Towards a Mathematical Foundation For Design Patterns[i]

Amnon H. Eden[ii,iii]
Joseph (Yossi) Gil[iv]
Yoram Hirshfeld[v]
Amiram Yehudai[iii]

Abstract

We identify a compact and sufficient set of building blocks which constitute most design patterns of the GoF catalog: uniform sets of classes or functions, function clans, class hierarchies, and regularities (morphisms) thereof. The abstractions observed are manifested within a model *in symbolic logic and defined in LePUS, a declarative, higher order language. LePUS formulae concisely prescribe design patterns in a general, complete, and accurate manner. We provide a natural, condensed graphic notation for every LePUS formula and demonstrate how design patterns are faithfully portrayed by diagrams in this notation. We conclude by defining* refinement *(specialization) between patterns by means of predicate calculus and illustrate how the logical formalism of LePUS facilitates tool support for the recognition and implementation of design patterns.*

Keywords: Design patterns, theoretical foundations

1. Introduction

Design patterns [GoF 95; Coplien & Schmidt 95; Vlissides, Coplien & Kerth 96; Buschmann et. al 96; Martin, Riehle & Buschmann 97; Pree 94] are invariably specified by means of natural language narrative, concrete sample implementations in some programming language, and OMT [Rumbaugh 95] (or UML [Booch et. al 97]) diagrams. Each traditional mean of expression is either inherently ambiguous (e.g., natural language) or incomplete (diagrams in object notations, sample implementations). Not surprisingly, combinations of inadequate descriptions do not lead to a well defined abstraction, but, at most, to a "prototype" of one.

Our interest, however, is in a formal specification language whose statements *accurately*, *completely*, and *concisely* convey the regularities embodied in design patterns. Such language serves as a *metalanguage* compared to OOPLs, as it determines which *sets* of programs conform to each specification (rather than enumerating instances thereof.) Ideally, this language employs a limited

---

i `http://www.math.tau.ac.il/~eden/bibliography.html#towards_a_mathematical_foundation_for_design_patterns`

ii Written as part of Eden's Ph.D. research. Supported in part by a grant from the German-Israeli Foundation for Scientific Research and Development (GIF) and Israel Ministry of Science and Arts.

iii Computer Science Department, Tel-Aviv University.

iv IBM Research and Technion.

v Department of pure mathematics, Tel Aviv University.

vocabulary of entities and relations in stating these constraints, preferably defined as constructs in an established branch of mathematical such as symbolic logic.

Observations

A prerequisite to any specification language is the observation of a compact set of building blocks of design patterns and the fundamental relations among them. The significance of these abstractions stands independently of the language by which they are conveyed.

We observed such abstractions as follows:

1. **Uniform sets** of functions or classes of arbitrary order (i.e., sets of sets of sets…) capture the *participants* specified in every pattern (Definition II).

2. **Inheritance class hierarchies** are treated as monoliths (e.g., regardless of inheritance levels involved) and particular morphisms are defined thereof (Definition IX).

3. **Clans**, or functions redefined throughout class sets and inheritance hierarchies are captured (Definition VII).

4. The structure and behavior embodied in most design patterns is captured by a small set of **ground relations** between the participants, 8 of which are listed in Appendix A, including: "*c is the first argument of f*", "f_1 *invokes* f_2", "f_1 *forwards the call to* f_2", "*f is defined in c*", and others.

   Although this set is subject to extensions, every ground relation is required to have a canonical, straight-forward implementation in most OOPLs. This simplifies the mapping of a LePUS formula to an OOPL and prohibits absurdities such as "*x is an observer*".

5. All **correlations** of interest between *functions*, *classes*, and *hierarchies*, and sets thereof <u>of any dimension</u> (Definition II), extend **systematically** from the *ground relations* by two generalizations thereof: *total* and *regular* (Definition III). For instance:

   - "*each observer holds a reference to the subject*" (Predicate 9, Figure 8) is a *total* "*Reference-To-Single*" relation of dimension 1
   - "*each function of Factory-Methods creates an object of a class in the Products hierarchy*" (Predicate 2, Figure 5) is a *regular* "*Creation*" relation of dimension 1
   - "*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*" (Predicate 7, Figure 6) is a *regular* "*Production*" relation of dimension 2

6. Finally, converging *regular relations* (isomorphisms) **commute** (Definition V) in most situations.

LePUS

LePUS (*LanguagE for Patterns Uniform Specification*) is a fragment of higher order monadic logic (HOML) defined in section 2. Every formula in LePUS has this form:

1. List of *participants* (that are either *classes*, *functions*, or *hierarchies*), and

2. List of the *relations* imposed amongst.

Accordingly, LePUS formulae contain a declaration of typed *variables* and a conjunction of *relation* predicates (Formula 1).

LePUS has a natural graphic representation (section) such that every formula in LePUS is equivalent to a condense and unelaborated diagram. Unlike OMT, LePUS diagrams manifest design patterns in a manner that is general, complete, and unequivocal, as demonstrated by portraying several patterns of the [GoF 95] catalog in section 4. Moreover, in section 5.1 we demonstrate how predicate calculus can be used to establish and verify (or refute) relationship between design patterns, such as "*pattern* π_1 *refines pattern* π_2", or "*pattern* π_1 *is a component of pattern* π_2". Finally, the prospects of tool support are discussed in section 5.2 and the implementation of a prototype is demonstrated.

Object Notations as Means of Pattern Specification

There is a difference between a concrete software system and a design pattern:

- An O-O design process results in a concrete system, whose description comprises classes, instances and relations between them, intended toward a solution of a specific problem.
- A design pattern reflects a generic *aspect of* 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 and other elements which do not conform to any known pattern.

Object notations [Booch 94; Rumbaugh et. al 91; Booch, Rumbaugh, & Jacobson 97] were devised to facilitate the O-O design process and to report its results, thereby delineating a concrete software system. On the other hand, no object notation was ever meant to account for *sets* of programs as the specification of design patterns requires[i]. For instance, functions (*methods*) are defined as "second rate" elements, and relations among them are undefinable[ii], thus conveyed through informal cues ("notes"). Finally, neither of these notations was ever granted with precise semantics[iii].

