

**École Nationale Supérieure des Techniques
Industrielles et des Mines de Nantes**

Computer Science Department

Research Report 03/3/INFO

Design Patterns Formalization

Aline Lúcia Baroni

**Yann-Gaël Guéhéneuc
Hervé Albin-Amiot**

Nantes, June 2003

CONTENTS

1. Introduction	1
2. The Need to Formalize Patterns.....	2
3. Existing Approaches and Their Limitations	6
3.1 The Inadequacy of Object Notations	6
3.2 Attempts At Precise Specification Languages.....	7
3.3 Tool Support	9
3.4 Related Formal Methods	10
4. LAYout Object Model	13
4.1 Problems of Implementing Design Patterns	13
4.2 Layered Object Model.....	14
4.3 Demonstration of <i>ADAPTER</i> pattern.....	16
4.4 Conclusions about LayOM.....	18
5. Extended Object Oriented Programming Languages Grammar	19
5.1 Language Support for Patterns.....	19
5.2 Demonstration with <i>DECORATOR</i> pattern	21
5.3 Conclusions about Extended Grammars	23
6. CONSTRAINT DIAGRAMS.....	24
6.1 Constraint Diagrams Notation.....	24
6.2 Three Layered Modeling.....	25
6.3 Demonstration of <i>ABSTRACT FACTORY</i> pattern	26
6.4 Conclusions about the model.....	28
7. The DisCo Method	29
7.1 Elements of Pattern Formalization.....	29
7.2 Demonstration of <i>OBSERVER</i> pattern.....	30
7.3 Conclusions about DisCo Method.....	32
8. LePUS: LanguagE for Patterns Uniform Specification.....	33
8.1 Textual Notation.....	34
8.2 Visual Notation.....	40
8.3 Demonstration of <i>STRATEGY</i> pattern	42
8.4 Demonstration of <i>FAÇADE</i> pattern	44
8.5 Conclusions about LePUS.....	45
9. Conclusions.....	53
10. References and Bibliography.....	54

FIGURES

<i>Figure 4.1 – Layered Object Model</i>	13
<i>Figure 4.2 – Class <code>TextEditWindow</code> Example</i>	15
<i>Figure 4.3 – Structure of <code>ADAPTER</code></i>	16
<i>Figure 4.4 – Class Adapter Example</i>	17
<i>Figure 4.5 – Example of a Declaration</i>	17
<i>Figure 5.1 – Simplified Base Grammar Interface</i>	19
<i>Figure 5.2 – Structure of <code>DECORATOR</code> According to GoF Catalog</i>	20
<i>Figure 5.3 – Using Attribute Comments to Define Roles in Patterns</i>	21
<i>Figure 5.4 – Roles in <code>DECORATOR</code> pattern</i>	22
<i>Figure 5.5 – Examples of Rules using the Extension Grammar</i>	23
<i>Figure 6.1 – Set Membership</i>	24
<i>Figure 6.2 – Set Membership: Each publication has a title</i>	24
<i>Figure 6.3 – UML Class Diagram + Abstract Instances</i>	25
<i>Figure 6.4 – Three-Model Layering</i>	25
<i>Figure 6.5 – Structure of <code>ABSTRACT FACTORY</code> according to GoF Catalog</i>	26
<i>Figure 6.6 – <code>ABSTRACT FACTORY</code> as a Type Model</i>	26
<i>Figure 6.7 – <code>ABSTRACT FACTORY</code> deployed as a Class Model</i>	27
<i>Figure 6.8 – <code>ABSTRACT FACTORY</code> as a Role Model</i>	27
<i>Figure 7.1 – An illustration of <code>OBSERVER</code> Pattern</i>	30
<i>Figure 8.1 – Basic Graphic Symbols of LePUS</i>	40
<i>Figure 8.2 – Superimposing a Function Symbol with a Class Symbol</i>	41
<i>Figure 8.3 – LePUS Diagram of <code>STRATEGY</code></i>	42
<i>Figure 8.4 – A Refinement of <code>STRATEGY</code></i>	43
<i>Figure 8.5 – LePUS formulae of <code>STRATEGY</code></i>	43
<i>Figure 8.6 – LePUS diagram of <code>FAÇADE</code></i>	44
<i>Figure 8.7 – LePUS formulae of <code>FAÇADE</code></i>	45

TABLES

<i>Table 1 – Specification of Each GoF pattern</i>	2
<i>Table 2 – Categories of Verbal Descriptions Inside GoFs' Catalog.....</i>	5
<i>Table 3 – Main features of existing specification languages.....</i>	11
<i>Table 4 - Primary ground relations and their intent.....</i>	35
<i>Table 5 - Model for the STATE sample code.....</i>	37

1. INTRODUCTION

Design patterns were introduced in software engineering as an effective mean of disseminating solutions to problems repeatedly encountered in object oriented programming [BEC87] and since their emergence, they have been widely accepted and adopted by software practitioners.

Design patterns contribution covers the definition, the design and the documentation of class libraries and frameworks, offering elegant and reusable solutions to design problems, and consequently increasing productivity and development quality. Each design pattern lets some aspects of the system structure vary independently of other aspects, thereby making the system more robust to a particular kind of change [GAM95].

The majority of publications in the pattern field focuses on *micro-architectures*; i.e., intentionally abstract description of generic aspects of software systems. Despite this abstractness, the academic community recognizes that a better understanding of design patterns by means of systematic investigation is essential. Reflective tasks in this direction include comparative analyses of design patterns, proposals for precise means of specification, attempts for tools, analysis of relationships among patterns, and other discussions.

However, few works offer methods of precise specification of design patterns, resulting in lack of formalism. In this sense, patterns remain empirical and manually applied. According to [BRÖ00], manual application is tedious and error prone. Precise specification can improve the application of design patterns as well as the analysis of relationships among them and tools in support of their application.

Very little progress has been made towards better understanding of the micro-architectures dictated by design patterns [EDE00]. This report tries to capture the "essence" of patterns, showing the importance of researches able to illuminate how design patterns are essentially structured.

In this report, we present a detailed study of design patterns specification. The document is organized as follows:

- Section 2 details the need to formalize design patterns, emphasizing their importance.
- Section 3 illustrates some of the existing methods to formalize a design patterns, mentioning the aspects according to which they can be formalized. Some tools to support design patterns application are presented. To conclude, a brief comparison of the methods is included.
- Sections 4, 5, 6 and 7 present some approaches to specify "the essence" of design patterns, and their limitations. In these sections, the most distinctive formalisms are explained and validation of patterns is showed, through examples.
- Finally the conclusion and references take part in sections 8 and 9.

For the sake of simplicity, in the rest of this report, the term pattern is used to refer to design patterns.

2. THE NEED TO FORMALIZE PATTERNS

Software patterns, and in particular design patterns, are published mostly within collections or catalogues: assembled works of generally independent content grouped according to some categorization.

Other than the categorization under domain applicability, very little effort was made to further refine the classification of patterns. Effort is largely focused on contributing to the existing bulk of design patterns.

The most influential publication of design patterns is the catalogue present in [GAM95], known as *Gang of Four – GoF – Catalogue*, which lists 23 patterns. Each one is formatted along a fixed structure that includes:

- | |
|---|
| <ol style="list-style-type: none">1. <i>Pattern Name and Classification</i>2. <i>Intent</i>3. <i>Also Known As</i>4. <i>Motivation</i>5. <i>Applicability</i>6. <i>Structure</i>7. <i>Participants</i>8. <i>Collaborations</i>9. <i>Consequences</i>10. <i>Implementation</i>11. <i>Sample Code</i>12. <i>Known Uses</i>13. <i>Related Patterns</i> |
|---|

Table 1 – *Specification of Each GoF pattern*

All the specification, except for the sections *Structure* and *Sample Code*, consists of statements enunciated in natural language (English). The *Structure* description is done through OMT diagrams [BLA91] and the *Sample Code* contains code in C++ or Smalltalk languages.

[EDE00] claims that these means of specification are inadequate mainly because natural language is inaccurate and vague – verbal specifications cannot resolve ambiguities in a definitive manner – while source code and diagrams depict a particular instance of a pattern rather than one general rule. Due to the lack of appropriate means of specification, *clients* of a design pattern are compelled to project from the given examples or to interpret the designation of verbal description that should apply to their context of interest.

To what extent can such specifications be understood precisely? Does a definite interpretation of the patterns exist at all? How much of the verbal specification indeed have unambiguous interpretation? These ways of specification are not sufficiently rigorous to allow unambiguous interpretation. Equivocal specifications are an obstacle in devising tools that support or semi-automate the application of design patterns in programs, because machines cannot perform without a definite specification of pertinent objectives.

Unambiguous or formal specifications for design patterns are widely recognized as important goal to achieve and may contribute to:

1. Resolve questions of relationships between patterns, mentioning if one pattern is a special case, a variation, a component of another pattern and so forth (e.g., is a

- particular program an instance of pattern $p1$ or of another pattern, $p2$? Or is $p1$ a special case of $p2$?).
2. Resolve the question of *validation*, which is checking, in a given sequence of concrete program, if a micro-architecture conforms to the specification of a particular pattern.
 3. Allow tool support in the activities related to patterns.

Furthermore, in the absence of a suitable, dedicated pattern specification language, design patterns were never entirely codified, and no attempt towards putting order to their interactions has ever gone beyond “this pattern *uses* that pattern”, or “these two patterns often *occur simultaneously*” [EDE98].

Notwithstanding, there are several objections to formalisms, which are voiced along the following lines:

1. Patterns are recipes that describe solutions to a broad category of problems. Pairing a solution with the relevant problems is an inherent benefit of using patterns. Focusing only on the solutions loses this benefit.
2. Formal specifications contribute little or nothing to the understanding when and how to use a pattern. “Formalizing the solution makes it harder to grasp the key ideas of the pattern... Programmers need concrete information that they can understand, not an impressive formula.” [BUS96].
3. Patterns are abstractions, or generalizations, and therefore are meant to be vague, ambiguous and imprecise. If they were specified in a precise form, or expressed in mathematical terms, they would no longer be patterns [BUS96].
4. There is no fixed element in patterns, and everything can be changed about them. In other words, if “the basic structure is fixed... this isn’t patterns any more.” [COP96].
5. Patterns are quasi-corporeal concepts whose essence is intangible, elusive and hence beyond the scope of a literal expression. A *good* pattern departs from mere micro-architectural prescription by some immaterial quality that cannot be explicitly expressed, a *quality without name*, and therefore cannot be interpreted outside its context or taken apart.

A critical review of these objections was published in *Giving ‘The Quality’ a Name* [EDE98] and [EDE00]. Eden’s work argues that it is possible to formalize patterns, and defeats these drawbacks, respectively, as follows:

1. Precise specification of the *solution* segment alone does not indicate that the *problem* specification is unimportant. Specifying a solution does not diminish the significance of the *problem* and other elements in the structure of patterns.
2. It is primarily a matter of opinion, or personal intuition, whether formalisms are important or not. Nevertheless, ambiguous descriptions are an obstacle in resolving details of implementation, which require complete and accurate understanding of the solutions.
3. It is wrongly assumed that exact (or formal) specifications can only describe concrete entities, and in order to be general one has to be vague. A specification can be precise and general at the same time.
4. If the statement 4 above is true, there is no way patterns can be well defined or understood.

5. Many members of the pattern community express this opinion repeatedly. Unfortunately, it is usually carried as an oral tradition or stated in informal occasions, such as mailing lists. As such, it cannot be properly quoted.

Because of the lack of a dedicated specification language, design patterns are invariably communicated through a list of prototypical instances, source code or simplified implementations, and classes – or instances – diagrams thereof. The pattern community can profit of formal means of reasoning on design patterns and on relationships among them.

Precise specification languages that evolved from the GoF catalogue are directed at restating information included in the *Structure*, *Participants* and *Collaborations* sections, while the remaining elements, such as the ones that appear under the sections *Intent*, *Related Patterns*, and *Consequences*, are overlooked.

Studying the GoF, [EDE98] proved that it is possible to categorize patterns. The study carried identified six categories of verbal descriptions with respect to precision and formality. The categories, quoting some examples, are reproduced below.

Interpretation Category	Examples (extracts from [GAM95])
1. Precise, singular	<ul style="list-style-type: none"> • <i>DECORATOR</i>: “maintains a reference to a Component object” • <i>VISITOR</i>’s ConcreteElement: “implements an Accept operation that takes a visitor as an argument”
2. Enumerated alternatives	<ul style="list-style-type: none"> • <i>FACTORY METHOD</i>’s Creator: “may call the factory method to create a Product object” • <i>STRATEGY</i>, Collaborations: “Alternatively, the context can pass itself as an argument to Strategy operations” • <i>DECORATOR</i>, Collaborations: “ It may optionally perform additional operations before and after forwarding the operations”
3. Precise generalization	<ul style="list-style-type: none"> • <i>VISITOR</i>: “declares a Visit operation for each class of ConcreteElement in the object structure” • <i>DECORATOR</i>: “defines an interface that conforms to Component’s interface”
4. Technical terms, yet open to various interpretations	<ul style="list-style-type: none"> • <i>PROXY</i>: “Virtual proxies... <i>cache</i> additional information about the real subject so they can postpone accessing to it” • <i>PROTOTYPE</i>: “implements an operation for <i>cloning</i> itself” • <i>MEMENTO</i>: “stores <i>internal state</i> of the Originator object”
5. Fuzzy, informal or teleological description	<ul style="list-style-type: none"> • <i>ADAPTOR</i>’s Adaptee: “defines an existing interface that <i>needs adapting</i>” • <i>COMPOSITE</i>’s Component: “implements default behaviour for the interface common to all classes, as

	<p style="text-align: center;"><i>appropriate</i>"</p> <ul style="list-style-type: none"> • <i>OBSERVER</i>'s Collaborations: "ConcreteObserver uses this information to <i>reconcile</i> its state with that of the subject"
6. Deliberate omission of detail	<ul style="list-style-type: none"> • <i>STATE</i>: "The State pattern does not specify which participant defines the criteria for state transitions"

Table 2 – *Categories of Verbal Descriptions Inside GoFs' Catalogue*

The first three categories comprise the relatively *precise* statements made throughout the verbal descriptions. A verbal specification is considered precise if it has a *compact* set of interpretations in several *conventional* object-oriented programming languages. Analyses done by [EDE98] indicate that large part of the specifications of the GoF patterns are precise and their descriptions fall under categories 1, 2, and 3.

Considering the category 4, if infinite interpretations exist, the formalization efforts can focus on representing well-defined subsets of interpretations, and declare each as a separated pattern. Success to formalize depends on the discrimination of useful subsets of such terms within the limits of the specification language.

Sentences of category 5 were considered impossible to be rendered precise by *simple* means. Success in this context depends on the frequency of such terms. The less they occur the most chances appear to formalize patterns.

In category 6, details are omitted because patterns are abstractions or *generalizations* of implementations, thereby similar to individuals of category 3. It is important that the specification language delivers expressions that accurately and correctly account for such generalizations.

The actual frequency of teleological specifications, category 5, is kept low in the GoF catalogue and there is a domination of statements of the first three categories, specifications can indeed be reliably translated to some precise form while preserving their essential characteristics.

3. EXISTING APPROACHES AND THEIR LIMITATIONS

This section reviews proposals of specification languages for design patterns and publications that describe tools to support pattern application. First, the inadequacy of object notations is showed. Then, attempts at precise specification languages and tool support are mentioned. Finally, a comparison of design pattern related formal methods is given.

3.1 The Inadequacy of Object Notations

There is a difference between a design process and a design pattern:

- A *design process*, typically taking place during the object-oriented design stage of software construction, results in a concrete system, whose description comprises classes, instances, and relations among them, intended toward a solution of a specific problem.
- A *design pattern* reflects a generic *aspect* rather than a particular system. An unbounded number of concrete systems or programs may conform to a single design pattern.

In other words, programs most often constitute *instances* of design patterns plus additional elements that do not conform to common patterns.

Object notations, such as Coad & Yourdon [COA91], Shlaer & Mellor [MEL92], Booch [BOO94], Rumbaugh *et al.* [BLA91], UML [BOO99] and others were created to facilitate the object-oriented design process and to report its results, thereby delineating a concrete software system. No object notation was ever meant to account for *sets* of programs, as the specification of design patterns requires. Notwithstanding, these notations were never granted with precise semantics [EDE00].

For these reasons, OMT diagrams (as used to describe patterns) do not incorporate variable symbols or sets of any kind and each diagram may account for, at most, a particular instance of a pattern. It cannot specify which modifications to the instance depicted preserve the *identity* of the pattern of interest and which violate it. Furthermore, OMT diagrams invariably fail to capture significant information about the implementation of a pattern. The missing information is conveyed using informal cues, sample implementations, and mainly using elaborated English narrative.

A possibility arises to apply UML as a *meta-language* rather than as a language for concrete programs. One may have a diagram similar to the UML notation, that describes how instances of the meta-classes Class, Method, Variable, etc. are related to each other, therefore specifying a pattern through its participants, generically.

However, UML, as any other object notation, allows only a predetermined, fixed number of associations, all of which are typically defined between classes. Besides, a small number of “properties” may relate to both classes and methods (such as *abstract* or *return-type*). To work with patterns, on the other hand, it is required to abstract the additional, typical associations that may exist between constructs in an object-oriented program. These include relations between methods and other methods and between methods and classes as well.

