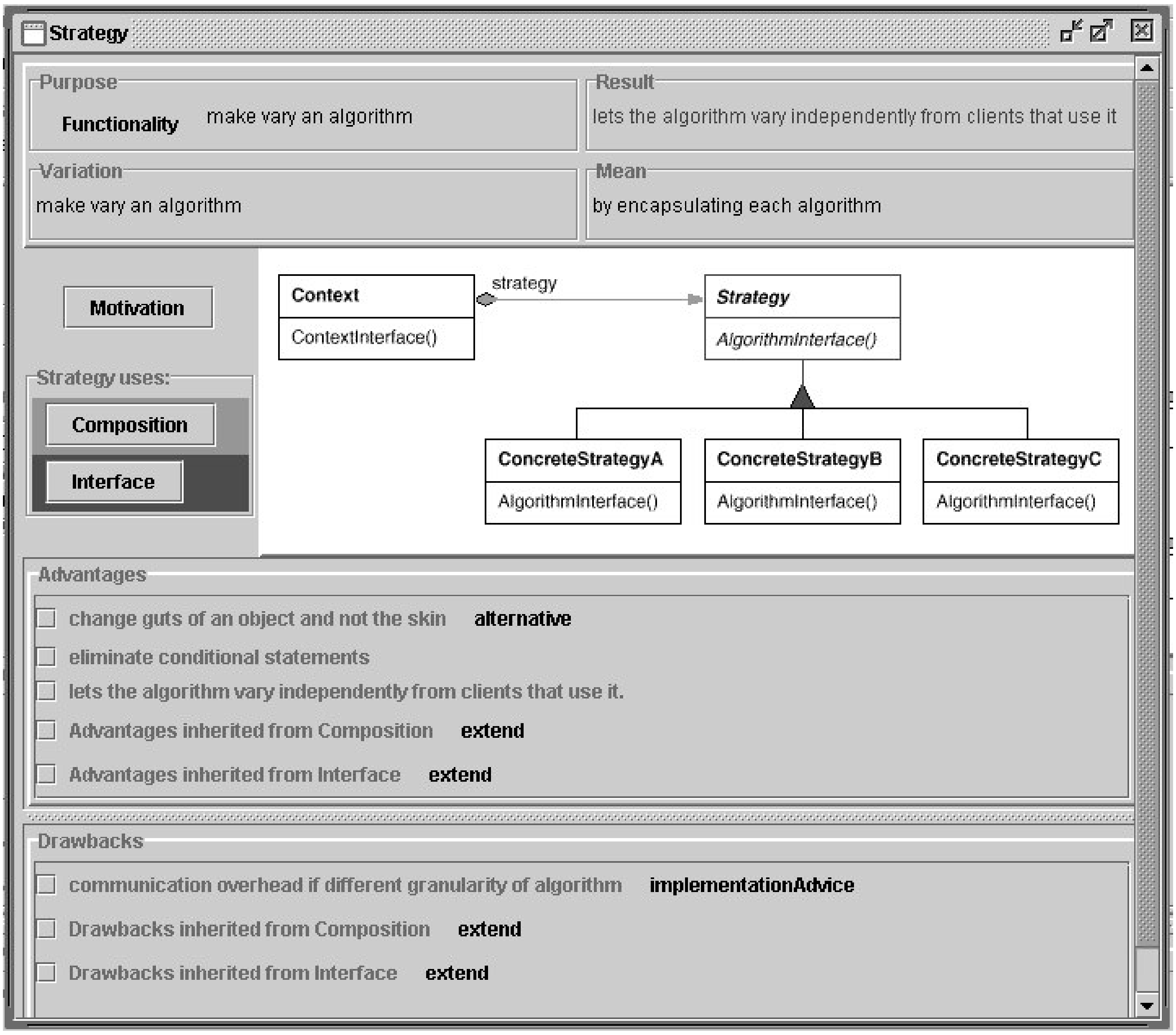
The set of all signatures defined by an object's operations. The interface describes the set of requests to which an object can respond.

Advantages

- ☐ The client is not aware of the specific type of the objects : more transparency
- ☐ The client is not aware of the specific implementation of the objects : more transparency

Drawbacks

- ☐ The client is not aware of the specific type of the objects : objects are constrained to the interface **alternative**
- ☐ The client is not aware of the specific implementation of the objects : objects are constrained to the interface **alternative**