For these reasons, OMT diagrams (as used in the [GoF 95] catalog) do not incorporate variable symbols. Each such diagram may account for, at most, a particular instance of a pattern. It cannot specify which modifications to the instance depicted are "legal", that is, preserve the "identity" of the pattern of interest, and which violate it. Furthermore, experience proves that even with respect to the particular instance of a pattern that is depicted in each, OMT class and object diagrams invariably fail to capture significant information about the implementation of a pattern even *within* the very specific instance depicted (not to mention a generic description thereof.) The missing information is also conveyed using informal cues, sample implementations, and mainly using elaborated descriptions in English.

A possibility arises to apply UML [Booch, Rumbaugh, & Jacobson 97] as a *metalanguage* rather than a language of concrete programs, much in the way UML itself is reflexively defined. One may have a UML like diagram that describes how instances of the meta-classes Class, Method, Variable, etc. are related to each other, therefore specifying a pattern through its participants

---

i Although the UML document [Booch, Rumbaugh, Jacobson 97] demonstrates how the notation gives rise to a particular implementation of a pattern.

ii except for a rudimentary specification, namely its signature and enclosing class.

iii Notwithstanding the beginning of research towards the formalization of UML generically. Note that a UML like language assumes that there are many different *ground relations* ("associations") between entities (classes) defined differently in any different system. In contrast, there is a bounded and small number of kinds of associations between design entities such as method and class, which are typically built into the system.

Nevertheless, object notations exclude sets of higher order, which are fundamental to the specification of design patterns, and in particular do not account for morphisms and correlations thereof (such as 1:1 and onto correlations between sets of any order.) Thus, none of UML-like *associations* and *cardinalities* are sufficiently expressive even as part of a metalanguage.

Related Work

Florijn, Meijers, and van Winsen [97] present a prototype for a tool that performs reverse engineering 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 labeled according to their roles. By this approach, pattern's roles are mere strings, which restricts the reasoning that can be made on such a model. This details of the implementation of this tool, however, are not given in full, and it cannot be determined whether the representations are in fact equivalent to semantic nets.

Eden, Gil and Yehudai [97c] define pattern as an algorithm, specifying the sequence of steps in its application, defined in pseudo-code specification. While we found algorithmic abstractions very natural for tool support, a declarative specification is more comprehensible and more easily subjected to formal reasoning. Finally, algorithmic specifications did not easily translate to graphic representations which were our preferred specification style.

The LayOM [Bosch 96] extends the classic object model with the concepts of *states* and *layers* to support the specification of patterns. Alencar, Cowan and Lucena [96] propose an environment that comprises "Abstract Data Views" and "Abstract Data Objects" (ADV/ADO), specified in a specialized "scheme" language. The mapping of the fundamental constructs of both specialized models onto those of common OOPLs is very elaborated and not self-sufficient.

Helm, Holland and Gangopadhyay [90] defined "Contracts", an extension to 1st order logic with representations for function calls, assignments, and an ordering relation between them. The "behavioral compositions" described do not address relations between classes that are a fundamental part of design patterns and the resulting specifications are also detailed and elaborated.

Hedin [97] proposes to extend the base grammar of a given OOPL language with attributes that can specify the roles of collaborators in design pattern. The specifications of a pattern in this scheme are tightly coupled with the grammatical rules defining the base language.

Pal [95] demonstrates how the behavior indicated by a pattern can be enforced by *Darwin-E* environment [Minski 94], which detects violation of the specifications by means of static analysis. Similarly, Klarlund, Koistinen and Schwartzbach [96] present a specialized language of parse trees, *CDL*, that can be used to validate design constraints. *CDL* formulae are proposed as means to express constraints over the behavior and relations of collaborators in a design pattern.

Budinsky, Finnie, Vlissides, and Yu [96] present a tool that supports the implementation of design patterns by generating code. As a specification language serves a special purpose, ad-hoc scripting language named COGENT, whose imperatives produce statements in the target OOPL. Quintessoft Inc. [97] distributes a CASE tool that generates C++ source code for a number of the GoF [95] patterns, but whose specifications are hard coded in the environment.

2. The Language

LePUS formulae manifest design patterns in the form of logic statements. More specifically, patterns are transcribed to formulae while every program is assumed to be represented as a *model*. Given the transcription of a pattern π to a LePUS formula ψ, we say that a program p **conforms** to π iff the *model* of p satisfies ψ.

LePUS is a small subset of higher order monadic logic (HOML) [Barwise 76]. Nonetheless, HOML is too rich and expressive a language for our purposes. Thus, our contribution lies primarily in recognizing a well defined subset of HOML that is only a fraction thereof, yet it effectively accounts for most patterns of interest.

LePUS incorporates the abstractions we have recognized as essential and repeating elements of design patterns, and the result is as expressive with respect to design patterns.

Most elements of LePUS are provided with a natural graphic representation. Thus every *well formed diagram*, as described in section 3.1, is equivalent to a particular formula in LePUS.

2.1 Models

A program[i] is represented as an *object oriented structure* or *model*: a collection of ground entities (*atoms*), namely *classes* and *functions* (also *methods*, *routines*) 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 will most often include unary relations such as "c is a class", "f is abstract", and the binary relations "c_1 inherits from c_2", "f is defined in c", "c is the first argument of f", and other relations as necessary. 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 our model.

We designate our two-sorted universe of discourse by the symbol $\mathbb{P}$, the set of the ground entities that are classes as $\mathbb{C}$, and the set of the ground entities that are functions by $\mathbb{F}$. Both $\mathbb{F}$ and $\mathbb{C}$ are referred to as *types* or *domains* with respect to the variables and relations in LePUS.

> Definition I: A model M is a tuple $\langle\mathbb{P}, \mathbb{R}\rangle$ where $\mathbb{P}$ is a universe of *ground entities*, each is either a *function* or a *class*, and $\mathbb{R} = R_1,\ldots R_n$ is the set of relations amongst.