Finally, object notations cannot express sets of higher order (for example, sets of sets of classes), which are fundamental to the specification of design patterns. In particular, object notations do not account for morphisms and correlations among sets (such as 1:1 and onto correlations among sets of any order). Thus, none of the UML-like *associations* and *cardinalities* is sufficiently expressive even as part of a meta-language [EDE00].

3.2 Attempts At Precise Specification Languages

In this part, the attempts to precise specification of patterns are briefly discussed. The main works are detailed below.

- LayOM

The Layout Object Model [BOS95] extends the classic object model with the concepts of *states* and *layers* to support the specification of patterns. The expressiveness of Bosch's specification language is demonstrated by phrasing constraints on the behaviour and protocols of interactions of the instances of the class's part of design pattern structure.

- Abstract Data Views and Abstract Data Objects

Alencar, Cowan and Lucena proposed an environment that comprises *Abstract Data Views* and *Abstract Data Objects* (ADV/ADO) [ALE96], specified in a specialized "scheme" language. Problems arise because the mapping of the fundamental constructs of the ADV/ADO model onto those of common Object Oriented Programming Languages is very elaborated and not self-sufficient. In addition, no sufficient evidences are provided for the expressiveness of their language.

- Contracts

Helm, Holland and Gangopadhyay [HEL90] defined *Contracts*, an extension to first order logic with representations for function calls, assignments, and an ordering relation among them. The *behavioural compositions* described in their publication do not address relationships among classes that are a fundamental part of design patterns, such as inheritance, various creation and forwarding relations, and sets of entities. The contracts formalism was created before design patterns, and maybe for this reason it is not appropriate to express them.

- Extended OOPLs Grammar

Hedin [HED97] proposes to extend the base grammar of a given OOP language - specified as a set of parsing rules – with attributes that express various programming conventions, and that are in particular useful for specifying the roles of collaborators in design patterns. In this approach, the elements of the pattern are tightly coupled with the grammatical rules defining the base language. A drawback of this approach is the low abstractness and low expressiveness of the "specification language", which can hardly serve to clarify disputes about design patterns.

- Constraints Design Language (CDL)

Similarly to Hedin's work, Klarlund, Koistinen, and Schwartzbach offer a specialized language of parse trees – *CDL* – that can be used to validate design constraints [KLA96]. CDL formulae are proposed as means to express constraints over the behaviour and relationships of collaborators in a design pattern. Therefore, CDL cannot help in resolving the issues that concern the design patterns users community, as observed by the authors of the methodology. While a design pattern is intended to propose a solution in a limited context, CDL is intended to enforce certain design invariants on a complete system. Besides, CDL is defined relative to a particular formal design or programming language.

- Constraint Diagrams

Lauder and Kent [KEN98] employed a set-theory-based notation designated as *constraint diagrams* to specify the relations between classes and instances. It can formalize patterns according to *Participants*, *Collaborations* and *Structure*. Whereas class diagrams are only able to show that there are relationships among certain kinds of object, constraint diagrams are able to visualize properties as well as compositions of those relationships [GIL98]. It is not clear, however, whether constraint diagrams may account for the kind of functional relationships among sets that commonly occur within patterns. Additionally, the abstraction level of constraint diagrams is low (i.e., they are too detailed) for the specification of highly complex design patterns.

- DisCo

DisCo is a way to formalize temporal behaviours of design patterns, paying special attention to their natural utilization when composing specifications of complex systems [MIK98]. The language used for composing specifications is textual and is based on the *Participants* and *Collaborations* of patterns.

- LePUS

LePUS is a formal notation dedicated to the specification of object-oriented design and architecture sufficiently abstract to represent patterns. A LePUS specification can be expressed either as a formula or as a semantically equivalent visual diagram. Both are well defined, concise and expressive. Expressions in LePUS specify patterns in terms of constraints on the properties of programs. Specifications of patterns in LePUS consist of two main parts: a representation of the *Participants*, and, constraints expressing the relationship that must (or must not) take place among the participants, reflecting their required *Collaborations*.

- Patterns Detection Language (PDL)

Albin-Amiot and Guéhéneuc [ALB01a] proposed a set of tools to use patterns in a round-trip fashion. First, they defined a meta-model to describe patterns, oriented towards patterns instantiation and detection. The meta-model can help formalizing patterns according to *Participants*, *Collaborations* and *Structure*. Then, they developed a source-to-source transformation engine to modify the user source code, making it comply with the patterns descriptions. So, in this approach, they use also the *Implementation* and *Sample Code* of the GoF, although only the *Implementation* can be formalized. Later on, they use an explanation-based constraint solver to detect patterns in the source code from their descriptions.

It's possible to notice that many different mechanisms exist, leading to the wrong idea that is easy to specify patterns. In spite of great effort from the authors, many formal notations cannot capture all the concepts related to patterns and all of them have some drawbacks. Regarding the aspects to which patterns can be formalize, it's feasible to use the *Participants*, *Collaborations*, *Structure*, *Implementation* while nothing is done considering other aspects like *Intent*, *Also Know As*, and so on.

3.3 Tool Support

- Darwin-E environment

Pal [PAL95] demonstrates how the behaviour indicated by a pattern can be enforced by *Darwin-E* environment [MIN94], which detects violation of the specifications by means of static analysis.

- Constraints Design Language

Klarlund, Koistinen, and Schwartzbach [KLA96] present a specialized language of parse trees, *CDL*, which can be used to validate design constraints. However, both attempts (the Darwin-E Environment and the CDL) are of a very limited extent regarding their application to design patterns, and demonstrate very few examples to support their usefulness in the understanding of design patterns.

- COGENT

Budinsky, Finnie, Vlissides, and Yu [BUD96] present a tool that supports the application of design patterns by generating code. The programmer can specify what statements of the target programming language need to be created using a special purpose, ad-hoc scripting language: COGENT, whose imperatives produce statements in the target object-oriented programming language. This code generation tool can serve purposes other than implementations of design patterns, and this generality is also one of its shortcomings, as the number of statements required to create a simple code structure is unreasonably large for most tasks. An even more serious deficiency of this tool is the lack of reengineering, that is, the ability to integrate design patterns with existing classes. In other words, this approach assumes that patterns involve classes or routines dedicated to its role as a collaborator within a particular design pattern. As a rule, this assumption is untrue: patterns rarely exist in isolation. Often it happens that a *Collaborator* in one pattern plays a role in another.

- C++ Code Generator

Similarly, Quintessoft Inc. [QUI97] distributes a CASE tool that can generate C++ source code following one of the GoF patterns after being supplied by a number of parameters by the programmer. In both cases, the code generated by this tool and the previous one do not relate directly to pre-existing constructs of class libraries, and thus may not modify, adjust their behaviour, or even derive information and act accordingly.

- Patterns Fragments

One attempt to integrate existing code with patterns is presented by Florijn, Meijers, and van Winsen [FLO97]. They present a prototype tool that performs reengineering to allow the programmer to attach (possibly multiple) roles, designated *pattern fragments*, to existing elements (classes, relations). A pattern is represented as a tree whose leaves are participants labelled according to their roles. By this approach, pattern's roles are mere strings, which restrict the reasoning that can be made on such a model. The authors do not deliver details on the tool implementation.

Tools, as well as the notations, are still under development, and much work needs to be done to take profit of patterns using case tools. In particular, it is necessary to create tools based on formalisms that capture all the rules established by design patterns. Furthermore, it is also necessary to create tools that supports code generation and at the same time can be adapted “easily” to the user code. Automating these tasks is a very challenging issue for the design patterns community.

3.4 Related Formal Methods

How should pattern languages be formalized? One possible direction is by using a formal specification language such as Z or Larch to capture the instance/class relationships and dependencies.

However, this alone is not sufficient: formal specification languages are powerful in describing the external characteristics of any particular system without specifying any implementation details, but a design pattern is in many cases an implementation strategy that captures a specific solution, and the process of formulating it directly refers to the steps that are taken in its implementation.

Formal specification languages were not designed to express implementation details. An even more serious limitation of specification languages is that they are usually not powerful enough to describe a generic family of systems, as it is required in the case design patterns.

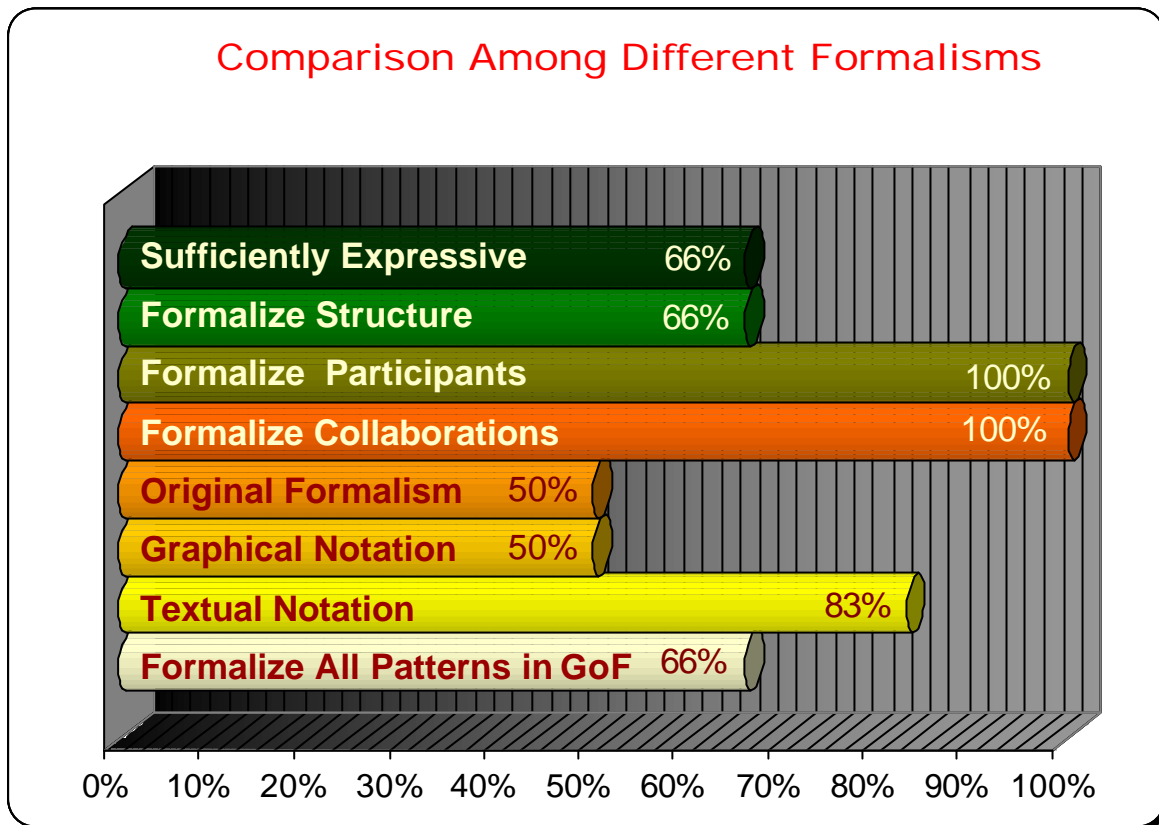
Table 3 puts together some characteristics of the specification methods described in the following sections. The choice of the explained formalisms was done due to the number of *good* criteria supported by the method, according to the table that follows. In other words, other formalisms do not attend *good* criteria or have some significant drawback(s).

Characteristics	LayOM	Extended Grammar	Constraint Diagrams	DisCo Method	LePUS	PDL
Ability to address <i>simple</i> relations among classes, such as inheritance, creation and forwarding relations.
Ability to deal with sets of entities.			
Capacity to formalize all the patterns in GoF catalogue, and possibly new ones	..	⚡
Textual formalism provided by the method
Graphical notation provided by the method		
Original formalism (not extended from other ones)			
Capacity to formalize patterns according to <i>collaborations</i>
Capacity to formalize patterns according to <i>participants</i>
Capacity to formalize patterns according to <i>Structure</i>
Capacity to formalize patterns according to other aspects					⚡	..
Sufficiently expressive	..	⚡	..	⚡

Table 3 – Main features of existing specification languages

- ◆ Indicates that the formalism attends/has the characteristic
- ⚡ Indicates that the sources of information don not provide a clear answer

The analysis of the previous table can lead to the following statistics:



Graphic 1 – Comparison Among Different Formalisms

Graphic 1 illustrates that all the studied formalisms are able to formalize the sections *Participants* and *Collaborations* of the *GOF catalogue*, while only 66 percent can deal with the section *Structure*. Due to the ability of formalizing all the GOF patterns and also to other characteristics mentioned in the next sections, four methods were considered sufficiently expressive, namely LayOM, Constraint Diagrams, LePUS and PDL.

In addition, only 3 (50%) are original formalisms, which means they are not originated from older ones. One could think these 3 new formalisms are the ones that include a graphical notation, but this is not completely true. The graphic tells that 66% of the formalisms do not present any graphical notation and by taking a look at table 3 it is possible to observe that only 2 original formalisms (representing 33%) created their own graphical notation. The graphic notation is important because it can help understanding the formalism, and it is easier to grasp.

4. LAYOUT OBJECT MODEL

LayOM [BOS96a], [BOS96b] provides language support for the explicit representation of design patterns in the programming language. It is an extended object-oriented language containing several components that are not part of the conventional object model, such as *states*, *categories* and *layers*.

The disadvantage of a programming language based on the conventional object-oriented paradigm such as C++ is that no support for the representation of design patterns is provided by the language. This leads to problems related to *traceability*, the *self-problem*, expressiveness and implementation overhead problems related to the implementation of design patterns.

The layered object model (LayOM) is proposed as a language model with explicit support for representing design patterns. Layers (see figure 4.1) are used to represent design patterns at the level of the programming language. They encapsulate the object and intercept messages that are sent to and by the objects. The layers are organized into classes and each layer class represents a concept, such as a relation with another object or a design pattern.

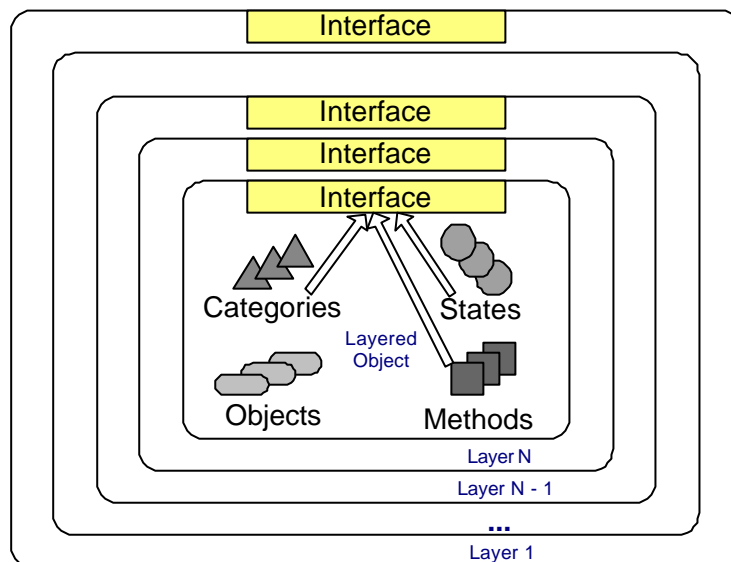


Figure 4.1 – Layered Object Model

4.1 Problems of Implementing Design Patterns

When implementing design patterns using a conventional object-oriented language, e.g. C++, one can identify several problems. One underlying problem is that the design pattern cannot be represented as a first-class entity. Since a design pattern often affects multiple aspects, e.g. methods, of a class, or even multiple classes or objects, a pattern language construct often cannot be inherited or composed in the traditional way. More powerful composition techniques are required that allow the design pattern to superimpose its behaviour on a class or object, while both the pattern and the domain entity remain identifiable entities.

Other experienced problems, when implementing the design patterns in a traditional object-oriented language, are:

- Traceability

The *traceability* of a design pattern is often lost because the programming language does not support a corresponding concept. The software engineer is thus required to implement the pattern as distributed methods and message exchanges, i.e., the pattern which is a conceptual entity at the design level is scattered over different parts of an object or even multiple objects. This problem has been identified by [SOU95].

- Self Problem

The implementation of several design patterns requires forwarding of messages from an object receiving a message to an object implementing the behaviour that is to be executed in response to the message. The receiving object can, for example, be an application domain object which delegates some messages to a strategy object. However, once the message is forwarded, the reference to the object originally receiving the message is no longer available and references to self refer to the delegated object, rather than to the original receiver of the message. The problem is known as the *self-problem* [LIE86].

- Reusability

Design patterns are primarily presented as design structures. Since design patterns often cover several parts of an object, or even multiple objects, patterns have no first class representation at the implementation level. The implementation of a design pattern can therefore not be reused and, although its design is reused, the software engineer is forced to implement the pattern over and over again.

- Implementation Overhead

The implementation overhead problem is due to the fact that the software engineers, when implementing a design pattern, often has to implement several methods with only trivial behaviour, e.g. forwarding a message to another object. This leads to significant overhead for the software engineer and decreased understandability of the resulting code.

To address these problems, a solution within the context of the layered object model, discussed in section 4.2, is proposed. The layered object model is an extensible object model and so-called layers encapsulate its objects.

4.2 Layered Object Model

A LayOM object contains, as any object model, instance variables and methods. The semantics of these components is very similar to the conventional object model. The only difference is that instance variables can have encapsulating layers adding functionality to the instance variable. In figure 4.2, an example LayOM class `TextEditWindow` is shown, containing one instance variable `loc`.