Strategy

Purpose

Functionality make vary an algorithm

Result

lets the algorithm vary independently from clients that use it

Variation

make vary an algorithm

Mean

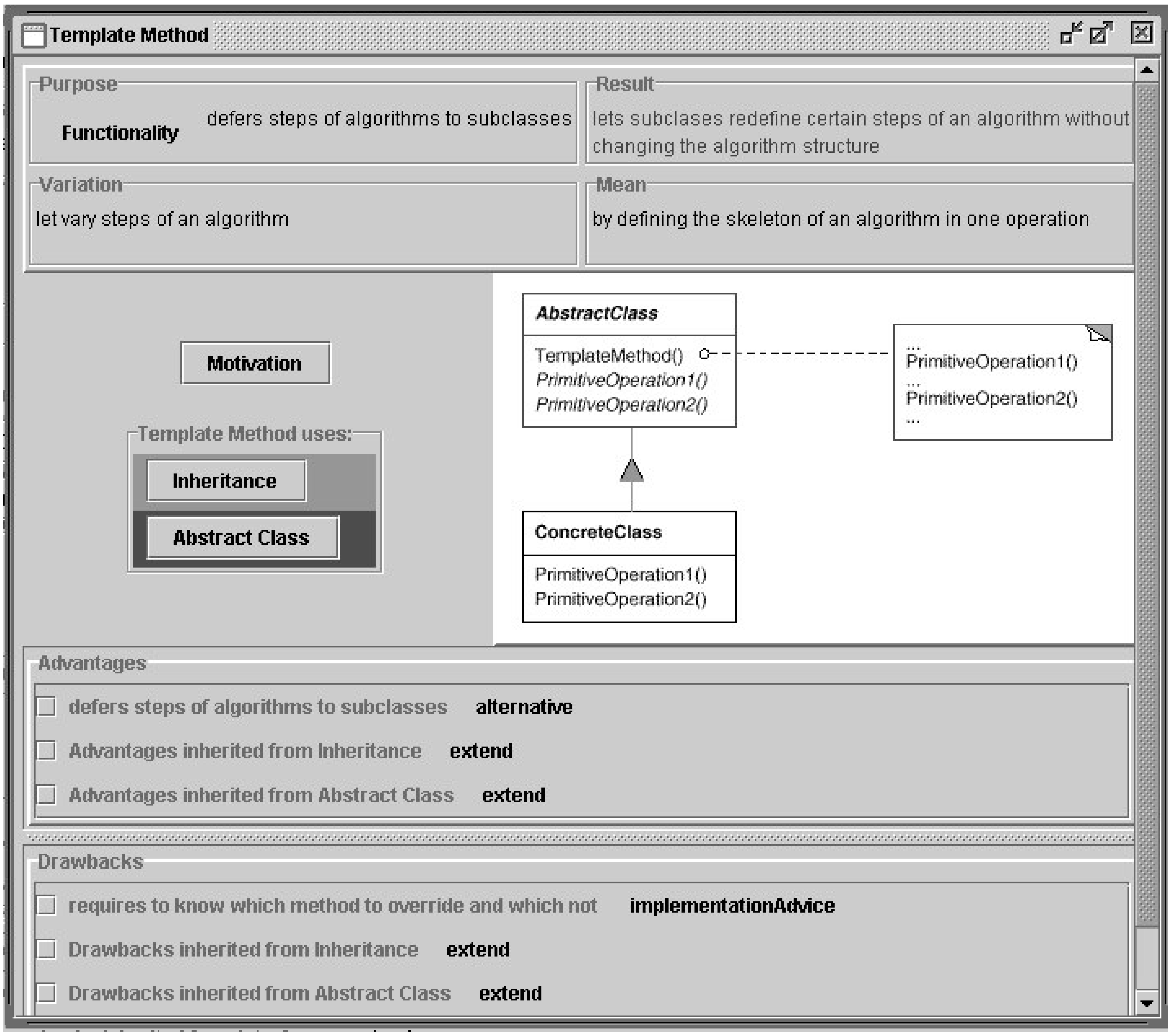
by encapsulating each algorithm

Advantages

- ☐ change guts of an object and not the skin **alternative**
- ☐ eliminate conditional statements
- ☐ lets the algorithm vary independently from clients that use it.
- ☐ Advantages inherited from Composition **extend**
- ☐ Advantages inherited from Interface **extend**

Drawbacks

- ☐ communication overhead if different granularity of algorithm **implementationAdvice**
- ☐ Drawbacks inherited from Composition **extend**
- ☐ Drawbacks inherited from Interface **extend**



Template Method

Purpose

Functionality: defers steps of algorithms to subclasses

Result: lets subclases redefine certain steps of an algorithm without changing the algorithm structure

Variation: let vary steps of an algorithm

Mean: by defining the skeleton of an algorithm in one operation

Advantages

- ☐ defers steps of algorithms to subclasses **alternative**
- ☐ Advantages inherited from Inheritance **extend**
- ☐ Advantages inherited from Abstract Class **extend**

Drawbacks

- ☐ requires to know which method to override and which not **implementationAdvice**
- ☐ Drawbacks inherited from Inheritance **extend**
- ☐ Drawbacks inherited from Abstract Class **extend**