Language Mappings

For some OOPLs (O-O programming languages) and for some relations there exists a well defined and simple mapping between the linguistic construct and the relation in LePUS. For instance, a *Reference-To-Single* relation is manifested in Eiffel as an *attribute*, a definite syntactical construct. This, however, is not true for every relation or every programming language. A relation can have more than one syntactical form (such as *Inheritance* in Java) or several implementations (such as *Creation* in C++).

In principle, we require a canonical, "simple" implementation possible for every relation of $\mathbb{R}$, and therefore rule out absurdities such as the unary relation "*X is an* O*BSERVER*".

[i] Or a class library for that matter

As design patterns are generalizations, however, a discussion about them is essentially about abstractions. An abstraction is essentially about ignoring some details. In LePUS we have made a compromise between the technically detailed level of the programming language and the "abstractness" of natural language. We consider relations such as "*f creates an object of c*" and "c_1 *inherits from* c_2" as essential abstractions of the right level for our discussion and prove them to be sufficient building blocks for the specification of many design patterns (see section 4 and [Eden, Hirshfeld & Yehudai 98a]) and for establishing relations between design patterns (5.1).

Nevertheless, to render the examples and case studies comprehensible we describe the intent behind the relations mentioned in Appendix A.

Dimensions

Ground entities of $\mathbb{P}$ are called *entities of dimension 0*. We refer to a set of classes (functions) as a *class* (*function*) *of dimension* 1, a set of sets as a *class* (*function*) *of dimension* 2, and so forth. Formally:

> Definition II: 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

Note we ignore non-uniform sets, namely, those which contain entities of different types or dimensions.

The domain of classes (functions) of dimension 1 is denoted $2^{\mathbb{C}}$ ($2^{\mathbb{F}}$), of dimension 2 as $2^{2^{\mathbb{C}}}$ ($2^{2^{\mathbb{F}}}$), and so forth.

2.2 Formulae

A *formula* in LePUS consists of the following building blocks, to be defined below:

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* (section 2.3)
2. **Relation** symbols, standing for
   - *ground* relations, corresponding to those in $\mathbb{R}$, including *transitive* relations
   - *generalized* relations, which derive systematically from the *ground* relations (Definition III)
   - *commuting* relation (Definition V)

A *formula* in LePUS has the following form:

> *Formula 1:*
>
> $$\exists(x_1, \ldots x_n) : \bigwedge_i P_i$$

Where P_i are predicates of the form $\Re_i(y_{i_1},\ldots y_{i_{n_i}})$, $\Re_i$ are relations of the above types, and $x_1,\ldots x_n$ are all the free variable in P_i.

Variables

To deal with higher dimension entities, LePUS incorporates higher order, typed variables; C^1 (F^1) vary over sets of functions (classes); 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).

Hierarchy variables shall be added later to represent *inheritance class hierarchies*.

Ground Relations

A relation symbol for a relation in $\mathbb{R}$ may be declared on ground variables. Also, for every binary relation β of $\mathbb{R}$ we define a transitive counterpart β^+ as the transitive closure of β. We denote *ground relations* the relation symbols for the relations in $\mathbb{R}$ and their transitive counterparts (whenever definable).

For instance, *Abstract*(c) is a ground relation, and is satisfied by a ground class of $\mathbb{P}$ if it is abstract. As another example, the (possibly indirect) inheritance relation between *concrete-class* and *abstract-class* transcribes to

Predicate 1:

$$Inheritance^+(concrete\text{-}class, abstract\text{-}class)$$

Other relations include *Class*(x) and *Defined-In*(f,c), for example.

Generalized Relations

All relations in LePUS derive systematically from the ground relations in a limited number of methods, as follows. We define below the indication of every *generalized relation* when applied to variables of dimension 1, but each definition extends naturally to higher dimensions.

Let α denote a ground unary relation, β a ground binary relation. Let lowercase w, v, $v_1,\ldots v_n$ designate *ground* variables, uppercase V, W designate 1-dimensional variables.

Definition III: The following ***generalized relations*** are admitted:

(***unary***) $\alpha(V) \overset{def}{=} \forall v \in V : \alpha(v)$

(***total***) $\beta^{\rightarrow}(V,W) \overset{def}{=} \forall v \in V\, \exists w \in W : \beta(v,w)$

(***regular***) $\beta^{\leftrightarrow}(V,W)$ indicates that β is an *invertible* function (1:1 and *onto*) from V to W

Total relations apply to variables of different dimensions with the obvious designation (Appendix B).

For instance, the *FACTORY METHOD pattern* requires that every function of the set *Factory-Methods* creates exactly one class of *Products*, and that every *Products*' class is created by exactly one *Factory-Methods*' function, we specify a regular *Creation* relation between the sets, as in

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

Predicate 2 is portrayed in Figure 4. We specify dimension and type of *Factory-Methods* and *Products* using existential quantifiers which appear outermost in the complete formula:

> *Formula 2:*
>
> $\exists\ Factory\text{-}Methods \in 2^{\mathbf{F}}, Products \in 2^{\mathbf{C}} : \langle Predicate\ 2 \rangle$

As another example, the predicate $Inheritance^{+\rightarrow}(Nodes, root)$ indicates an inheritance class hierarchy whose base class is *root* and each *Nodes*' class inherits (possibly indirectly) from *root*.

Restrictions on relations' domain are specified through exclusive relations:

> Definition IV: For a generalized relation $\mathfrak{R}$ we define:
>
> $$\mathfrak{R}(V!, W) \stackrel{def}{=} \mathfrak{R}(V, W) \wedge [\forall v : \mathfrak{R}(v, w) \wedge (w \in W) \rightarrow (v \in V)]$$
>
> We say that $\mathfrak{R}$ is ***exclusive*** to V with respect to W.

The dual case is defined for $\mathfrak{R}(V, W!)$, and $\mathfrak{R}(V!, W!)$ is equivalent to $\mathfrak{R}(V!, W) \wedge \mathfrak{R}(V, W!)$.