A *state* in LayOM is an abstraction of the internal state of the object. In LayOM, the internal state of an object is referred to as the *concrete state*. Based on the object's concrete state, the software engineer can define an externally visible abstraction of the concrete state, referred to as the abstract state of an object. The abstract object state is generally simpler in both the number of dimensions, as well as in the domains of the state dimensions. In figure 4.2, the abstract state `distFromOrigin` is shown. It abstracts the location of the mouse and the window origin into a distance measure.

A *category* is an expression that defines a client category. A client category describes the discriminating characteristics of a subset of the possible clients that should be treated equally by the class. For example, the class in figure 4.2 defines a Programmer client category, restricting the use of the object to instances of class `Programmer` and its sub-classes.

The behavioural layer types use categories to determine whether the sender of a message is a member of a client category. If the sender is a member, the message is subject to the semantics of the specification of the behavioural layer type instance.

A layer encapsulates the object and intercepts messages. It can perform all kinds of behaviour, either in response to a message or otherwise. Previously, layers have primarily been used to represent relations between objects. In LayOM, relations have been classified into *structural relations*, *behavioural relations* and *application-domain relations*.

Structural relation types define the structure of a class and provide *reuse*. These relation types can be used to *extend* the functionality of a class. Inheritance and delegation are examples of structural relation types.

The second type of relations is the *behavioural relation* that is used to relate an object to its clients. Client objects use the functionality of the class and the class can define a behavioural relation with each client (or client category).

Behavioural relations restrict the behaviour of the class. For instance, some methods might be restricted to certain clients or in specific situations.

The third type of relations is *application domain relation*. Many domains have, next to reusable application domain classes, also application domain relation types that can be reused. For instance, the *controls* relation type is a very important type of relation in the domain of process control. For more information on relation types, refer to [BOS95] and [BOS96a].

```
class TextEditWindow
  layers
    rs : RestrictState(Programmer,
      accept all when distFromOrigin<100 otherwise reject);
    pin : PartialInherit(Window, *, (moveOrigin));
    po : PartOf(TextEditor);
  variables
    loc : Location;
  methods
    moveOrigin(newLoc : Location) returns Boolean
    begin
      loc := newLoc;
      self.updateWindow;
    end;
  states
    distFromOrigin returns Point
    begin return ((lox.x - self.origin.x).sqr +
      (lox.y - self.origin.y).sqr).sqrt; end;
  categories
    Programmer
    begin sender.subClassOf(Programmer); end;
end; // class TextEditWindow
```

Figure 4.2 – Class *TextEditWindow* example

The class in figure 4.2 has three layers. The `PartOf` layer defines an instance of `TextEditor` as a part of the class. The `PartialInherit` layer defines that class `TextEditWindow` inherits all methods from class `Window`, except method `moveOrigin`. The `RestrictState` layer restricts access to instances of class `Programmer` and its subclasses, but only if the distance between the mouse location and the window origin is less than 100 units.

Next to an extended object model, the layered object model is also an *extensible* object model, i.e., the software engineer can extend the object model with new components. LayOM can, for example, be extended with new layer types, but also with structural components, such as *events*. The notion of extensibility, which is a core feature of the object-oriented paradigm, has been applied to the object model itself.

4.3 Demonstration of ADAPTER pattern

The *ADAPTER* design pattern is used to convert the interface of a class into another interface that is expected by its clients. It allows classes to cooperate; what would be incompatible due to the differences in expected interfaces.

In a conventional object-oriented language, the *Adapter* is implemented as an object that forwards the calls, after adaptation, to the *Adaptee*, i.e. the adapted object. In figure 4.3, the structure of an *Adapter* for object adaptation as presented in GoF is shown. Class adaptation is not shown in this figure.

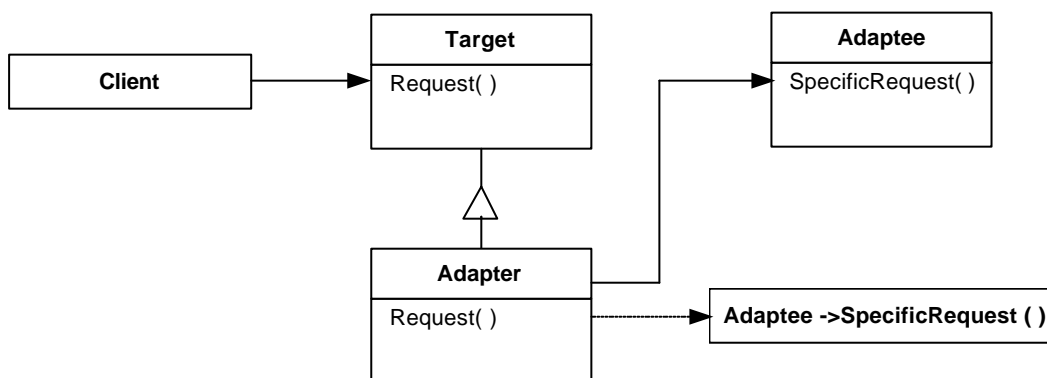


Figure 4.3 – Structure of ADAPTER

Although the adapter indeed allows classes to work together, which otherwise could not be possible, there are some disadvantages associated with the implementation of the pattern. One disadvantage is that for each element of the interface that needs to be adapted, the software engineer has to define a method that forwards the call to the actual method `SpecificRequest`. Moreover, in case of object adaptation, those requests that would not have required adaptation have to be forwarded as well, due to the intermediate adapter object. This leads to implementation overhead for the software engineer. It also suffers from the self-problem and lacks expressiveness. Because the behaviour of the *ADAPTER* pattern is mixed with the domain related behaviour of the class, traceability is reduced.

In the layered object model, the functionality of the *ADAPTER* design pattern does not require a separate object (or class) to be defined. Instead, a layer of type `Adapter` is

defined that provides the functionality associated with the design pattern. The layer can be used as part of a class definition, in which case it represents class adaptation.

It can also be defined for an object thus representing object adaptation. The syntax of layer type `Adapter` is the following:

```
<id> : Adapter(accept <mess-sel1> + as <new-mess-sel1>,
              accept <mess-sel2> + as <new-mess-sel2>, ...);
```

The semantics of the layer type is that a message with a message selector `<mess-sel>` that is specified in the layer is passed on with a new selector `<new-mess-sel>`. The `Adapter` layer type also allows more than one message selector to be translated to a new message selector. The layer will translate both messages send to the object encapsulated by the layer and messages send by the object.

The `Adapter` layer can be used for class adaptation by defining a new `adapter` class consisting only of two layers. Figure 4.4 shows an example class adapter.

```
class adapter
  layers
    adapt: Adapter(accept mess1 as newMessA,
                  accept mess2, mess3 as newMessB);
    inh: Inherit(Adaptee);
end; // class adapter
```

Figure 4.4 – *Class Adapter Example*

In the example `class adapter` translates a `mess1` message into a `newMessA` message and, a `mess2` or `mess3` message into a `newMessB` message. Class `Adaptee` presumably implements the methods `newMessA` and `newMessB` and the `Inherit` layer will redirect these and other messages to the instance of class `Adaptee` that is contained within the layer.

Adaptation at the object level can be achieved by encapsulating the object with an additional layer upon instantiation. In this case, the adaptation will only be effective for this particular instance and not for the other instances of the same class. Figure 4.5 presents an example of an adapted object declaration.

```
...
// object declaration
  adaptedAdaptee : Adaptee with layers
    adapt : Adapter(accept mess1 as newMessA,
                  accept mess2, mess3 as newMessB);
end;
...
```

Figure 4.5 – *Example of a declaration*

The instance `adaptedAdaptee` will be extended with an additional layer of type `ADAPTER` that adapts its interface to match the interface expected by its clients. The

`Adapter` layer will be the most outer layer of the object, intercepting all messages going into and out of the object.

Layer type `Adapter` can also be used in an inverted situation, i.e., a situation where a single client needs to access several server objects, but the client expects an interface different from the interface offered by the server objects. In this case, the client object (or its class) can be extended with an `Adapter` layer, translating messages send by the client into messages understood by the server objects.

The `Adapter` layer type allows the software engineer to translate the `ADAPTER` pattern directly into the implementation, without losing the pattern. There is a clear one-to-one relation between the design and the implementation. A second advantage is that the software engineer is not required to define a method for every method that needs to be adapted. The specification of the layer is all that is required. In addition, in case of object adaptation, the software engineer, in the traditional implementation approach, also needs to define a method for the methods of the adapted class that do not have to be adapted. When using the `Adapter` layer, this is avoided.

A disadvantage of the `ADAPTER` layer type definition is that the arguments of the message will be passed on as sent. In some situations, one would like to pass the arguments on in a different order or add or remove some arguments.

4.4 Conclusions about LayOM

The layered object model (LayOM) is an extended and extensible object model. It is *extended* because it contains *states*, *categories* and *layers* as additional components. It is an *extensible* object model because it can be extended with new components and new layer types.

A development environment that translates classes and applications defined in LayOM into C++ code supports the model. LayOM can be used without losing compatibility with legacy systems and code developed elsewhere as it is translated to C++. The generated C++ code can then be used to construct applications, either direct or integrated with existing C++ code (C++ code can be integrated in other code as any C++ program).

Thus, the software engineer using LayOM has the advantages of the extended expressiveness and avoids potential disadvantages as being limited to a particular language because the environment generates C++ code.

Another advantage of using LayOM instead of a traditional object-oriented language is that it does not suffer from the problems related to the implementation of design patterns. These problems can be categorized into the lack of traceability of design patterns in the implementation, the self-problem that several pattern implementations suffer from, the lack of reusability of design pattern implementations and the implementation overhead for the software engineer when implementing a design pattern.

5. EXTENDED OBJECT ORIENTED PROGRAMMING LANGUAGES GRAMMAR

Hedin [HED97] presents a technique based on attribute grammars to identify design patterns in the source code. He considers that when a design pattern is applied, this can be viewed as introducing a number of rules in the code, meaning that subsequent updates to the program should be done without breaking the rules. So, it is possible to think of a pattern application as a kind of language construct that identifies the objects playing the particular roles in the pattern and that specifies some rules that these objects must follow. This is the main idea of Hedin's work.

His paper deals only with *traceability* (the identification of patterns in the code) and *rule enforcement* (how to make sure the code adheres to the rules of a pattern).

5.1 Language support for patterns

Attribute extension is a technique for describing and enforcing programming conventions. The technique is based on attribute grammars, describing conventions by declarative semantic rules and it makes use of three kinds of specification:

- *A base grammar interface*, which is a context-free grammar for the base language, extended with functions for basic static-semantic information such as name bindings and type information. The functions may return references to other nodes in the syntax tree.
- *An extension grammar*, which is an attribute grammar describing the programming conventions, making use of the base grammar interface to avoid specifying basic information from scratch. It defines roles and rules of patterns.
- *Attribute comments*, which are special comments used to annotate a program. Technically, an attribute comment `/*= a =*/` defines a boolean attribute to have the value true.

The base grammar interface is specified in an object-oriented notation with a *node class* for each language construct. A simplified base grammar interface is shown below.

```

nodeclass Declaration ::= { }
nodeclass ClassDecl extends Declaration ::=
  (optional SuperClassId, ClassId, DeclList) {
    ClassDecl function superClassBinding()
  }
nodeclass MethodDecl extends Declaration ::=
  (optional ReturnType, MethodId, FormalParamList, MethodBody) { }
nodeclass VarDecl extends Declaration ::= (VarId, VarType) { }
nodeclass VarType ::= (ClassId) {
  ClassDecl function typeDeclBinding()
};
nodeclass Statement ::= { }
nodeclass MethodCall extends Statement ::=
  (ReceiverId, SelectorId, ActualParamList) {
    ClassDecl function receiverDeclBinding();
    MethodDecl function selectorDeclBinding()
  }
nodeclass WhileStmt extends Statement ::= (Expression, Statement) { }

```

Figure 5.1 – *Simplified Base Grammar Interface*

To support a pattern at the language level, it is necessary to identify the different roles in the pattern and then formulate the rules for these roles. Classes play the most important roles, but some roles may also be played by methods or variables. Typically, the roles correspond to the generic names used in the pattern structure diagrams in GoF's catalogue. For example, in the *DECORATOR* pattern [GAM95], the following roles are identified:

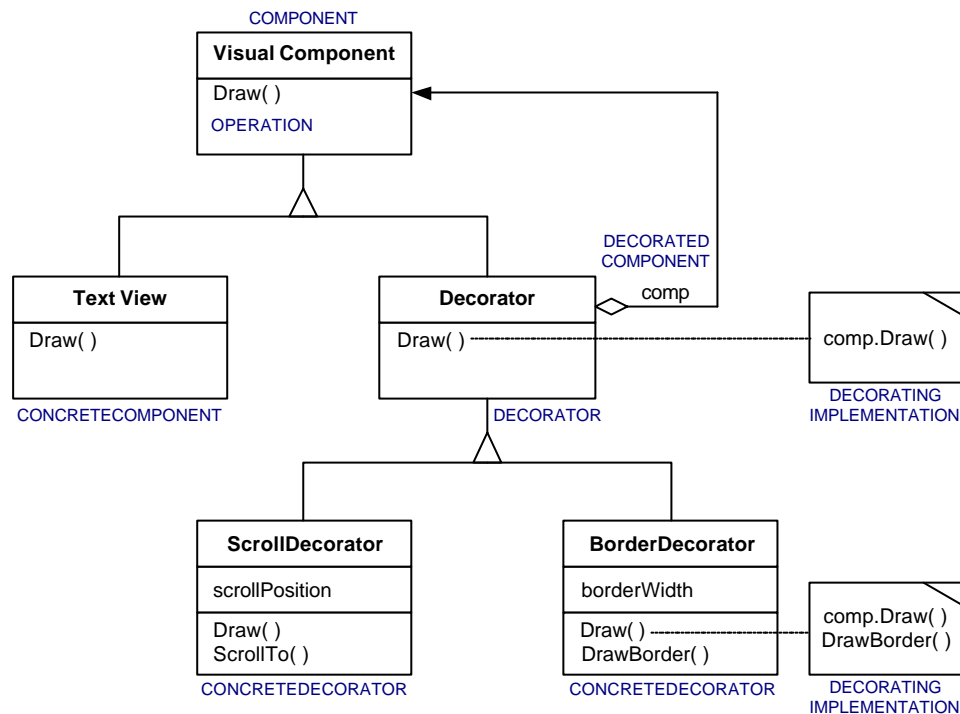


Figure 5.2 – Structure of *DECORATOR* according to GoF catalogue

- COMPONENT: An abstract class for components to be decorated.
- OPERATION: An operation belonging to the interface of COMPONENT.
- CONCRETECOMPONENT: An ordinary subclass of the COMPONENT.
- DECORATOR: A special subclass of the COMPONENT serving as an abstract decorator. This class has a DECORATEDCOMPONENT and possibly some DECORATINGIMPLEMENTATIONS.
- DECORATEDCOMPONENT: A variable declared in a DECORATOR, which denotes the decorated COMPONENT.
- CONCRETEDECORATOR: A subclass of the DECORATOR that may contain DECORATINGIMPLEMENTATIONS.
- DECORATINGIMPLEMENTATION: An implementation of an OPERATION which delegates the call to the DECORATEDCOMPONENT, and optionally also performs additional actions before and/or after the delegating call.

To support the identification of a pattern in the source code, annotations are made in the source code with pattern roles, using attribute comments. Nevertheless, it is not necessary to mark *all* the roles in the source program, because many of the roles can be derived from the other roles. For the decorator, it is sufficient to explicitly mark the

COMPONENT, DECORATOR, and DECORATEDCOMPONENT roles. These roles are called *defining roles*. Other roles are called *derived roles*.

The use of derived roles ties the pattern closer to a specific implementation. For example, if all methods in COMPONENT play the OPERATION role rules then the programmer cannot have some methods in COMPONENT that are outside of the DECORATOR pattern. Derived roles reduce the flexibility of the implementation but on the other hand, they reduce the burden on the programmer to explicitly state the pattern roles.

Patterns may overlap so that a given class plays different roles in different patterns. This is no problem with this approach, because a class (or method, etc.) may be marked by several attribute comments to indicate its different roles. The rules for applying a pattern can be expressed in terms of the pattern roles.

The identified roles must be consistent with each other. The rules that express such consistencies are known as *role rules*. If the role rules are satisfied, the pattern application is sufficiently complete to make it possible to go on with checking the collaborations among the different roles. To check that the collaborations occur according to the pattern, we formulate a number of *collaboration rules*.

5.2 Demonstration with DECORATOR pattern

An application program making use of the DECORATOR pattern is annotated by the defining roles, using attribute comments, like in the following illustration:

```
(*= DECORATORPATTERN_COMPONENT =*)
class VisualComponent {
    void draw() { };
};
class TextView extends VisualComponent {
    void draw() {...};
};
(*= DECORATORPATTERN_DECORATOR =*)
class Decorator extends VisualComponent {
    (*= DECORATORPATTERN_DECORATEDCOMPONENT =*)
    VisualComponent comp;
};
class BorderDecorator extends Decorator {
    void draw() {drawBorder(); comp.draw()}
};
```

Figure 5.3 – Using attribute comments to define roles in patterns

For the DECORATOR pattern the following role rules are identified:

- R1: A DECORATOR should be a subclass of a COMPONENT.
- R2: A DECORATOR should have a DECORATEDCOMPONENT variable.
- R3: A DECORATEDCOMPONENT variable must only occur in a DECORATOR.