Access restrictions can be imposed by specifying *exclusive* relations. For instance, the *FAÇADE* pattern (Figure 9) requires that instances of *Subsystem-Classes* are created only by functions defined in the *façade* class (denoted "*Creators*") <u>and no others</u> [GoF 95]; this requirement is manifested by a total *Creation* relation that is exclusive to *Creators* with respect to *Subsystem-Classes*. This restriction is specified in Predicate 3, also portrayed in Figure 9:

> *Predicate 3:*
>
> $Creation^{\rightarrow}(Creators!, Subsystem\text{-}Classes)$

Commutativity

Commutative diagrams are common in category theory [Pierce 93] to indicate that the depicted morphisms *commute*, i.e., they are equal in the category of reference. We are interested in the requirement that a chosen list of *regular* relations *commute* (overlap) in a given domain.

> Definition V: Given the variables V, W, and the binary regular relations $\beta_1^{\leftrightarrow}$, $\beta_2^{\leftrightarrow}$ (each may be a composition of regular relations), we define the ***Commute*** relation as follows:
>
> $Commute_{\beta_1, \beta_2}(V, W)$ indicates that β_1 and β_2 are equal in V, W

Commute may apply to more than two relation symbols and more than two variables: Representing *regular relations* as edges and variables as vertices (section 3.1), Definition V is generalized by the condition that for every pair of vertices V and W, all the paths in the graph from V to W are equal, in the sense that each path in the graph determines a (composition of) relation and these relations are equal in V and W.

To illustrate *commutativity* we reuse the example that originated from the *FACTORY METHOD* [GoF 95]. We require a 1:1 and onto correlation between *Factory-Methods* and *Products* such that (i) object(s) of each *product* class are being created in exactly one *Factory-Method*, and (ii) the return type of each *Factory-Method* equals the *product* class of the object created thereof.

Predicate 2 transcribes part (i) of this requirement. To specify part (ii) we add

> *Predicate 4:*
>
> $$\textit{Return-Type}^{\leftrightarrow}(\textit{Factory-Methods}, \textit{Products})$$

But requirement (ii) is not satisfied yet, as neither Predicate 2 nor Predicate 4 force the return type of each *Factory-Method* to equal the type of objects created thereof. This property is guaranteed by additionally requiring the *Return-Type* and *Creation* relations to *commute* in the sets *Factory-Methods* and *Products*, connoted as

> *Predicate 5:*
>
> $$\textit{Commute}_{\textit{Return-Type},\textit{Creation}}(\textit{Factory-Methods}, \textit{Products})$$

The conjunction of Predicate 2 with Predicate 4 and Predicate 5 is portrayed in Figure 4.

To summarize, a *well formed predicate* combines a relation symbol ρ (either *ground*, *generalized*, or the *Commute* relation) declared on variables of number, type, and dimension as allowed by ρ. Finally, a *well formed formulae* consists only of well formed predicates. Formally:

> Definition VI: A formula $\exists(X_1, \ldots X_n): \bigwedge_i P_i$ is ***well formed*** iff all predicates P_i are well formed.

2.3 Auxiliary Abstractions

In addition to *higher dimensional functions* and *classes*, we introduce additional abstractions into LePUS such as *hierarchies* and *clans*, and the *product* shorthand. Neither of these abstractions effectively changes LePUS as each is defined by means of constructs defined above.

Function Families

A set of functions with identical signature[i] is an abstraction of special interest in OOP: such sets are commonly designed to allow *dynamic dispatch*, that is, deferral to execution time of the choice of the function to dispatch. We call such a set of functions a *clan* with respect to the classes within they are defined.

To formally define *clans* we must assume that *Same-Signature*(f_1,f_2) is either a ground relation or is a syntactic sugar for a conjunction of predicates that equate the return types and the number and types of arguments of f_1 and f_2:

---

[i] A "signature" of a function here is referred to as its return type and its arguments' types and number. We ignore minor differences between different OOPLs that allow various relaxations on complete identity of the signatures.

Definition VII: F^d is a ***clan*** in C^d iff F^d consists of functions of identical signature, each of which is defined in a different class of C^d. Formally:

$$Clan(F^d,C^d) \overset{def}{=} Defined\text{-}In^{\leftrightarrow}(F^d,C^d) \wedge [\forall F_1^{d-1},F_2^{d-1} \in F^d : Same\text{-}Signature^{\rightarrow}(F_1^{d-1},F_2^{d-1})]$$

Definition VIII: F^{d+1} is a ***tribe*** in C^d iff every function $F^d \in F^{d+1}$ is a *clan* in C^d.

Note that the elements of a *tribe* do not necessarily have the same signature, as elements of a *clan* do. In the case where F and C are of dimension 0, $clan(f,c)$ simply means that f is defined in c. In the general case, F^d is a clan in C^d iff the union of classes in F^d is a clan in C^d.

For instance, since an *update* operation is defined in each observer class of the *OBSERVER* pattern (Figure 8), *update* is a *clan* of dimension 1 in the *Observers* set of classes. Similarly, *factory-method* of the *FACTORY METHOD* pattern (Figure 5) is manifested as a *clan* in the *Creators* set.

For instance, each *Visit* clan is defined for every class of the *Elements* hierarchy of the *VISITOR* pattern (Figure 7). Therefore the set of all *Visit* functions forms a *tribe* in the *Visitors* hierarchy. Similarly, each *Creators*' function of the *ABSTRACT FACTORY* pattern (Figure 6) produces objects of different class hierarchy of *Products*, thereby forming a *clan* per *Products* hierarchy. The set of all *Creators* clans gives rise to a *tribe* in *Factories*, giving rise to the predicate:

Predicate 6:

$Tribe(Creators, Factories)$

Hierarchies

Inheritance class hierarchies (or *hierarchies*) are compositions of particular interest in OOP, hence they are introduced as encapsulated abstractions. A hierarchy is defined as set of classes that consists of (i) an abstract *root* (base) class and (ii) a set of the classes that constitute the rest of the hierarchy (*Nodes*), each of which must (possibly indirectly) inherit from *root*. Formally:

Definition IX: A ***hierarchy*** is a set of ground classes that constitutes a ground class h_r^0 and a set of remaining elements h_N^1, that conforms to the following conditions:

- $Inheritance^{+\rightarrow}(h_N^1, h_r^0)$
- $Abstract(h_r^0)$

We add that h_r^0 is the only class in the hierarchy that follows the two conditions.