The collaboration rules are:

- R4: A CONCRETEDECORATOR must have a DECORATINGIMPLEMENTATION for each OPERATION declared in the COMPONENT. The DECORATINGIMPLEMENTATION may be declared in CONCRETEDECORATOR or in any of its super classes.
- R5: A DECORATINGIMPLEMENTATION must contain a delegating call to the corresponding OPERATION of its DECORATEDCOMPONENT.

The collaboration rules are usually more interesting than the role rules in that they are more easily broken by mistake, and therefore more interesting to enforce. For example, if we are working with a window system applying the *DECORATOR* pattern, it is easy to forget to update the *CONCRETEDECORATOR* classes with delegating operations each time a new *OPERATION* in the *COMPONENT* is added. This error might not show up immediately, because applications that do not make use of the decorating objects will work fine. A system that enforces the pattern rules would detect such errors at compile-time.

To specify the roles and rules for a pattern, an extension grammar is written which extends the base grammar interface with attribute declarations and equations defining the attribute values. Figures 5.4 and 5.5 shows some brief examples of the extension grammar. The complete demonstration is in [HED97].

To support the annotations of the defining roles *COMPONENT*, *DECORATOR*, and *DECORATEDCOMPONENT*, one option is to declare three program-defined attributes as follows:

```
addto ClassDecl {
  progdef boolean DecoratorPattern_Component = false;
  progdef boolean DecoratorPattern_Decorator = false;
}
addto VarDecl {
  progdef boolean DecoratorPattern_DecoratedComponent = false;
}
```

Role attributes:

```
Role COMPONENT
addto ClassDecl {
  syn boolean Component = DecoratorPattern_Component;
}
```

The other roles have similar implementations.

Figure 5.4 – Roles in *DECORATOR* pattern

Rule 1: A DECORATOR should be a subclass of the COMPONENT

This can be checked by an error attribute as follows:

```
addto ClassDecl {
  error string missingComponentRole =
    if DecoratorPattern_Decorator and
      superclassBinding() != null and
      not superclassBinding().Component
    then "Decorator pattern: Missing Component role for Decorator."
    else "";
}
```

Rule 3: A DECORATEDCOMPONENT variable must only occur in a DECORATOR.

This rule says that if a variable is marked as a DECORATEDCOMPONENT, it must be an instance variable declared in a DECORATOR.

```
addto VarDecl {
  error string misplacedDecoratedComponent =
    if DecoratedComponent and
      enclosing(Declaration) == enclosing(ClassDecl) and
      enclosing(ClassDecl).Decorator
    then ""
    else "Decorator Pattern: Misplaced DecoratedComponent";
}
```

Figure 5.5 – *Examples of rules using the extension grammar*

5.3 Conclusions about Extended Grammars

The language for design patterns outlined in this section supports both the identification of patterns in source code (traceability), and automatic checking that the patterns are applied consistently, according to given rules.

Nevertheless, the programmer needs to explicitly annotate the source code with some design patterns roles – there is no recognition from the source code only. This is not critical, because it is the programmer who knows what pattern is intended to be used by the system. One advantage of annotations is that if the programmer, by mistake, breaks the pattern rules, it is still possible to recognize the pattern and give some warning to the developer.

This mechanism has a cost in flexibility: any pattern formalization pins down precise rules for the pattern, and it might be difficult to foresee all reasonable implementation variations of the pattern.

6. CONSTRAINT DIAGRAMS

This methodology specifies patterns by breaking them up into three models (*role*, *type* and *class*) [KEN98]. The *role model* is the most abstract and depicts only the essential spirit of the patterns, excluding inessential application-domain-specific details. The *type model* refines the role model with abstract states and operation interfaces forming a (usually domain-specific) refinement of the pattern. The *class model* implements the *type model*, thus deploying the underlying pattern in terms of concrete classes.

The use of these models makes it easier to express patterns in their full generality (essence) contrary to the notation provided by GoF catalogue. For example, the GoF presentation of *ABSTRACT FACTORY* suffers from the major demerit that it actually represents a single deployment of the *ABSTRACT FACTORY* pattern rather than the generalized pattern itself. More specifically, patterns in GoF define specific operation interfaces for the classes, which are implemented by fixed concrete classes. This significantly reduces the general applicability of the patterns because other deployments would require different numbers of and different properties for these types and operations.

To solve the problem of expressing names and quantities, constraints diagrams presented by [KEN97] can be used. Collections are represented in terms of sets, upon which it is possible to specify constraints applying to set members. Sets enable talking about collections generally (without premature commitment to cardinality or naming), and constraints enable talking about collections precisely.

6.1 Constraint Diagrams Notation

Constraint diagrams depict sets as Venn diagrams [GIL98]. An arbitrary member of a set is depicted via a dot within or on the edge of the set (see figure 6.1). Two unconnected dots are definitely distinct. The presence of two dots connected via a dashed line indicates that the dots do not necessarily represent distinct elements (i.e., they may be the same element). Two dots connected via a strut (see figure 6.2) represent alternative positions for a single element (i.e. an element may reside in one and only one of the positions represented by the connected dots at any time).

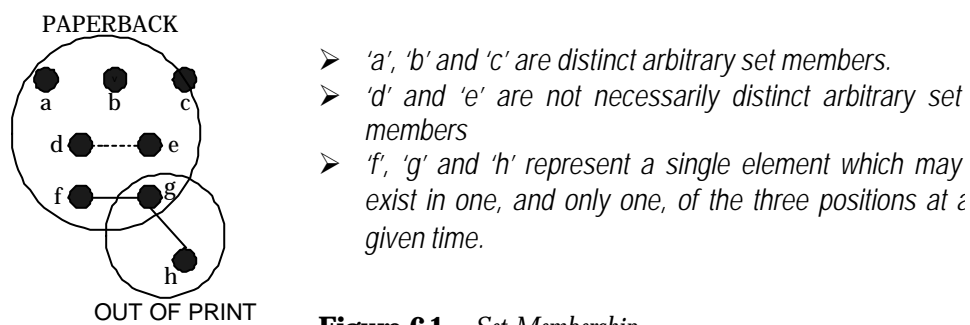


Figure 6.1 – Set Membership

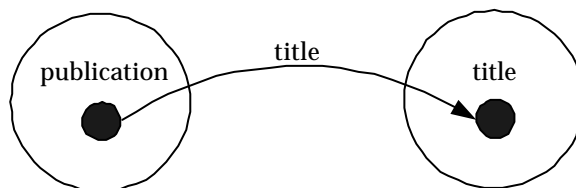


Figure 6.2 – Set Membership: Each publication has a title

The UML notation is extended to talk about instances of classes in abstract terms. This is done with the addition of a fourth compartment to a class symbol, which holds abstract instances of the class, specified as a constraint diagram.

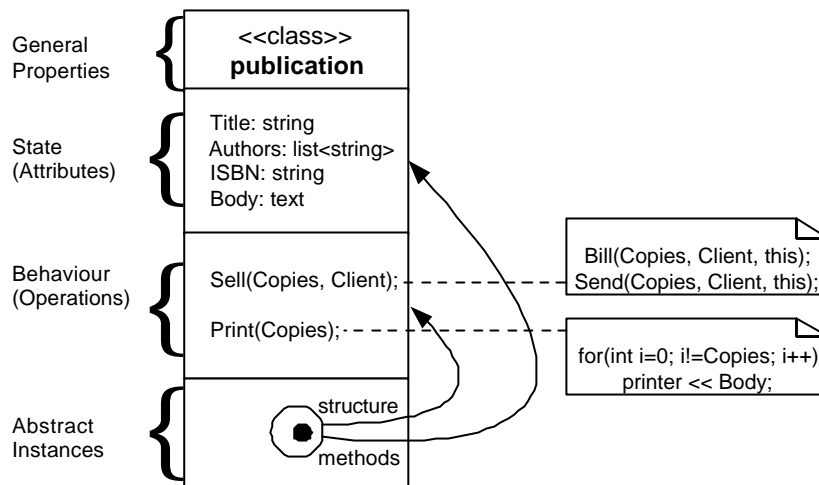


Figure 6.3 – UML Class Diagram + Abstract Instances

6.2 Three Layered Modeling

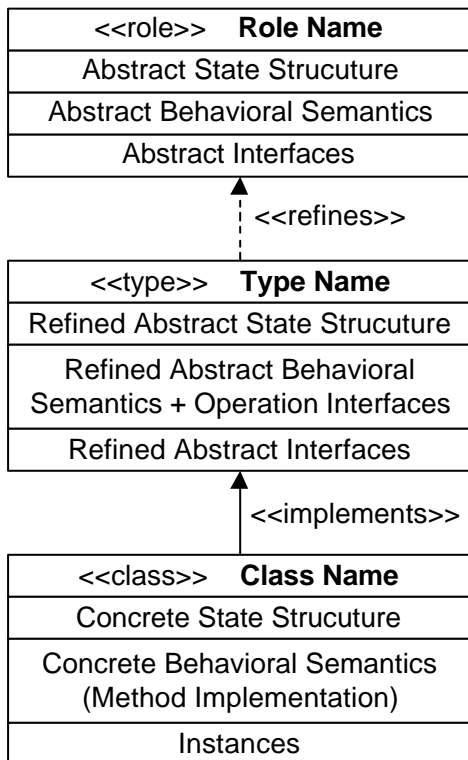


Figure 6.4 – Three-Model Layering

The first layer (the role model) expresses the pattern purely in terms of highly abstract states and highly abstract behavioral semantics, forming a constraint set that captures the essential spirit of the pattern without dilution in non-essential (application-domain specific) details.

The middle level (the type model) refines the role model adding usually-domain-specific refinements to the abstract states and semantics, and concrete syntax for operations described by the abstract semantics.

The final layer (the class model) deploys the type model in application-specific terms via the specification of concrete state (attributes) and concrete semantics (method implementations) that implements the abstract state and abstract semantics, respectively.

6.3 Demonstration of *ABSTRACT FACTORY* pattern

The GoF representation of the structure of *ABSTRACT FACTORY* (figure 6.5) as a class model (figure 6.7) may give the false impression that components of a design pattern are actually classes, while they are better thought in more abstract ways. Specifically, the pattern may be re-expressed as a type model, which in turn may be refined continually into a hierarchy of derived type models, each adding constraints to the type model above the hierarchy. However, when a commitment is made to concrete (rather than abstract) states and concrete method implementations, by adding more details, the class model is achieved.

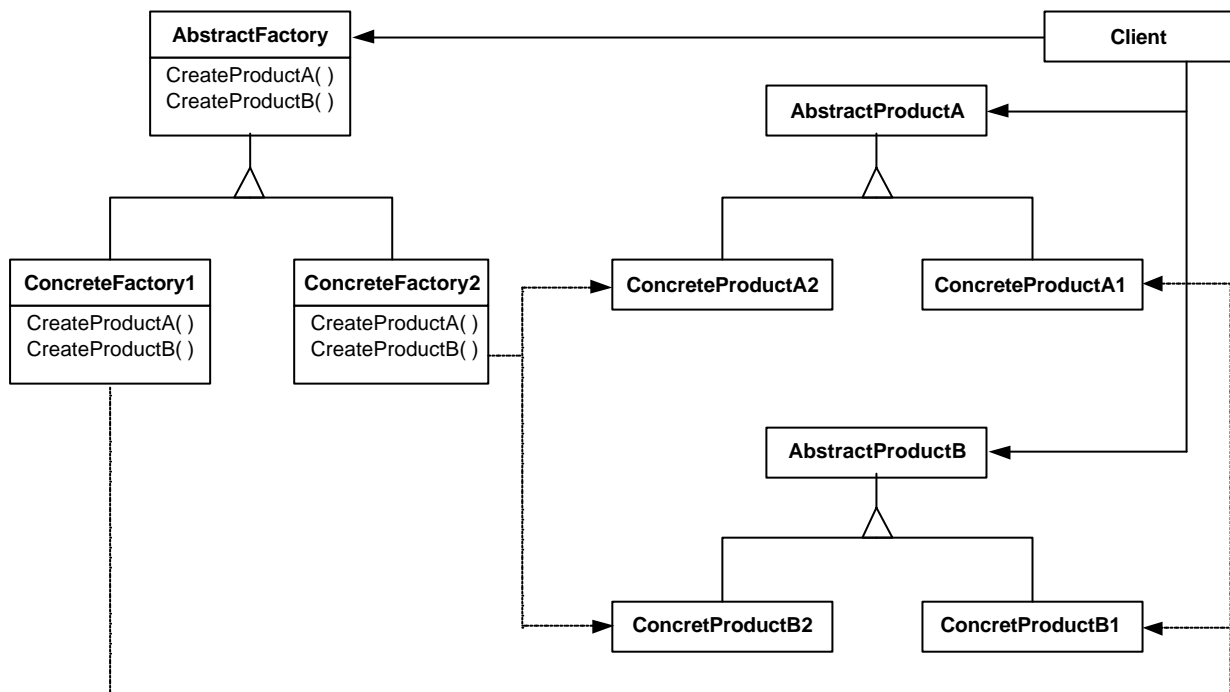


Figure 6.5 – Structure of *ABSTRACT FACTORY* according to GoF catalogue

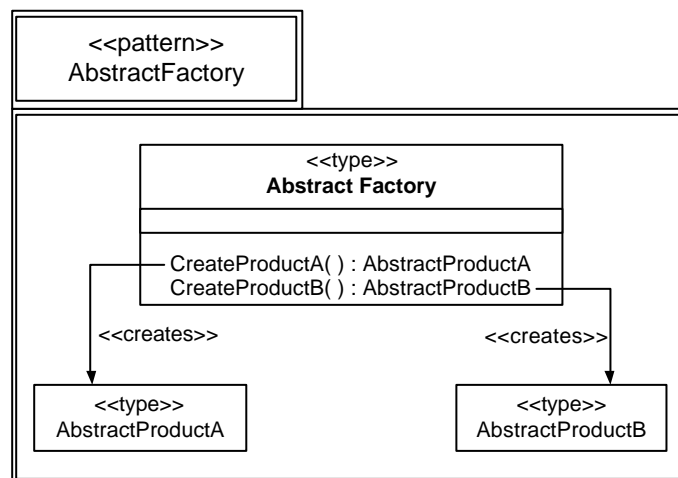


Figure 6.6 – *ABSTRACT FACTORY* as a type model

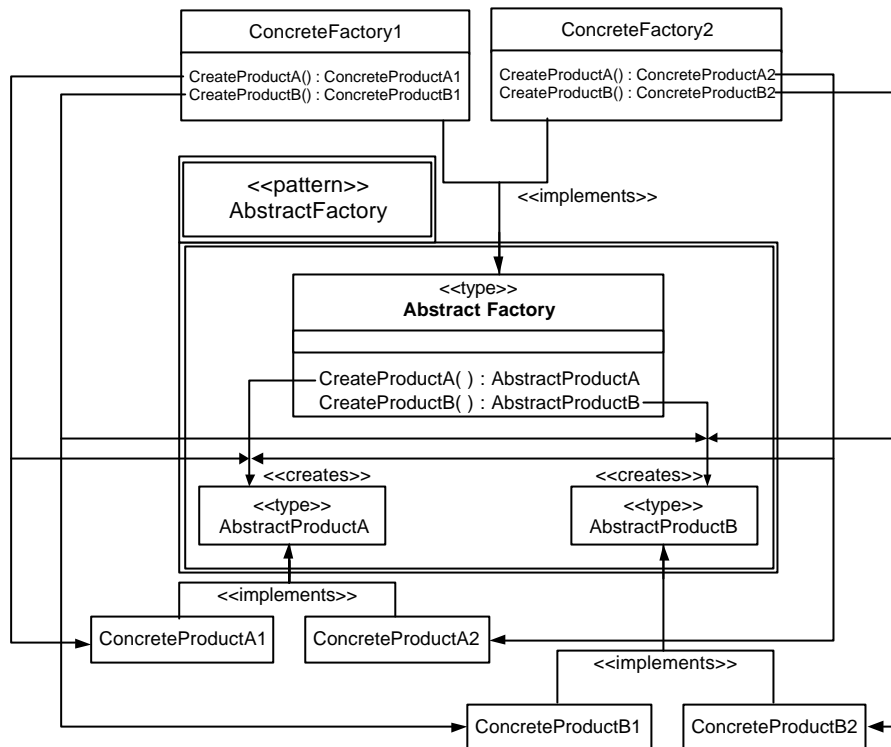


Figure 6.7 – *ABSTRACT FACTORY* deployed as a Class Model

The main contribution of the type model is that it only abstracts structure from the pattern, leaving details of concrete implementation to derived class models. Thus, it is possible to see figure 6.6 as a specific realization of the type model that is sufficiently abstract to permit many other class model realizations.

However, to capture only the essential spirit of the pattern, and remove non-essential features, which constraint the pattern generality, roles are utilized in the models. Roles are collaborating actors. Thus, a role model is a description of a structure of co-operating objects along with their static and dynamic properties. Constraints (on state and on behaviour) in the role model must be respected by further refinements.

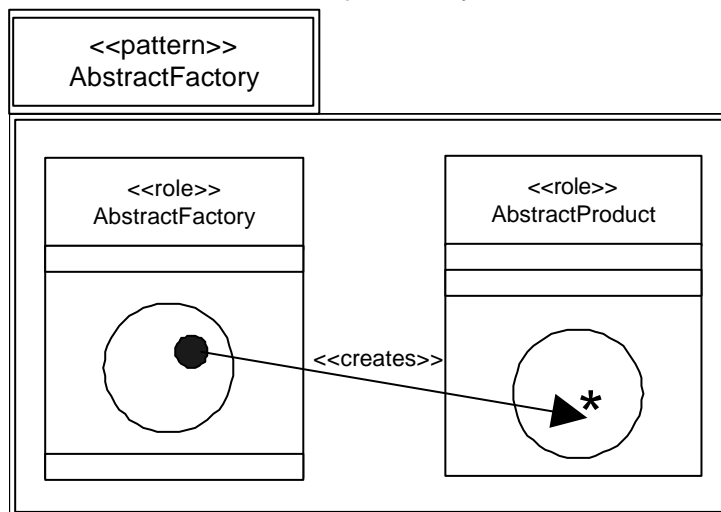


Figure 6.8 – *ABSTRACT FACTORY* as a role model

Figure 6.8 is a pure representation of *ABSTRACT FACTORY*. It conveys, in terms of structure and behaviour, the whole pattern, and nothing but the pattern. Note that a solid star represents the creation of a set instance.

The figure shows that players of the ABSTRACTFACTORY role share a set of semantics for operations. Each operation in that set is defined as creating a specific type of ABSTRACTPRODUCT. Thus, all players of the ABSTRACTPRODUCT role must implement a set of methods adhering to this semantics, creating via these methods the same set of ABSTRACTPRODUCT types. The operation section of the ABSTRACTFACTORY role is not expressed in terms of concrete interfaces; rather it is expressed as a constraint diagram depicting a set of semantics for the operations. Neither name to these operations nor their concrete cardinality was required prematurely. Instead, meta-level constraint information was expressed, being respected by any type-model derived from this role model.

Once more, a role model can be refined continually into a hierarchy of role models. However, as soon as a commitment is made to concrete-operation syntax a type model is derived from the role model.

6.4 Conclusions about the model

This work shows that visual notations can present the essence of patterns precisely and expressively. These concepts are achieved by adopting a three-model layering of pattern descriptions, wherein the essential of the pattern is represented as a role model, further refined by a type model, and implemented by a class model.

The essence of three-model layering is to utilize abstraction without loss of expressiveness, thus achieving maximal generality and un-ambiguity in pattern description. The authors have also worked in the sense to make patterns formal specification easier to interpret providing only a visual notation.

To demonstrate the practical application of the technique, the authors mention a partnership with a commercial enterprise, which intends to apply the model in the mining of a large legacy system for migrating it to component-based technology. They also investigate the requirements of case tool support for this model.

7. THE DISCO METHOD

A DisCo specification [MIK98] is a definition of (a pattern for) a system. In each system, the developer can introduce *classes*, *relations* and *actions*. Below the specification is shown. Later on, one example is illustrated with the *OBSERVER* pattern.

7.1 Elements of Pattern Formalization

Classes are formulas defining the form for possible objects. For instance, a class C can be defined as:

$$\text{class } C = \{x\},$$

where x is an un-typed variable. With this definition, each instance o of class C contains a local variable denoted as $o.x$.

Relations are used for associating objects with one another. They are defined in the format:

$$\text{relation } (n).R.(m): C \times D,$$

where relation R associates n instances of class C with m instances of class D . Asterisk (*) is shorthand for any possible number of instances.

Actions are atomic units of execution, which can be understood as multi-object methods. An action consists of a list of required participants and parameters, an enabling condition, and the definition of state changes caused by an execution of the action. For example, an action A can be given as:

$$\begin{aligned} &A(c:C; i): \\ &\quad i \neq c.x \\ &\quad \textcircled{R} \quad c.x' = i, \end{aligned}$$

where c is a role for an object in class C , and i denotes an un-typed value given as a parameter. Expression $i \neq c.x$ is the enabling condition under which the action can be executed. The following line defines the state change caused by an execution of the action.

Participants, i.e., objects that take a role in an action, and parameters, i.e., plain variables that denote individual values, are non-deterministically selected from those that are suitable. For example, the above action is enabled for values of parameter i that are different from the value of $c.x$. If there are several actions that could be executed, one is non-deterministically selected from those that are enabled.

Each DisCo specification is a description of the temporal behaviour of a closed system, which can be observed but not affected from outside. The behaviour of the system is thus completely defined by the specification, without any implicit control flow. A specification does not require the developer to fix the numbers of objects that are needed.

DisCo modularity consists of applying superposition to existing specifications, ensuring by construction that *safety properties* ('nothing bad will ever happen') are preserved. Each refinement step can introduce a relatively small set of global aspects instead of a large number of aspects local to an object. Such a refinement can be understood as a system-wide layer that introduces slices of objects, which resemble program slices.

As the unit of modularity is a behavioural layer rather than an individual object or class, the emphasis is on the behaviour of the system as a whole instead of local behaviours of independent objects. In practice, this means that new classes and variables can be introduced, and operations affecting the new variables can be added when refining

specifications. New operations are given as new actions, or as arguments to existing ones, with an option to add new conjuncts to enabling guards as well. Classes are extended using the format:

$$\text{class } C = C+ \{y\},$$

and action refinements are given in the format:

$$\begin{aligned} B(d:C; j): \\ \text{refines } A(c:C = d; i = j) \\ \dot{U} j \leq c.y \\ \textcircled{R} c.y' = c.y + j. \end{aligned}$$

As new conjuncts can be added to enabling condition, *liveness properties* ('something good will eventually happen') are not guaranteed to be preserved by construction.

In Disco, class refinement is a special form of inheritance. When extending a class, a new class is created, that is a subclass of the original class. In a normal refinement no instances of the original class exist outside the extension. If this is undesirable, inheritance can be used explicitly. A class D derived from base class C can be introduced in the format:

$$\text{class } D = C + \{\dots\}.$$

Multiple inheritance is also allowed. The adoption of inheritance implies a requirement to be able to specialize actions for different kinds of participants. For this purpose, an action B that specializes an action A for participants in subclass D can be given as:

$$\begin{aligned} B(d:D; j): \\ \text{refines } A(c:C = d; i = j) \text{ for } c \in D. \end{aligned}$$

This results in two actions. Action $B(d:D; j)$ is available for objects in class D , and action $A(c:C; i)$ is to be used by instances of class C that do not belong to class D .

7.2 Demonstration of *OBSERVER* pattern

At this point, the *OBSERVER* pattern given in GoF is formalized. Informally, this pattern can be described as follows.

There are *Subjects* and *Observers*. Each subject is a container of data whose contents can be modified, and each observer is an object that can be interested in the contents of a subject. The pattern describes how subjects and observers are connected with one another, and how they communicate to preserve data consistency. The pattern is illustrated in Figure 7.1. Intuitively, the characteristic property of the pattern is that whenever an observer receives data, the subject that the observer is attached to contains the same data.

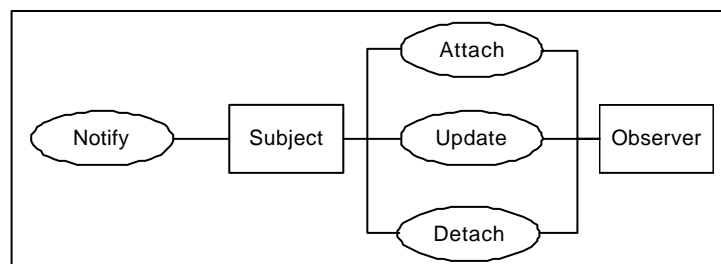


Figure 7.1 – An illustration of *OBSERVER* pattern

Based on the description of the pattern, it is relatively obvious that classes whose instances represent subjects and observers are needed. Each object may have an internal state, yielding to:

$$\begin{aligned} & \text{class Subject} = \{\text{Data}\}, \\ & \text{class Observer} = \{\text{Data}\}. \end{aligned}$$

In the formalization, *Data* is included in both classes to model the temporal behaviour related to the contents of the associated objects. Obviously, by omitting the data, an abstraction of this specification is obtained.

Two instances of the above classes are associated with each other whenever an observer is interested in the contents of a subject. This relation, representing an attachment of an observer to a subject, can be formalized as

$$\text{relation } (0..1). \text{ Attached. } (*): \text{ Subject } \times \text{ Observer.}$$

In addition to the attached observers, each subject needs to know to which observers its content has been delivered since the latest modification. Thus, subjects are associated with the observers that have already been updated. This yields to the following relation:

$$\text{relation } (0..1). \text{ Updated. } (*): \text{ Subject } \times \text{ Observer.}$$

In the pattern, an observer can become interested in the content of a subject, and may also cancel this interest. In the formalization, actions *Attach* and *Detach* are used for modeling beginning and cancellation of interest, respectively. Action *Attach* sets the *Attached* relation between the subject and the observer that are involved. Action *Detach* clears this relation, and ensures that possible *Updated* relations between these objects are cleared, too. These actions are formalized as follows:

$$\text{Attach}(s:\text{Subject}; o:\text{Observer}):$$

$$\emptyset s.\text{Attached}.o$$

$$\textcircled{R} s.\text{Attached}.o,$$

$$\text{Detach}(s:\text{Subject}; o:\text{Observer}):$$

$$s.\text{Attached}.o$$

$$\textcircled{R} \emptyset s.\text{Attached}.o$$

$$\tilde{U} \emptyset s.\text{Updated}.o.$$

Action *Notify* denotes that the content of a subject has been modified. At this level of abstraction this is interpreted as a need to update all observers that are attached to the subject. Thus, upon executing *Notify*, the subject must no longer be associated with any observer by *Updated* relation. This results in the following action:

$$\text{Notify}(s:\text{Subject}, d):$$

$$\textcircled{R} \emptyset s.\text{Updated}. \text{class Observer}$$

$$\tilde{U} s.\text{Data}' = d,$$

where parameter *d* models the new value, set upon notification and class *Observer* denotes all instances of class *Observer*. As no restrictions are imposed on the value of parameter *d*, its value is non-deterministically selected at this level of abstraction.

Action *Update* represents a transmission of modified data from a subject to an observer. Thus, it sets *Updated* relation for the subject and the observer that participate in the action, yielding to:

Update(s:Subject; o:Observer; d):*

s.Attached.o

$\dot{U} \emptyset s.Updated.o$

$\dot{U} d = s.Data$

$\textcircled{R} s.Updated.o$

$\dot{U} o.Data' = d.$

Marking participant *o* with an asterisk denotes a *fairness requirement*, stating that if an object could repeatedly take this role in this action, the action is repeatedly executed for the object. Such requirements are essential for *liveness properties*.

Due to the appropriate units of modularity and adequate level of abstraction, formalization of design patterns and specification obtained by utilizing DisCo are rigorous and practical.

7.3 Conclusions about DisCo Method

Each pattern in DisCo is formalized as a behavioural layer, introducing slices of objects that resemble program slices. This way, underlying patterns can be identified in a complete specification, enabling interpretations where patterns are naturally used as building blocks for specifications of more complex systems.

The use of layers enables projections of behaviours defined by complete specifications to individual patterns. Besides, DisCo allows multiple inheritance and without it the combination of patterns would be difficult. One example of patterns combination is presented in [MIK98].

The use of an abstract notion of atomic actions simplifies the communication between objects. However, it is not so easy to keep this abstraction when mapping the formalism to a programming language. Some standard refinements can be used to derive abstract cooperation into directly implementable communication.

Property-preserving refinements supported by the DisCo method provide a sophisticated way to combine rigorous and pattern-oriented software development. An implication of property preserving refinements is that the properties of a pattern can be validated and verified at pattern level, and when using the pattern, refinement steps enforce that the pattern is correctly in the resulting specifications.

8. LEPUS: LANGUAGE FOR PATTERNS UNIFORM SPECIFICATION

This chapter shows briefly the aspects of a declarative language called LePUS. LePUS is presented to academic community through many articles, and it seems to be the best solution found. Information about it can be found in [EDE98], [EDE98b], [EDE00], [EDE01], [GRO01] and many other papers.

LePUS is also a very recent method of formalization: it was first introduced in 1997. Since then, many works have been done to improve it, and until the actual moment it has been discussed and presented in conferences around the world. In 2002, LePUS will be in focus during the *NSF Design, Service, and Manufacturing Grantees and Research Conference* in Puerto Rico, as well as in the *6th World Conference on Integrated Design and Process Technology*, California, USA. Works related to LePUS can be found in [EDE02].

In LePUS, a program is represented primarily as a set of *ground entities* and relationships among them. The various interactions and associations that occur between *Participants* of design patterns are abstracted into a small, simple set of *relations*.

Following is a list of the primary abstractions that build on this scenario:

1. *Classes and functions (also methods or routines)* make the atomic, *ground* entities and are treated as irreducible units rather than composites. To account for what are ordinarily defined as the components (*members*) of classes, *relations* with other methods or other classes are used. Similarly, function arguments and return type are viewed as relations with the respective classes.
2. *Ground relations* between ground entities capture the structure and behaviour embodied in most design patterns. A small set of relations was defined as a basic part of the universe, such as: “*c is the first argument of f*”, “*f1 invokes f2*”, “*f1 forwards its arguments to f2*”, “*f is defined in c*”, and others (see Table 3).
3. *First order (also 1-dimensional) uniform sets* make the fundamental and most common abstraction. While *uniform* indicates elements of equal type (namely, either functions *xor* classes), sets of unbounded size are of interest with respect to a particular property, such as:
 - A set of classes that inherit from a given *abstract* class;
 - A set of functions with identical signatures redefined in such a hierarchy;
 - A set of classes whose instances are created by another set of functions, or *factory methods* in GoF terms.

A set of ground (also *0-dimensional*) classes/functions is also designated as *1-dimensional class/function*.

4. *Inheritance class hierarchies* make a special case of class sets. One class of each *hierarchy* is *abstract*, designated as the hierarchy’s *root*, while the remaining classes inherit (possibly indirectly) from it. Defined in this way, *hierarchies* are constructs that may be qualified by relations to other constructs, such as other hierarchies or *clans* regardless of the number of inheritance levels involved.
5. *Clans* make a special case of function sets: all the functions involved in a particular *clan* share the same signature, and each is defined in a different class (of a given

set). Members of a *clan* bear relationships, which typically allow dynamic selection of functions (*dynamic dispatch*).

6. *Tribes* are simply sets of clans with respect to a common class set.
7. *Uniform sets of higher order* are sets containing sets of identical type and dimension. A uniform set of classes/functions of dimension d is also denoted class/function of dimension $d+1$.
8. *Total Relations* are total functions that describe the relations between two sets of entities.
9. *Bijjective (also regular) correlations* between sets of *functions, classes, and hierarchies* are commonplace, such as in *ABSTRACT FACTORY* [GAM95]: “for every Product class, a Creator function redefines the Creator in Abstract Factory in the corresponding Concrete Factory class, creates, and returns the respective Product object” \rightarrow a regular Production relation.

These correlations occur between sets and sets thereof. *Regular* and *total* (functional) relations are the only types of relations that occur between sets. All relations between entities of any dimension extend systematically from ground relations as either *total* or *regular* relations.

10. *Commutativity*: Converging *regular relations* most often share the same entities in the *range* set. For instance, given two sets: a set M of *factory methods* and a set P of *product classes*, two regular relations are defined from M to P :
 - *Production* (m, p), indicating that m creates and returns an object of class p ;
 - *Return-Type* (m, p), indicating that the return type of function m is of class p .

The additional property observed is that the two relations between M and P converge in P , such that if function m_1 produces (an object of) class p_1 and its return type is p_2 then $p_1 = p_2$.

8.1 Textual Notation

Naturally, different programming languages incorporate different constructs, and the specification of design patterns at the appropriate level of abstraction requires more than the lowest common denominator. To avoid problems of language dependency, LePUS does not incorporate relations that are not primitives of all object-oriented programming languages. As such, the building blocks employed in its expressions, most notably the ground relations, are descriptions made by the authors of design patterns.

Following this principle the ground relations used in LePUS are obtained from the specification of the GoF design patterns. Table 4 lists the set of ground relations used in the specification of the GoF patterns and the intent behind each. However, the set of relations in table 4 is subject to extensions according to the needs of each system.

The set of the ground entities that are classes is represented as \mathbf{C} , and \mathbf{F} represents the set of the ground entities that are functions. Both \mathbf{F} and \mathbf{C} are referred to as *types* or *domains* with respect to the variables and relations in LePUS.

<p>Abstract: $F \cup C$ Indicates whether a class or a method is abstract.</p> <p>Creation: $F \times C$ Indicates that the body of the method contains an expression that results in the creation of a new instance of the class.</p> <p>Defined-in: $F \times C$ Indicates that a method is defined in a certain class.</p> <p>Forwarding: $F \times F$ Indicates a special kind of <i>invocation</i>, where the actual arguments in the invocation expression are the formal arguments defined for the first method.</p> <p>Invocation: $F \times F$ Indicates that within the body of the first method there is an explicit invocation expression of the second method (also: “may call”).</p> <p>Inheritance: $C \times C$ Indicated that the first class inherits from the second.</p> <p>Reference-to-single (-multiple): $C \times C$ Indicates that one class defines a member whose type is a reference to a single (multiple) instance(s) of the second class.</p> <p>Return-Type: $F \times C$ Indicates that the “return type” of the method is of the class.</p> <p>Same-Signature: $F \times F$ Indicates that the two functions have the same name and formal arguments. Satisfying this relation is a prerequisite for overriding (as required by the definition of <i>clans</i>.)</p>

Table 4 - Primary ground relations and their intent

For understanding purposes, let assume that a program p (or a class library of interest) is represented as an *object-oriented structure* or *model*: a collection of ground entities (*atoms*), composed of *classes* and *functions*, and relations among them.