We extend our model $M = \langle \mathbb{P}, \mathbb{R} \rangle$ to account for *hierarchy* entities, whose set is denoted $\mathbb{H}$, giving: $\mathbb{H} \subset 2^{\mathbb{C}}$. Similarly to the other domains, a hierarchy entity has dimension 1 (since it is a class of dimension 1), and a set of *hierarchies* of dimension d makes a hierarchy of dimension d+1. We represent the extended model by the tuple $\langle \mathbb{P}, \mathbb{R}, \mathbb{H} \rangle$. We add to our language *ground hierarchy variables* which range over the domain $\mathbb{H}$, denoted by the lowercase letters h, $h_1,\ldots$, and define their dimension as 1. Consecutively, hierarchy variables H^d of dimension d range over hierarchies of dimension d. Thus, the two statements in Formula 3 are equivalent:

Formula 3:

$$\exists Observers \in \mathbb{H}$$

$$\exists Observers_N \in 2^{\mathbb{C}}, Observers_r \in \mathbb{C} :$$
$$Inheritance^{+\rightarrow}(Observers_N, Observers_r) \wedge Abstract(Observers_r)$$

Hierarchy variables are treated in *regular* relations as 1-dimensional classes.

For instance, the *FACTORY METHOD* pattern (Figure 5) requires that objects of each class of *Products* are "produced" (see *Production* in Appendix A) by a different "factory method", each of which is defined in a different "creator" class. We can define both *Creators* and *Products* as hierarchy variables, transcribing the complete pattern as follows:

Formula 4:

$$\exists\ factory\text{-}method \in 2^{\mathbb{F}}, Creators, Products \in \mathbb{H} :$$
$$clan(factory\text{-}method, Creators) \wedge$$
$$Production^{\leftrightarrow}(factory\text{-}method, Products) \wedge$$
$$Return\text{-}Type^{\leftrightarrow}(factory\text{-}method, Products) \wedge$$
$$Commute_{Return\text{-}Type, Creation}(factory\text{-}method, Products)$$

Note that Formula 4 (portrayed in Figure 5) is a *specialization* of (the formula of) Figure 4. In section 5.1 we formalize this notion of *specialization* by the definition of *refinement*.

A regular relation also manifests the correlation which appears in the *ABSTRACT FACTORY* pattern (Figure 6) between the *Creators* tribe (Predicate 6) and *Products*, a set of hierarchies, thereby indicating a 1:1 and onto relation between each *Creators*' *clan* and a hierarchy in *Products*:

Predicate 7:

$$Production^{\leftrightarrow}(Creators, Products)$$

(*Production* relations are described in Appendix A.)

Hierarchy variables are not allowed in *total* relations but only in *H-total relations*, a variation thereon. Intuitively, in an *H-total* relation, a ground hierarchy variable stands for the hierarchy's *root* if positioned as the argument for the $\exists$ quantifier in the *total* counterpart, but is interpreted as the hierarchy's set of *Nodes* where as the argument for the $\forall$ quantifier.

Below we denote the "root" class of a hierarchy h by $root(h)$ and the set of node classes by $Nodes(h)$. Note that this writing is *not* part of LePUS.

Definition X: The ***H-total generalization*** of a binary ground relation β is a relation $\beta^{\rightarrow\mathrm{H}}$ with the following indication, depending on its arguments:

$\beta^{\rightarrow\mathrm{H}}(V,h)$ means $\beta^{\rightarrow}(V, root(h))$

$\beta^{\rightarrow\mathrm{H}}(h,V)$ means $\beta^{\rightarrow}(Nodes(h), V)$

$\beta^{\rightarrow\mathrm{H}}(h_1,h_2)$ means $\beta^{\rightarrow}(Nodes(h_1), root(h_2))$

The combination of hierarchy variables with ground variables in *H-total* relations is defined in Appendix B.

For instance, the formula of the *OBSERVER* pattern (Figure 8) portrays the ground class variables *subject*, *concrete-subject*, and the hierarchy variable *Observers*. One predicate portrayed is

Predicate 8:

$$Reference^{\rightarrow H}(subject, Observers)$$

indicating that *subject* has a reference (data member) of type *root*(*Observers*). An another predicate portrayed in the diagram is

Predicate 9:

$$Reference^{\rightarrow H}(Observers, concrete\text{-}subject)$$

indicates that each class of *Nodes*(*Observers*) holds a reference of type *concrete-subject*.

3. The Graphical Notation

In order to allow highly concise expressions for design patterns and to promote the readability of formulae, we have granted graphic representation to a selection of LePUS' elements. Section 4 demonstrates the use of the graphic language to effectively account for versatile design patterns.

Diagrams

Each *well formed diagram* in LePUS is equivalent to a *well formed formula*. Figure 1 lists the figures of the graphical notation and the linguistic construct delineated by each, where

- **icons** stand for variables
- **unary** relation marks stand for unary relations applied to the designated variable
- **arcs** stand for binary relations applied to the variables they connect
- **commute** designation(s) circumscribe the relations and domains (sets) of commuting