A *structure* that arises from program p shall contain the classes and functions that are defined in p and the relations thereof. These relations may result from an explicit declaration in the program or have some implicit form.

Alternatively, a *model* can be viewed as a classic *relational database*, which consists of elements (*classes* and *functions*) and tables that correspond to the relations of the model.

Definition 1: Model

A *model* M is a pair $\langle P, R \rangle$ where P is a collection of ground entities that can be composed of functions and classes, and $R = R_1, \dots, R_n$ is the set of relations amongst.

Example 1: Model of a STATE Implementation

Below the code segment taken from the Sample Implementation of the STATE pattern in the GoF catalogue is showed. It contains slightly modifications to build a model that represents the code segment.

```

struct TCPConnection {
    void ActiveOpen() { _state->ActiveOpen(this); }
    void Close() { _state->Close(this); }
private:
    TCPState* _state;
};

struct TCPState {
    virtual void ActiveOpen(TCPConnection*) {}
    virtual void Close(TCPConnection*) {}
};

struct TCPEstablished : TCPState {
    static TCPState * Instance();
    virtual void Close(TCPConnection*) {
        // send FIN, receive ACK of FIN
        ChangeState(t, TCPListen::Instance());
    }
};

struct TCPClosed : TCPState {
    static TCPState * Instance();
    virtual void ActiveOpen(TCPConnection* t) {
        // send SYN, receive SYN, ACK, etc.
        ChangeState(t, TCPEstablished::Instance());
    }
};

```

The equivalent model is given as:

Entities:**Classes:**

- TCPConnection
- TCPState
- TCPEstablished
- TCPClosed

Functions:

- TCPConnection::ActiveOpen
- TCPConnection::Close
- TCPState::ActiveOpen
- TCPState::Close
- TCPEstablished::Close
- TCPEstablished::Instance

- TCPClosed::ActiveOpen
- TCPClosed::Instance

Relations:

- *Reference-To-Single* (TCPConnection, TCPState)
- *Inheritance* (TCPEstablished, TCPState)
- *Inheritance* (TCPClosed, TCPState)
- *Forwarding* (TCPConnection::ActiveOpen, TCPState::ActiveOpen)
- *Forwarding* (TCPConnection::Close, TCPState::Close)
- *Invocation* (TCPEstablished::Close, TCPListen::Instance)
- *Invocation*(TCPClosed::ActiveOpen,TCPEstablished::Instance)

Table 5 - Model for the STATE sample code

LePUS reflects the notion of nested sets through the definition of *dimension*.

Definition 2: Dimension

The dimension of an entity is defined inductively:

- Ground entities have dimension '0'
- A set of entities of dimension d and a uniform type is an entity of dimension $d+1$

Each LePUS *formula* stands for the complete and final representation of the solution of a single design pattern, namely, specifying the *Participants* and *Collaborations*. In accordance with this formulation, a *formula* in LePUS consists of *variables* and *relations*.

Formula 1: General Form

$\exists(x_1, \dots, x_n) : \bigwedge_i \mathfrak{R}_i(y_i, \dots, y_{i_n})$ where \mathfrak{R}_i are relation symbols of the types defined below, and x_1, \dots, x_n are all the free variables.

To use the GoF's terms, a pattern \mathbf{p} is represented by the formula $\mathbf{j}(\mathbf{p})$ such that $\mathbf{j}(\mathbf{p}) = \exists(x_1, \dots, x_n) : \bigwedge_i \mathfrak{R}_i(y_i, \dots, y_{i_n})$, the variables x_1, \dots, x_n are the pattern's *participants*, and the relations \mathfrak{R}_i specify the way they collaborate. Using this structure it is possible to retain much of the successful format and style adopted by the GoF catalogue.

Alternative types of the elements of LePUS formulae are summarized as:

1. *Variables*:

- *Ground variables*, ranging over ground entities;
- *Higher dimension variables*, ranging over higher dimension (i.e., sets of) entities;
- *Hierarchy variables*, ranging over inheritance class hierarchies.

Ground variables are typed, each represents a ground entity as expected. To deal with higher dimension entities, LePUS incorporates higher order typed variables: C^1 / F^1 vary over sets of classes/functions; C^2 / F^2 range over sets of sets of classes/functions. Generally, C^d / F^d are variables of dimension d that vary over classes/functions of dimension d . Lowercase letters c / f are shorthand for C^0 / F^0 , and C (F) as shorthand for C^1 / F^1 .

2. Relation:

- *Ground* relations, corresponding to those in \mathbf{R} , including *transitive* relations;
- *Generalized* relations, which derive systematically from the *ground* relations;
- *Commuting* relation.

A relation represents an association between *Participants* in the model, or in the GoF's terminology, a *Collaboration*. A relation symbol representing a relation in \mathbf{R} may be declared on ground variables. In addition, for every binary relation \mathbf{b} of \mathbf{R} we define a transitive counterpart $\mathbf{b}+$ as the transitive closure of \mathbf{b} .

Example 2: Abstract Class Representation as a Relation

Abstract(c) is a ground relation, and it is satisfied by a (O -dimensional) class c of \mathbf{P} if it is abstract.

Example 3: Transitive Inheritance

The (possible indirect) inheritance relation between concrete-class and abstract-class transcribes the predicate 1:

◆ *Predicate 1: Inheritance+* (concrete-class, abstract-class)

Higher order relations in LePUS are all derived systematically from the ground relations in one of the methods defined below. Let \mathbf{a} denote a ground unary relation, \mathbf{b} a ground binary relation; let lowercase $w, v, v1, \dots, vn$ designate *ground* variables; uppercase V, W designate 1-dimensional variables, then:

Definition 3: Generalized Relation

The following generalized relations are admitted:

- *Unary* $\mathbf{a}(V) \stackrel{def}{=} \forall v \in V : \mathbf{a}(v)$
- *Total* $\mathbf{b}^{\rightarrow}(V, W) \stackrel{def}{=} \forall v \in V \exists w \in W : \mathbf{b}(v, w)$
- *Regular* $\mathbf{b}^{\leftrightarrow}(V, W) \stackrel{def}{=} \forall v \in V \exists w \in W : \mathbf{b}(v, w) \wedge \forall w \in W \exists v \in V : \mathbf{b}(v, w)$
(\mathbf{b} is an *invertible* function (1:1 and *onto*) from V to W)

Example 4: Total Invocation Relation in the BRIDGE

Each one of the Operation functions implements as abstract operation by invoking some sequence of Operation-Imp functions in the BRIDGE pattern. This description can be translated to the predicate: $Invocation^{\rightarrow}(Operations, Operations - Imp)$, which partakes in the specification of the BRIDGE.

Example 5: Total Inheritance Relation in (Single) Inheritance Class Hierarchy

The predicate $Inheritance^{+\rightarrow}(Nodes^1, root^0)$ indicates an inheritance class hierarchy whose base is root and each class of Nodes inherits (possibly indirectly) from root (root is treated here as a singleton set).

Example 6: Regular Creation Relation in the FACTORY-METHOD

The FACTORY-METHOD pattern requires that every function of the set Factory-Methods creates exactly one class of Products, and that every Product's class is created by exactly one Factory-Methods' function. To represent exactly this, a regular Creation relation between the sets is establish in Predicate 2:

◆ *Predicate 2: $Creation^{\leftrightarrow}(Factory-Methods^1, Products^1)$*

The type of each one of the variables: $Factory-Methods^1$ and $Products^1$ are specified in the complete formula:

Formula 2: Towards the FACTORY-METHOD

$\exists Factory-Methods \in 2^F, Products \in 2^C : \langle Predicate2 \rangle$

Example 7: Regular Defined-In Relation in the OBSERVER

An update function should be defined in each observer class of the OBSERVER pattern. Defining $Update1$ as a set of functions and $Observers1$ as a set of classes, it is possible to set Defined-In as a regular relation between the sets.

At this point, these examples must have clarified how LePUS formalism works. Later in this chapter, the concepts will become more intuitive through examples.

8.2 Visual Notation

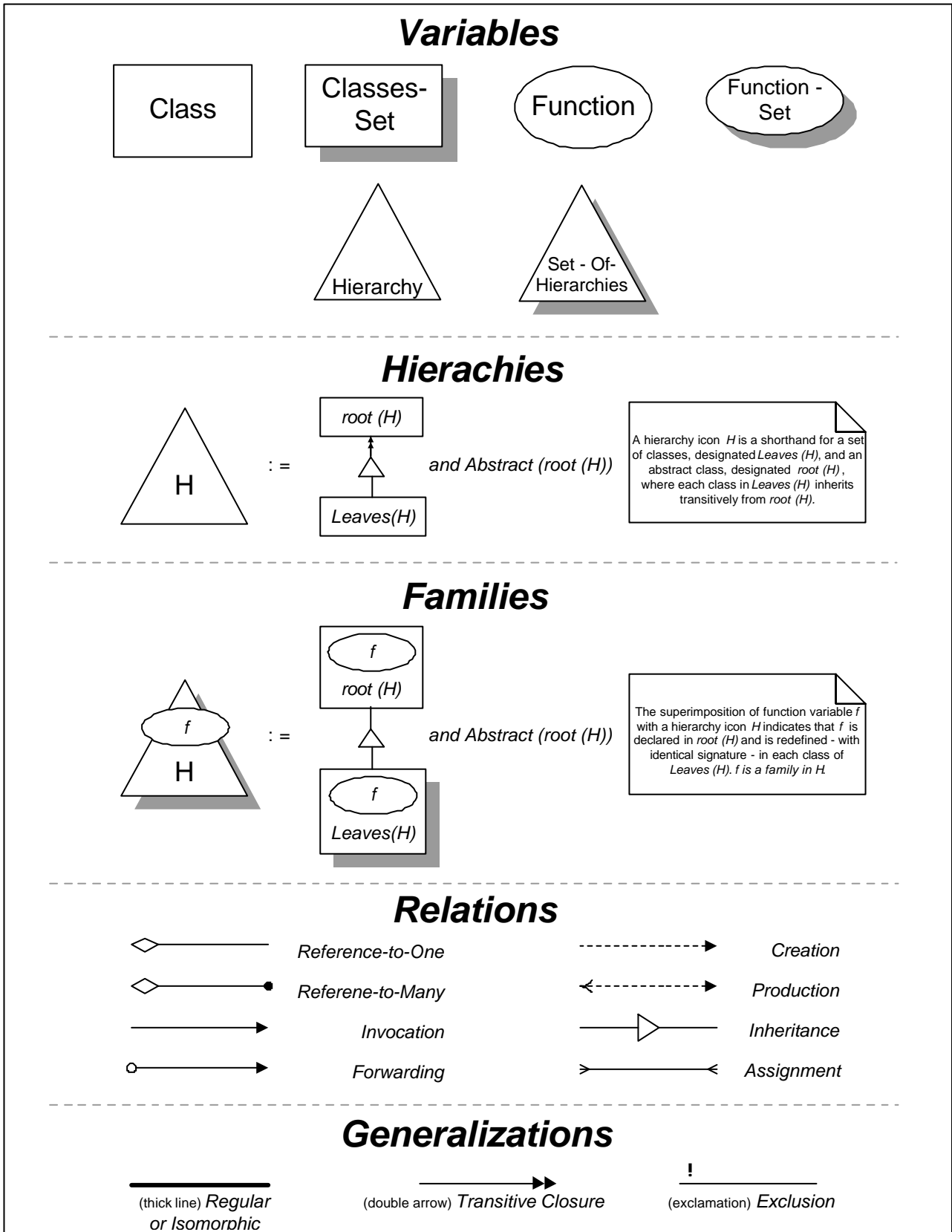


Figure 8.1 – Basic graphic symbols of LePUS

A diagram in LePUS is a graph whose vertices are *icons*, possibly adorned with a *unary relation mark*, and whose edges are labelled by (possibly generalized) binary relations each.

A graph depicts a formula as follows:

- *Icons* stand for variables.
- *Unary relation marks* stand for unary relations applied to the designated variable. More specifically, the mark *a*, designating vertex *v*, gives rise to the predicate: *a* (*v*).
- *Edges* stand for binary relations applied to the variables they connect. More specifically, an edge *?*, connecting vertices *v1*, *v2*, gives rise to the predicate: *?* (*v1*, *v2*).
- *Commute* designation(s) circumscribe the edges (relations) and vertices (set variables) that commute. A segment of a diagram that is also a *well-formed diagram* may be circumscribed by the *commutative* designator, thereby indicating that the regular relations thereof *commute* over the indicated sub-domains (set variables).

Figure 8.1 lists the most common elements of the graphical notation, including variables, relations and basic interpretations of the symbols. The direction of the binary relations edges is from left to right.

Clans and *tribes* relationships are portrayed as functions, i.e., (shaded) ellipses, superimposed on the respective class set. Figure 8.2 lists all valid superimpositions and the indicated dimensions of the clan/tribe.

Thus, for instance, the ellipse *f* superimposed on the shaded square *C* (second figure on the top row, Figure 8.2) indicates a function of dimension 1, the reason being the dimension of the class set *C* is 1, and by the definition of a *clan*: $dim(f) = dim(C)$.

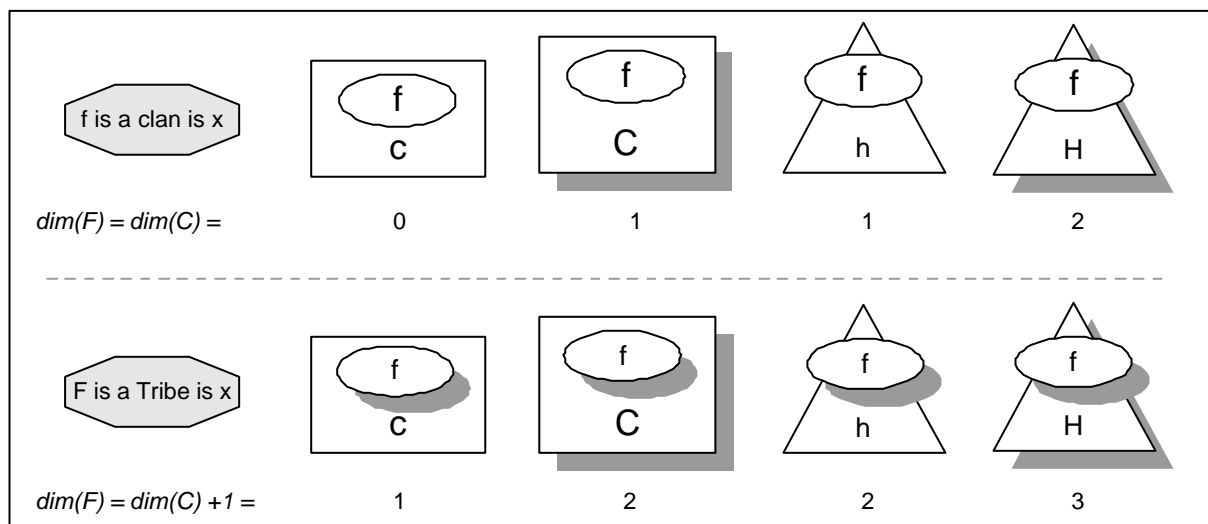


Figure 8.2 – Superimposing a function symbol with a class symbol is used to represent clans and tribes relationships

8.3 Demonstration of *STRATEGY* pattern

LePUS is first demonstrated using the specification of (the solution proposed by) the *STRATEGY*, depicted in Figure 8.3.

One key concept introduced in Figure 8.3 is that of an *inheritance class hierarchy*, or simply *hierarchy*, drawn as the *Strategies* triangle in this example. A hierarchy consists of an abstract *root class* and of a set of concrete *leaf classes*, all of which inherit (possibly indirectly) from the root class (see also figure 8.1).

Another key concept introduced in Figure 8.3 is that of a *family of functions*. A set of functions F is a family in a set of classes C if all the function in F have the same signature (arguments and return type), and each is defined in a different class in C . If one class of C inherits from another, such as the case with hierarchies, then different functions in F override each other.

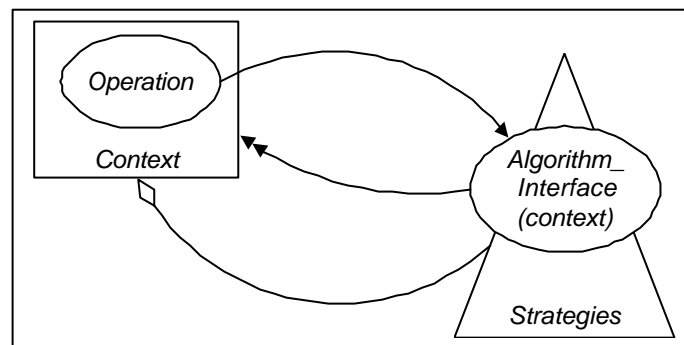


Figure 8.3 – LePUS diagram of *STRATEGY*

- Context

Context is a class variable that comprises (at least) a function marked *operation*, and holds a reference to the root class in the *Strategies* hierarchy, which may be interpreted as a pointer data member. A relation arc ending in a hierarchy is interpreted as relating to the root class in the hierarchy.