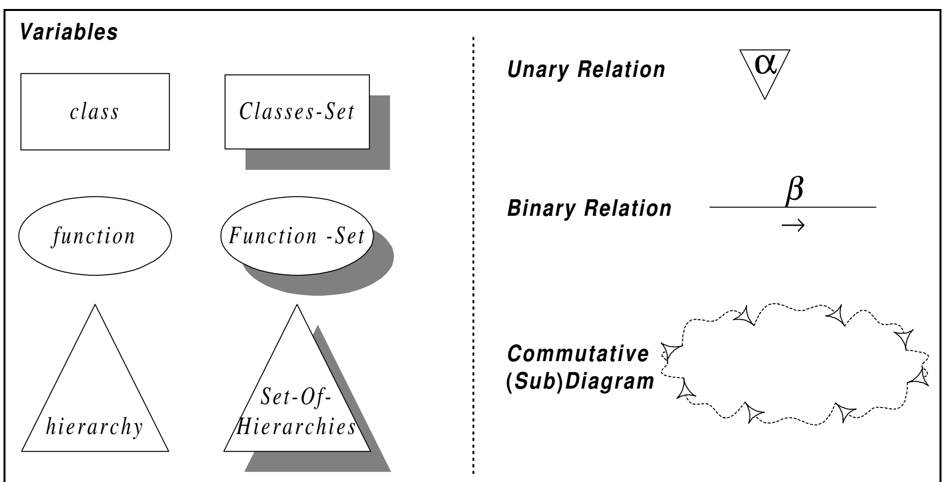

Figure 1: Basic graphic symbols of LePUS

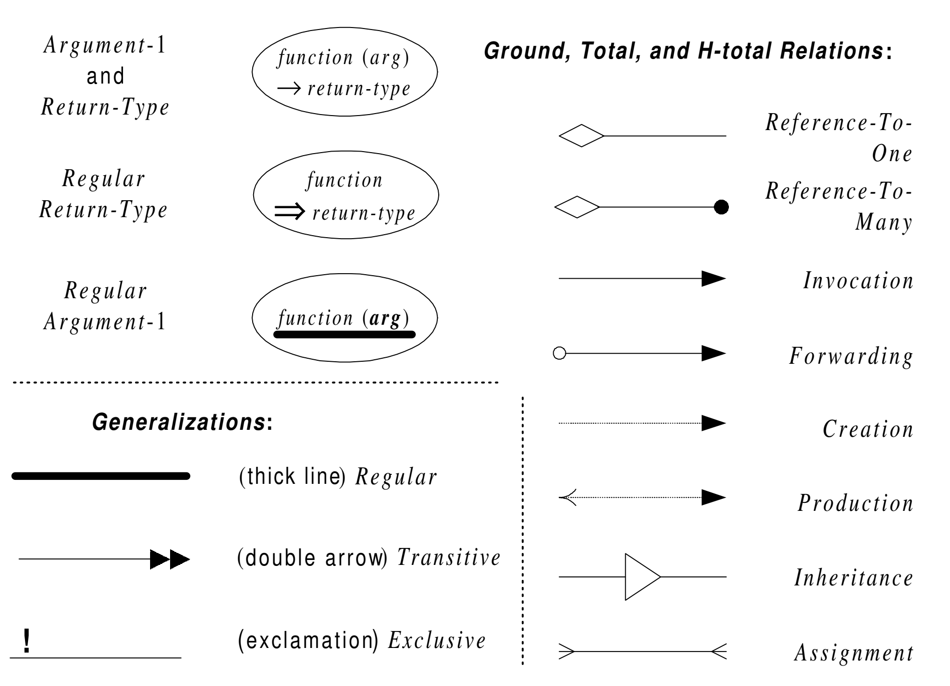

Figure 2: Graphic synonyms for predominant relations. The direction of the binary relation edges is left to right.

A ***diagram*** in LePUS is a graph whose vertices are icons (variable), possibly adorned with a unary relations, and whose edges are each labeled by a (possibly generalized) binary relation. An edge ρ, connecting vertices v_1, v_2, gives rise to the predicate: $\rho(v_1, v_2)$. A *well formed diagram* gives rise only to *well formed formulae* (Definition VI). Thus, for instance, *Inheritance* edges must connect class icons.

Graphic Synonyms

We define special edge styles for relations of predominant interest (e.g., *Inheritance*) and *relation modifiers* adornments standing for *generalized relations*. The interpretation of each of the relations of Figure 2 appears in Appendix A.

Generalized relations are depicted as *relation modifiers*, adornments that are added to the arrows. Thus, for instance, the relation β^+ is represented by adding a double arrow head to the edge (or multiplying the arrow head if there exists one) which stands for β. Similarly, *regular* generaliza-

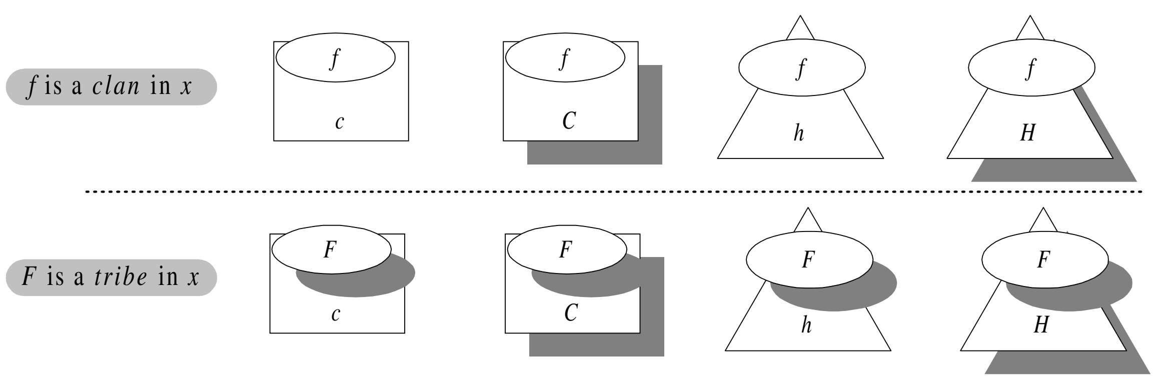

Figure 3: Superimpositions represent clans *and* tribes

tion are signified by widening the edge's line, and exclusive relations are marked by adding the exclamation mark next to the exclusive variable.

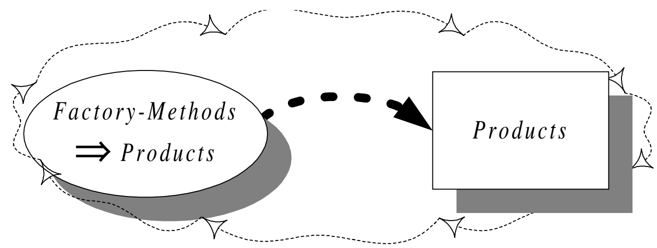

Figure 4: A LePUS diagram portraying Predicate 2, Predicate 4, and Predicate 5.