- Operation

Operation is a function variable. It *invokes* a function of the *algorithm_interface* set, as indicated by the *Invocation* relation that links the variables. Superimposing it with the *Context* class indicates it is defined in it.

- Strategies

Strategies triangle stands for a complete inheritance class-hierarchy, comprising an abstract class in its (single) root (*'STRATEGY'* of the GoF example) and the indefinite number of classes (*'concrete_strategy'*) at its leaves. The triangle icon does not indicate whether there are intermediate classes in the inheritance hierarchy.

- Algorithm Interface

Algorithm_interface designated ellipse indicates a function with a single argument of type *Context*. However, superimposing it with the *Strategies* hierarchy means that a function by such signature is defined in each class of the hierarchy. We say that the set of

algorithm_interface functions forms a *family* in the *Strategies* hierarchy, stressing the correlation between the functions, themselves, and the classes of the hierarchy.

A *Transitive Invocation* relation links the *algorithm_interface* family with the context class. This *Transitive* modifier assigned to the *Invocation* relation indicates that every function of the *algorithm_interface* family invokes directly or indirectly some function defined in *Context*.

The diagram of Figure 8.3 depicts in LePUS only the most general notion of the *STRATEGY* design pattern. Nonetheless, it is possible to use LePUS to detail refinements of the pattern and introduce contextual aspects of possible implementations. For instance, we can incorporate the client and indicate its role, as demonstrated in Figure 8.4. Figure 8.5 shows the corresponding textual notation.

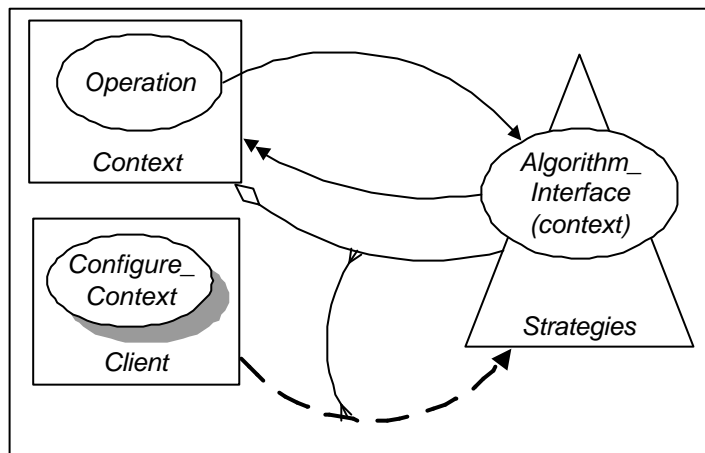


Figure 8.4 – A Refinement of *STRATEGY*

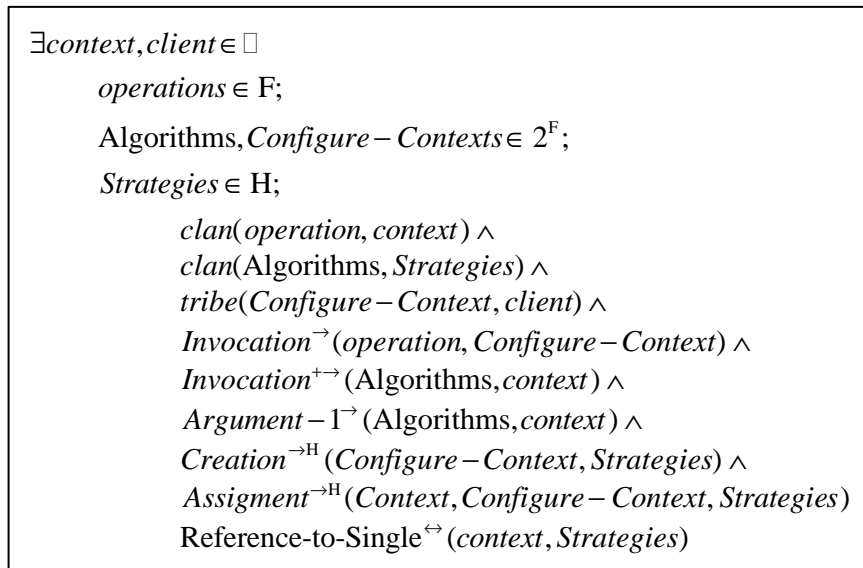


Figure 8.5 – LePUS formulae of *STRATEGY*

- Client

Client is a class variable that incorporates a set of methods, *Configure_context*

- Configure Context

Configure_context, being a shaded ellipse, designates an unbounded set of functions, each of which has *Context* as the single argument. *Configure_context* is connected with an arc of a *Creation* relation pointing at the *Strategies* hierarchy, indicating that every function of *Configure_context* creates some class in *Strategies*.

An *Assignment* arc connects the *Creation* arc with the *Reference* arc, indicating that the reference is being assigned by an object of the *Strategies* hierarchy that is created by a method of *Configure_context*.

Relations defined on sets are interpreted as total functions. For instance, the *Creation* arc from *Configure_context* to *Strategies* indicates that every function of *Configure_context* creates some class in *Strategies*. Formally, a binary relation $R(S_1, S_2)$ indicates that $\forall x \in S_1 \exists y \in S_2 : R(x, y)$, where S_1 and S_2 are sets of entities of the type as determined by the relation's definition.

8.4 Demonstration of FAÇADE pattern

The FAÇADE pattern (see figure 8.6 for the LePUS graphic notation and figure 8.7 for its textual notation) departs from the previous patterns in the *hiding* concept. What appears to be the essence of this pattern is the role of the *façade* class as a front that shields the *Subsystem-classes* from other modules, serving as form of a control panel. This is precisely the intent of the *Exclusion* modifier, drawn as an exclamation mark, intuitively described as designating that a particular entity is the only one that holds this relation to another entity.

$$R(p!, q) \leftrightarrow R^{-1}(q) = p$$

$$R(p, q!) \leftrightarrow R(p) = q$$

where R is some binary relation, p and q are entities that suit that types expected in their respective positions within the relation.

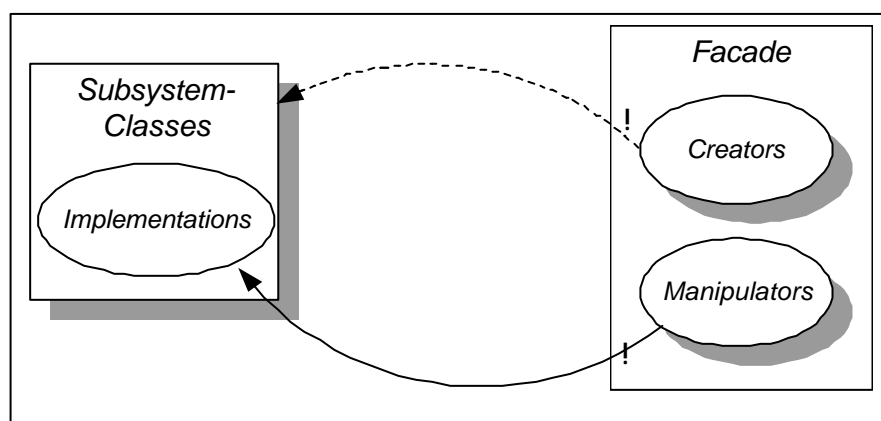


Figure 8.6 – LePUS diagram of FAÇADE

$$\begin{aligned} &\exists \text{Implementations, Creators, Manipulators} \in 2^F; \\ &\text{Subsystem-Classes} \in 2^C; \\ &\text{facade} \square : \\ &\quad \text{clan}(\text{Implementations}, \text{Subsystem-Classes}) \wedge \\ &\quad \text{tribe}(\text{Creators}, \text{facade}) \wedge \\ &\quad \text{tribe}(\text{Manipulators}, \text{facade}) \wedge \\ &\quad \text{Creation}^{\rightarrow}(\text{Creators!}, \text{Subsystem-Classes}) \wedge \\ &\quad \text{Invocation}^{\rightarrow}(\text{Manipulators!}, \text{Implementations}) \end{aligned}$$

Figure 8.7 – LePUS formulae of FAÇADE

- Creators

Creators is a set of classes defined in *façade*, each of which *Creates* some class of *Subsystem-classes*. The *Exclusion* modifier, designated by the exclamation mark, indicates there is no other function outside *Creators* that creates a class of *Subsystem-classes*.

- Manipulators

Manipulators, similarly, is a set of functions defined in *façade*, comprising all the functions that invoke any function of *implementations*.

8.5 Conclusions about LePUS

A brief outline of LePUS was presented. LePUS is founded in logic; it models semantics as well as purely structural relationships, and facilitates reasoning with high-order sets. LePUS operates at a higher level of abstraction than other descriptive notations and permits concise description of complex software artefacts.

The visual formalism conveys powerful abstractions in a single picture. It can be used to formulate precisely and clearly the detailed documentation of applications frameworks.

LePUS can be used to formalize patterns but, more important, is that the author plans to build LePUS reasoning techniques into software tools. Although the formalism is undecidable, LePUS formulas can always be validated with respect to a particular finite system. Software tools based on LePUS should be able to verify that software conforms to patterns, detect the existence of patterns in legacy code, and perform other useful tasks that currently resist automation.

Clearly, it is possible to see that LePUS is a powerful mechanism. Perhaps, its only disadvantage comes from the fact that it is not easy to understand, and to convert a pattern into its formulas, one may need to master the concepts first. The visual notation smoothes this drawback.

9. PDL: PATTERNS DETECTION LANGUAGE

In their work, Albin-Amiot and Guéhéneuc [ALBa01], [ALBb01] formalized patterns using a set of basic bricks called *entities* and *elements*. These bricks are defined in the core of a meta-model dedicated to the representation of patterns (the pattern meta-model). In order to evaluate the meta-model more rapidly, they have not defined it as an extension of an existing one (like *UML* for example). The pattern meta-model, which was experimented in *Java*, provides a mean to describe structural and behavioural aspects of design patterns. From this description, it gives the required machinery to produce code and to detect instantiated patterns in code.

The intended contribution of their approach is the reification of design patterns as first-class modeling entities. Reified design patterns are used to produce their associated code implementation, according to the context of their application, and to detect their occurrences in user's code. The way to apply a design pattern is deduced solely from its declaration, not from external hints or specifications. Consequently, this work is the most complete one, because it not only formalizes patterns but also uses this formalization to introduce or to detect patterns in the user's code.

9.1 The Patterns Meta-Model

This section presents a meta-model that handles uniformly instantiation and detection of design patterns (figure 9.1). The meta-model embodies a set of entities and the interaction rules between them. All the entities needed to describe the structure and behaviour of the structural design patterns introduced in [GAM95] are present.

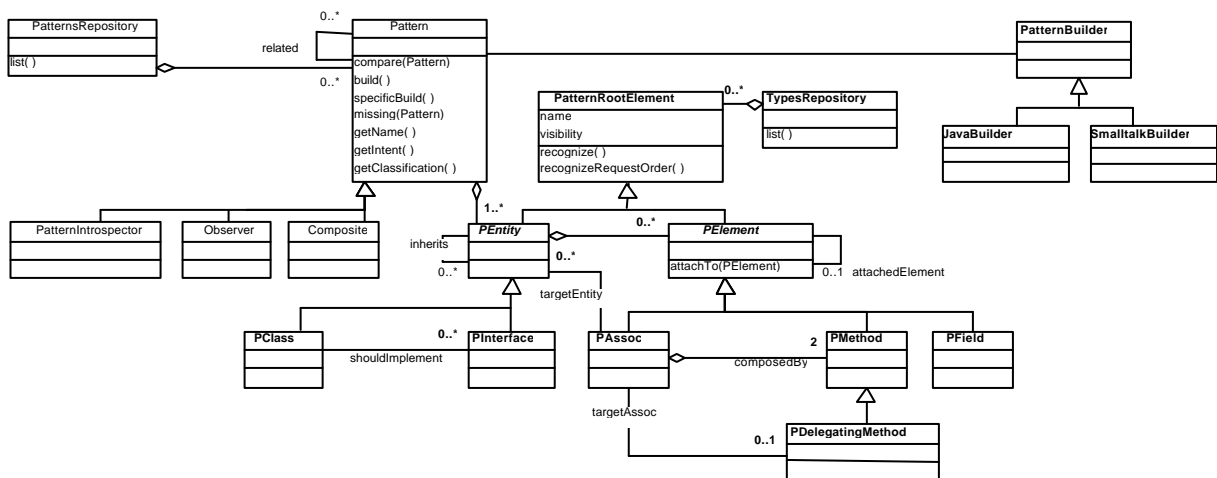


Figure 9.1 – Simplified UML Class-Diagram of the Meta-Model

The meta-model defines in details the semantics of a pattern. A pattern is composed of one or more classes or interfaces, instances of `PClass` and `PInterface` and subclasses of `PEntity`. An instance of `PEntity` contains methods and fields, instances of `PMethod` and `PField`. The association and delegation relationships are expressed as elements of `PEntity`. An association (class `PAssoc`) belongs to `PEntity` and references another `PEntity`. This relationship is simple since all the associations used in [GAM95] are binary and mono-directional. For example, an association that links a class `B` to a class `A` is defined using two instances of class `PClass`, `A` and `B`, and one instance of class `PAssoc`. The

instance of class `PAssoc` belongs to `A` and references `B`. Delegation is expressed in a similar way using the class `PDelegatingMethod`. For example, the delegation of the behaviour of a method `foo` of `A` to a method `bar` of `B` is realised using an instance of class `PDelegatingMethod`. The instance of class `PDelegatingMethod` belongs to `A` and references the method `bar` of `B`. The `PDelegatingMethod` object also references the association between `A` and `B` to deduce from it the cardinality of the message send: simple or "multicast".

The pattern meta-model is not an extension of an existing meta-model (such as UML). It was built from scratch using the Java programming language and the JavaBeans *formalism* (properties, idioms, and introspection).

An abstract model of a design pattern represents the design pattern *generic* micro-architecture and behaviour. An abstract model of a pattern is expressed using constituents of the meta-model. The abstract model of a pattern contains the design pattern micro-architecture and behaviour. An abstract model of a pattern may be reified: the design pattern abstract model is cloned and adapted to the current context (names, cardinalities, specificities of the code, etc) and becomes a concrete model: A first-class object that can be manipulated and about which we can reason. Reified design patterns are used to generate source code and to detect sets of entities with similar architecture and behaviour in source code. Patterns are instantiated and detected according to their abstract models, rather than according to external specifications.

The patterns meta-model suffer from some limitations: (1) It lacks expressiveness with respect to the most dynamic aspect of the application because it expresses relationships among entities, not among instances; (2) It needs to be further specialized to allow a finer-grain control over the constraints and the transformation rules.

9.2 Demonstration of the *COMPOSITE* pattern

This section shows how a pattern is instantiated through the previous meta-model. After instantiation, the pattern (or some variation of a pattern) can be detected in the user source-code. As we are mainly interested in the formalization, the detection details are out of the scope of this document. For further details on detection, please refer to [ALBa01] and [ALBb01]. The following figure presents the general procedure to instantiate a pattern.

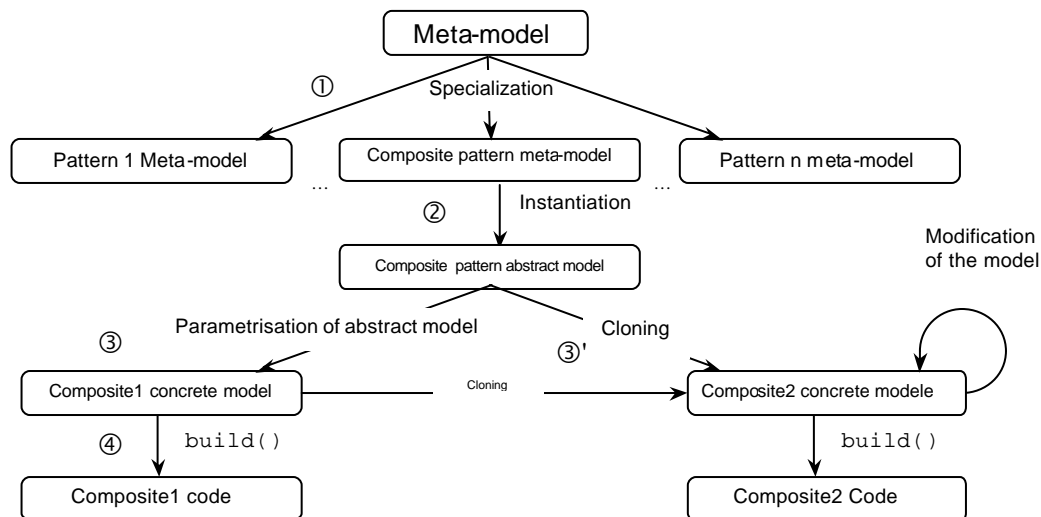


Figure 9.2 – *Process of Instantiation of the COMPOSITE Pattern*

The first step (represented by ① in figure 9.2) consists in specializing the patterns meta-model to obtain a meta-model with all the structural and behavioural needed constituents. In the case of the *COMPOSITE*, the meta-model previously presented is sufficient (note that the meta-model in figure 9.1 is a simplified one). However, it would be necessary to add a new *PEntity*, called *ImmutablePClass*, to implement a *COMPOSITE* in which leaves are immutable (In this case, a new subclass *ImmutablePClass* to the class *PClass* would be added).