A segment of a diagram that is also a *well formed diagram* may be circumscribed by the *commutative* designator, indicating that the regular relations thereof *commute* over the indicated domains (variables). Figure 4 demonstrates how commuting regular relations are delineated following the sample predicates of sections 2.2.

4. Case Studies

In this section we set forth the diagrams of selected design patterns from the [GoF 95] catalog. The selection of patterns to be transcribed to LePUS was made to demonstrate versatile applications of the language. The diagrams of 9 additional design patterns is available online from [Eden, Hirshfeld & Yehudai 98a].

The *FACTORY METHOD* pattern is manifested by 4 predicates portrayed in Figure 5:

1. To indicate the correlation between, *factory-methods* are a *clan* (Definition VII) in the *Creators* hierarchy
2. The regular *Return-Type*$^{\leftrightarrow}$ relation (Predicate 4, section 2.2)
3. *Production*$^{\leftrightarrow}$ (*Factory-Methods*, *Products*)
4. *Commutativity* on *Return-Type*$^{\leftrightarrow}$ and *Production*$^{\leftrightarrow}$ (Predicate 5).

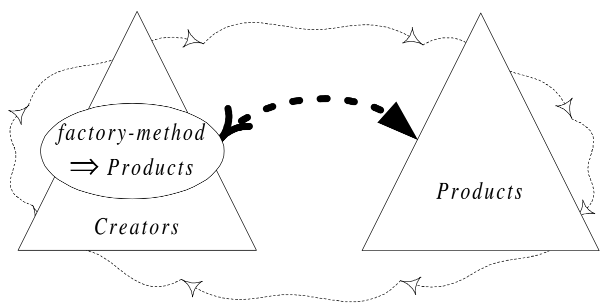

Figure 5: LePUS diagram of the FACTORY METHOD pattern

The *ABSTRACT FACTORY* (Figure 6) differs from the *FACTORY METHOD* only in the dimensions of *Products* and *factory-methods* (called *Creators* here as in [GoF 95]). Nonetheless, the same regular *Return-Type*$^{\leftrightarrow}$ and *Production*$^{\leftrightarrow}$ relations equally apply! As a result of increasing the dimension of *Creators*, it becomes a *tribe* (Definition VII) in the *Factories* hierarchy.

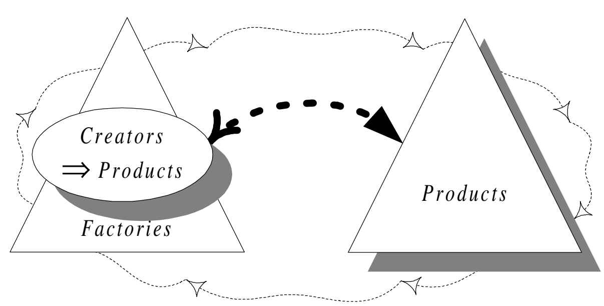

Figure 6: LePUS diagram of the ABSTRACT FACTORY pattern

The diagram of the *VISITOR* pattern (Figure 7) portrays one *H-total* relation: $Argument_1^{\leftrightarrow}(accept, Visitors)$, and four regular relations:

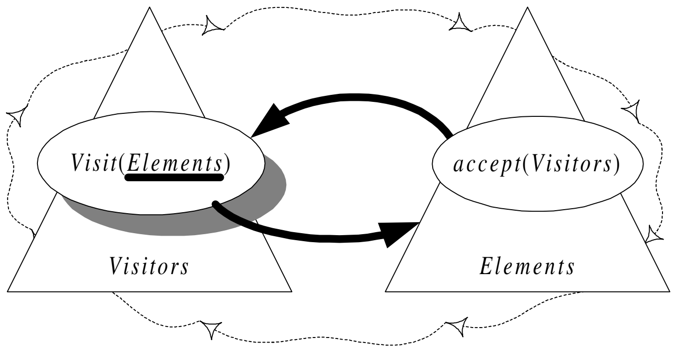

Figure 7: LePUS diagram of the VISITOR *pattern*

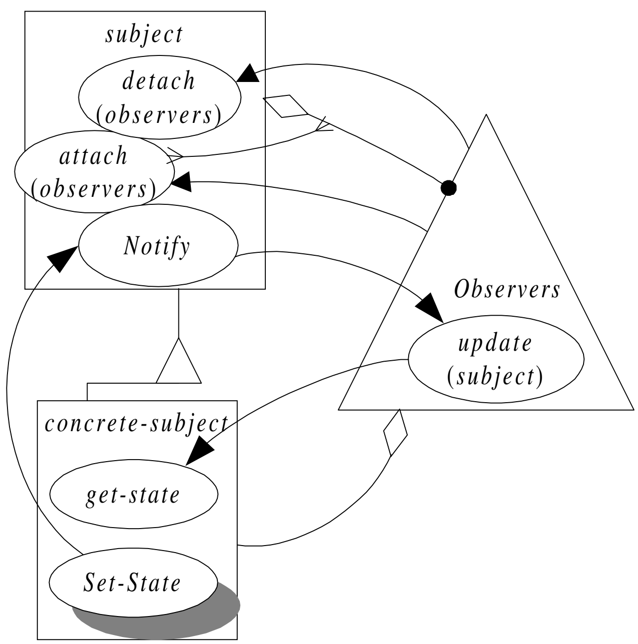

Figure 8: LePUS diagram of the OBSERVER *pattern*

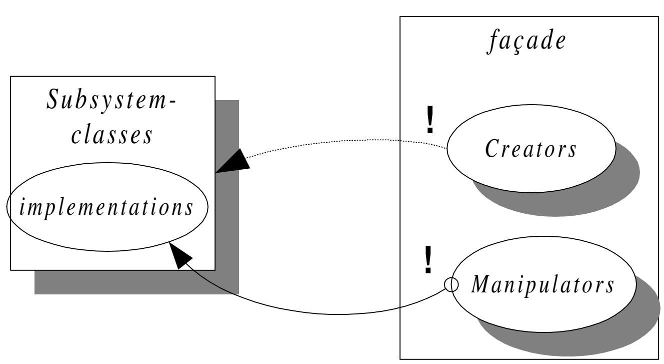

Figure 9: LePUS diagram of the FAÇADE *pattern*

1. *Defined-In*$^{\leftrightarrow}$(*accept*,*Elements*)
2. *Invoke*$^{\leftrightarrow}$(*accept*,*Visit*)
3. *Argument*$_1^{\leftrightarrow}$(*Visit*,*Elements*)
4. *Invoke*$^{\leftrightarrow}$(*Visit*,*Elements*) where a class *c* in *Invoke*(*f*,c) is interpreted as the set of functions defined in *c*