The second step (represented by ② in figure 9.2) consists in the instantiation of the meta-model. The meta-model of the *COMPOSITE* is instantiated into an abstract model. This abstract model corresponds to a reification of the *COMPOSITE* and holds all the needed information related to the pattern. Thus, the *COMPOSITE* takes reality and becomes a first-class object. Since abstract models of design patterns are first-class objects, it is possible to reason about them and to use them as normal objects. Thus, it is possible to introspect them and to modify their structure and behaviour both statically and dynamically.

Figure 9.3 shows the structure of the *COMPOSITE* as in [GAM95]. From its structure and its notes, a new class diagram is obtained, where all informal indications are explicit. This class-diagram, shown in figure 9.4, is the abstract model of the *COMPOSITE* pattern.

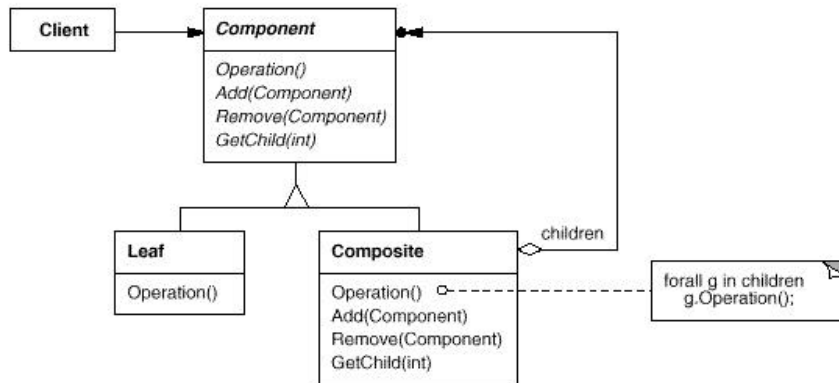


Figure 9.3 – Structure of the *COMPOSITE*

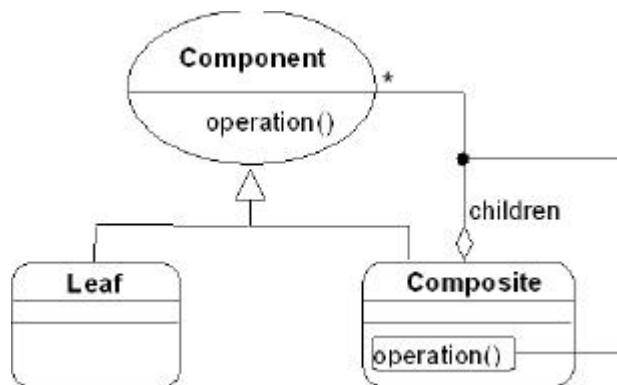


Figure 9.4 – *COMPOSITE* Abstract Model

The abstract model in figure 9.4 is expressed in a declarative manner into the class `Composite`. The table below shows the declaration of the `COMPOSITE`. This declaration is obtained from the abstract model. The way a pattern is later applied is deduced from its declaration. This adds an extra layer of abstraction and allows a same declaration to be used for both application and detection.

COMPOSITE Abstract Model Definition
<code>class Composite extends Pattern {</code>
<i>Declaration takes place in the Composite class constructor</i>
<code>Composite(...) { ...</code>
<i>Declaration of the “Component” actor</i>
<code>component = new PInterface("Component") operation = new Pmethod("operation") component.addPElement(operation) this.addPEntity(component)</code>
<i>Declaration of the association “children” targeting “Component” actor with cardinality n</i>
<code>children = new PAssoc("children", component, n)</code>
<i>Declaration of the “Composite” actor</i>
<code>composite = new PClass("Composite") composite.addShouldImplement(component) composite.addPElement(children)</code>
<i>The method “operation” defined into “Composite” actor implements the method operation of “Component” actor and is linked to its through the association “children”</i>
<code>aMethod = new PdelegatingMethod("operation", children) aMethod.attachTo(operation) composite.addPElement(aMethod) this.addPEntity(composite)</code>
<i>Declaration of the “Leaf” actor</i>
<code>leaf = new PClass("Leaf") leaf.addShouldImplement(component) leaf.assumeAllInterfaces() this.addPEntity(leaf) }</code>
<i>Declaration of specific services dedicated to the Composite pattern</i>
<code>...</code>
<i>For example, the service “addLeaf” to dynamically adds “Leaf” actor to the current instance of the Composite pattern</i>
<code>void addLeaf(String leafName) { PClass newPClass = new PClass(leafName) newPClass.addShouldImplement((PInterface)getActor("Component")) newPClass.assumeAllInterfaces() newPClass.setName(leafName) this.addPEntity(newPClass) }</code>

Table 6 – Declarative Description of the `COMPOSITE`

Abstract models are stored inside a pattern repository (class `PatternsRepository`, figure 9.1). This repository helps to access defined design patterns and to assess the relevance of the solution they represent.

The third (represented by ③ and ③' in figure 9.2) step consists in the instantiation of the abstract model into a concrete model. The concrete model represents the pattern applied to fit within a given application. In the example below, it defines a hierarchy of graphical components (as shown figure 9.5).

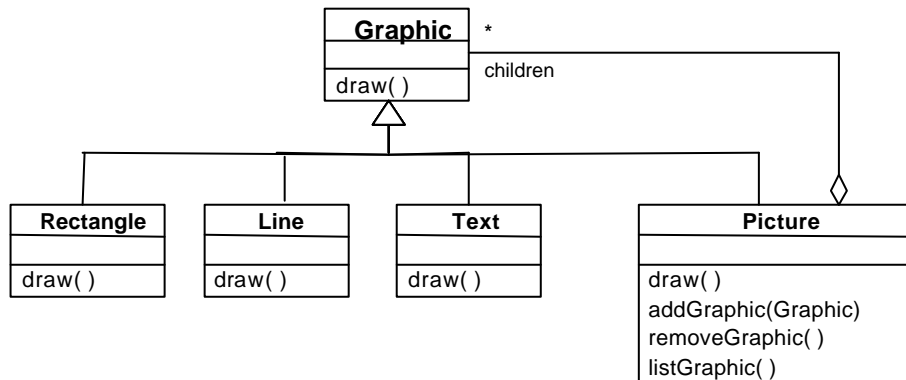


Figure 9.5 – UML Diagram of the Concrete Model

The instantiation of the abstract model is realised by instantiating the class `Composite` defined in ②. Then, each participants of the pattern is named to match the concrete application (figure 9.5). An alternative (③') to this parameterization is to clone an abstract or a concrete model to obtain a new concrete model. The advantage of such an operation is to allow fundamental modifications of the model, such as modifications that shortcuts the normal behaviour of the model, while ensuring the integrity of the initial model.

The following source code is given as an example since it can be deduced from the pattern abstract model and the context.

Declaration of a New COMPOSITE Concrete Model
<pre> Composite p = new Composite() p.getActor("Component").setName("Graphic") p.getActor("Component"). getActor("operation").setName("draw") p.getActor("Leaf").setName("Text") p.getActor("Composite").setName("Picture") p.addLeaf("Line") p.addLeaf("Rectangle") </pre>

Table 7 – One example of the COMPOSITE Concrete Model

The final step (represented by 4 in figure 9.2) consists in code generation. It is automatically performed once the concrete model currently manipulated receives the "build()" message (see figure 9.1). The *Java* source code obtained from the concrete model diagram (figure 9.5) is presented below:

```
/* Graphic.java */
public interface Graphic {
    public abstract void draw();
}

/* Picture.java */
public class Picture implements Graphic {
    // Association: children
    private Vector children = new Vector();

    public void addGraphic(Graphic aGraphic)
    {children.addElement(aGraphic);}

    public void removeGraphic(Graphic
                               aGraphic)
    {children.removeElement(aGraphic);}
    // Method linked to: children
    public void draw()
    {for(Enumeration enum =
         children.elements();
         enum.hasMoreElements();
         ((Graphic)enum.nextElement()).draw());
    }
}

/* Text.java */
public class Text implements Graphic {
    public void draw(){}
}

/* Line.java */
public class Line implements Graphic {
    public void draw(){}
}

/* Rectangle.java */
public class Rectangle implements Graphic {
    public void draw(){}
}
```

```
/* Graphic.java */
public interface Graphic {
    public abstract void draw();
}

/* Picture.java */
public class Picture implements Graphic {
    // Association: children
    private Vector children = new Vector();
    public void addGraphic(Graphic aGraphic)
    {children.addElement(aGraphic);}
    public void removeGraphic(Graphic aGraphic)
    {children.removeElement(aGraphic);}
    // Method linked to: children
    public void draw()
    {for(Enumeration enum = children.elements();
        enum.hasMoreElements();
        ((Graphic)enum.nextElement()).draw());
    }
}

/* Text.java */
public class Text implements Graphic {
    public void draw(){}
}
...
```

Figure 9.6 – *Example of a Source Code (in Java) Corresponding to the Concrete Model Presented*

9.3 Conclusions about PDL

This work created a meta-model that offers a way to define patterns at the design level. The structure and properties of a design pattern are defined using the constituents defined in the meta-model. The same abstract model can be used both for instantiation and detection, and due to this it is the most complete formalism. Additionally, an abstract model contains all the information related to its instantiation and detection, thus ensuring its traceability.

The main limitation of patterns instantiation concerns the integration of the generated code with the user's code. A solution proposed by the authors would be to transform the wanted implementation to fit the user's code. Instantiation would include the generation of the strict implementation of the pattern, and then this implementation would be integrated into the user's source code using source-to-source transformation. The authors are currently investigating the definition of a transformation engine (*JavaXL*) able to automatically transform user's source code according to a pattern declaration.

Another problem of this approach concerns the integration of the user's code specificity into the detection mechanism (see more details of it in [ALBa01]). For a complete discussion of the limitations regarding detection, please refer to the complete articles of the authors.

However, both limitations regarding instantiation and detection do not affect the formalization technique, where the abstract models correspond to the visual notation and the declarative descriptions correspond to the textual notations.

10. CONCLUSIONS

The application of design patterns can be decomposed in three distinct activities:

1. The choice of the right pattern, which fulfils the user requirements.
2. Its adaptation to use these requirements.
3. The production of the code required for its implementation.

These activities are challenges that still need to be solved. This document has present one step to achieve the solution through formalisms, which consequently generate tools to manipulate patterns.

The main advantage of the formalization of a pattern is that it removes ambiguity. Moreover, coherent specifications of patterns are essential to improve their comprehension, to allow formal reasoning about their relationships and properties, and to support and automate their application.

Unambiguous specification of designs is of paramount benefit when mining existing systems for new patterns. It enables patterns expression in computational form, permitting automated checking of designs for inconsistency or incompleteness. Case tool support of patterns allows the designer to work at the pattern level, rather at the level of individual classes. Consequently, the design is freed to work at a higher level of abstraction.

Tools can enable the designer to browse purely and precisely specified pattern catalogues, selecting design patterns that closely match the designer's requirements, adapting selected patterns via refinement, and combining and deploying these adapted patterns as appropriate for the application domain.

In this paper, many notations to formalize patterns were studied. It is clear that a great effort has been spent to increase the level of formalisms, in order to achieve the appropriate mechanisms used in tool support. It is important to emphasize the difficulty that arises from pattern formalization. In this report we show that all the existing formalisms are based mainly in *Participants*, *Structure* and *Collaborations* of the GoF catalogue, while the other aspects remains overlooked. This fact restricts the expressiveness of languages, because formal notations, as far as we know, cannot express all the statements included in natural languages.

Despite this limitation, formalisms are certainly a great issue for discussion. Their development and expansion will provide good tools to support patterns, and will make them easy to use and maintain. Moreover, a plenty of good methods and tools already exist, and certainly much more effort will be spent to improve them, or create new ones, in the coming years.

Only evolution can dictate the paths for design patterns and tools to support them, but surely research is necessary to improve usability and quality factors related to these fields.

11. REFERENCES AND BIBLIOGRAPHY

11.1 Books and Papers

- [ALB01a] Albin-Amiot, H.; Guéhéneuc, Y. *Meta-Modeling Design Patterns: Application to Pattern Detection and Code Analysis*. Workshop on Adaptive Object-Models and Metamodeling Techniques. ECOOP (European Conference on Oriented Programming), 2001.
- [ALB01b] Albin-Amiot, H.; Guéhéneuc, Y. *Design Patterns: A Round-trip*. Workshop for PhD Students in Object-Oriented Systems. ECOOP (European Conference on Oriented Programming), 2001.
- [ALE96] Alencar, P. S. C.; Cowan, D. D.; Lucena, C. J. P. *A formal Approach to Architectural design Patterns*. Proceedings of the 3rd International Symposium of Formal Methods Europe, 1996.
- [ASK97] Askit, M.; Matsuoka, S. Proceedings of the 11th European Conference on Oriented Programming – ECOOP '97. Lectures Notes in Computer Science No. 1241. Berlin: Springer-Verlag, 1997.
- [BLA91] Blaha, M.; Eddy, F.; Lorensen, W.; Premerlani, W.; Rumbaugh, J. *Object Oriented Modeling and Design*. Prentice Hall, 1991.
- [BOO94] Booch, G. *Object Oriented Analysis and Design with Applications*. Benjamin/Cummings, 1996.
- [BOO99] Booch, G.; Jacobson, I.; Rumbaugh, J. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [BOS95] Bosch, J. *Layered Object Model - Investigating Paradigm Extensibility*. Ph.D. dissertation, Department of Computer Science, Lund University, November 1995.
- [BOS96a] Bosch, J. *Relations as Object Model Components*. Journal of Programming Languages, Vol. 4, 1996.
- [BOS96b] Bosh, J. *Language Support for Design Patterns*. Proceedings TOOLS Europe '96, 1996.
- [BRÖ00] Brössler, P.; Prechelt, L.; Tichy, W. F.; Unger, B.; Votta, L. G. *A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions*. IEEE Transactions on Software Engineering, 2000.
- [BUD96] Budinski, F.J.; Finnie, M. A.; Vlissides, J. M.; Yu, P.S. *Automatic Code Generation from Design Patterns*. Object Technology, vol. 35, No. 2, 1996.
- [BUS96] Buschmann, R.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley and Sons, 1996.
- [COA91] Coad, P.; Yourdon, E. *Object Oriented Analysis*. Yourdon Press, 1991.
- [COP96] Coplien, J. O. *Code Patterns. The Smalltalk Report*. SIGS Publications, 1996.
- [EDE98] Eden, A. H. *Giving 'The Quality' a Name*. New York: SIGS Publications: Journal of Object-Oriented Programming, Vol. 11, No. 3, pp. 5-11, June 1998.
- [EDE98b] Eden, A. H.; Hirshfeld, H.; Yehudai, A. *Precise Notation for Design Patterns*. Technical report 327/98. Department of Computer Science - Tel Aviv University, 1998.
- [EDE00] Eden, A. H. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD Dissertation. Department of Computer Science - Tel Aviv University, 2000.

- [EDE01] Eden, A. H. *Formal Specification of Object-Oriented Design*. International Conference on Multidisciplinary Design in Engineering, November 2001.
- [FLO97] Florijn, G.; Meijers, M.; Winsen P.; *Tool Support in Design Patterns*. In: [ASK97].
- [GAM95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.
- [GIL98] Gil, J.; Howse, J.; Kent, S. *Constraint Diagrams: A step Beyond UML* IEEE Computer, 1998.
- [GRO01] Grogono, P.; Eden, A. H. *Concise and Formal Descriptions of Architectures and Patterns*. Technical Report. Department of Computer Science - Concordia University. 2001.
- [HED97] Hedin, G. Language Support for Design Patterns using Attribute Extension. Proceedings of Workshop on Language Support for Design Patterns and Frameworks, 1997.
- [HEL90] Helm, R.; Holland, I. M.; Gangopadhyay, D. *Contracts: Specifying Compositions in Object Oriented Systems*. Proceedings of OOPSLA '90, 1990.
- [KEN97] Kent, S. *Constraint Diagrams: Visualizing Invariants in Object-Oriented Models*. Proceedings of OOPSLA97. ACM Press, 1997.
- [KEN98] Kent, S.; Lauder, A. Precise Visual Specification of Design Patterns. Proceedings of ECOOP '98, 1998.
- [KLA96] Klarlund, N.; Koistinen, J.; Schwartzbach, M. I. *Formal Design Constraints*. Proceedings OOPSLA '96 Conference on Object-Oriented Programming Systems, Languages, and Applications. ACM SIGPLAN Notices, ACM Press, 1996.
- [LIE86] Lieberman, H. *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*. Proceedings of OOPSLA '86, 1986.
- [MEL92] Mellor, S.; Shlaer, S. *Object Lifecycles: Modeling the World in States*. Yourdon Press, 1992.
- [MIK98] Mikkonen, T. *Formalizing Design Patterns*. Proceedings of ICSE '98. IEEE Computer Society Press, 1998.
- [MIN94] Minski, N. H. *Law Governed Regularities in Software Systems*. Technical report LCSR-TR-220, Rutgers University – LCSR, 1994.
- [PAL95] Pal, P. P. *Law-Governed Support for Realizing Design Patterns*. Proceedings of 17th International Conference TOOLS, 1995.
- [QUI97] Quintessoft Engineering, Inc. *C++ Code Navigator 1.1*. 1997.
- [SOU95] Soukoup, J. *Implementing Patterns in Pattern Languages of Program Design*. Addison-Wesley, 1995.

11.2 Internet

[BEC87] <http://c2.com/ppr/about/author/kent.html>

[EDE02] <http://www.cs.concordia.ca/~faculty/eden/bibliography.html>