The *commute* designation on the four regular relations indicates that the three converging paths in the relations graph from *accept* to *Elements* commute; in other words, if $accept_i$ invokes $Visit_j$, which in turn invokes a function in $Element_k$, then $accept_i$ is defined in $Element_k$; additionally, the first argument of $Visit_j$ is of the same class $Element_k$. Also note that the sizes of the four sets involved must be equal.

The *OBSERVER* pattern (Figure 8) requires a larger number of predicates, some of which presented in section 2.3. The reason being the lack of regularity in the multitude elements of the pattern.

Assignment is a ternary relation that is uniquely connecting an icon to an (*Reference*) edge rather than to another vertex, thereby indicating the relation applies to the pair of vertices linked by the *Reference* edge, namely: *Assignment*(*detach*,*attach*,*Observers*)
.

Figure 9 depicts the *FAÇADE* pattern. Essentially, this pattern is about hiding the Subsystems behind the *façade* class. This requirement is manifested by the *exclusive* relations from the two *tribes* in *façade*; for instance, the predicate transcribed by Predicate 3 indicates that objects of *Subsystem-Classes* are created only within the *Creators* tribe defined in *façade*.

5. Applications

In this section we demonstrate two aspects of the utility of LePUS: (1) A rigorous reasoning mechanism of relationships between patterns, and (2) tool support in the implementation and recognition of design patterns.

5.1 Relationship Between Patterns

Relationship between design patterns were discussed in several publications. For example, Kim and Benner [96] discuss variations of the *OBSERVER* [GoF 95] pattern, and Rohnert [96] discuss variations of the *PROXY* [GoF 95]. The authors of the GoF catalog themselves discussed associations between the patterns ("Organizing the Catalog" [GoF 95 pp. 9-11]; "Design Patterns Relationship" [GoF 95 pp. 11-13]). Zimmer [95] divides the relations between the patterns of the [GoF 95] catalog in 3 types: "*X is similar to Y*", "*X uses Y*", and "*Variant of X uses Y*".

The formal transcription of design patterns to LePUS greatly facilitates the discovery and validation of relations between patterns. Our efforts in this direction have been towards the notion of *refinement*, a generalization of the relations "*X is a 'component of Y*", "*X is a special case of Y*", *tribe* and a *clan* (Definition VIII), and between *hierarchy* (Definition IX) and 1-dimantional class.

Refinement

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

This quote of Ralph Johnson is taken from [Vlissides 97a,b], a report on a discussion about the similarity between *MULTICAST*, a candidate design pattern, and the established *OBSERVER* [GoF 95] which was held between the authors of [GoF 95] catalog. The specification of the *MULTICAST* is made using the common means in the specification of design patterns, namely, verbally accounted examples and class diagrams. The four confer and disagree about the extent of similarity between the patterns. The discussion spreads over a number of pages, where each author stresses in words the distinctions or commonalties to stress his point.

LePUS specifications render such debates much simpler, providing formal means of specification and improved reasoning faculties on design patterns; in Johnson's words, we "went meta". As a result, *refinement* can be defined and established by means of predicate calculus:

Definition XI: A pattern π_1 ***refines*** pattern π_2 iff $\phi_1 \rightarrow \phi_2$ is true, where π_1 and π_2 transcribe to the formulae ϕ_1, ϕ_2 respectively.

Although the definition of *refinement* may be subjected to future "refinements", its usefulness with respect to the discussion above is apparent and illustrated in [Eden, Hirshfeld & Yehudai 98c], a follow-up to [Vlissides 97a,b]: We draw the diagrams of the patterns involved and prove that *MULTICAST* refines *TYPED MESSAGE* by showing that if *TYPED MESSAGE* transcribes to φ, than *MULTICAST* transcribes to a LePUS formula that is equivalent to $\phi \wedge \varphi$.

5.2 Tool Support

The uniformity of LePUS' *well-formed-formulae* and the systematic *generalizations* of relations (Definition III) permit a straight-forward representation of patterns in PROLOG, such that:

- *Ground variables* are represented as simple `variables`, and *higher dimension lists* are `list variables`
- *Ground relations* are represented as PROLOG `predicates`
- *Generalizations* (of all sorts) are implemented by PROLOG `rules`
- LePUS' *Formulae* too are represented as `rules`, which are satisfied if the list of `predicates` that apply is satisfied (recursively).

We have implemented a prototype for a tool that performs tasks of recognition and application of design patterns. In our implementation we used a statically typed, object oriented version of the language [*Visual Prolog* 97]. The tool operates in two modes:

- **Reverse engineering**: An Eiffel program is translated to a PROLOG `database` of `atoms` (*ground entities*) and `facts` (*ground relations*). This task is performed either automatically or with human assistance.

  A pattern can be proved (or refuted) with respect to a particular sequence of `atoms` in the database $a_1, \ldots a_n$ by running a PROLOG *query* with this sequence as its actual arguments.

  A pattern can be searched with respect to any sequence of atoms $a_1, \ldots a_k$. The execution of a PROLOG *query* with only some of the arguments supplied returns a list of all `atoms` that conform to the pattern's *rules* with the arguments supplied. If no input is supplied the query is interpreted as a search on all the possible sequences of the database.

- **Forward engineering**: We also can change each *predicate* into a *rule* which *asserts* the required relation, such as, for instance:

  ```
  inheritance(A,B) :- assert(inherit(A,B)).
  ```

  Executing a PROLOG query in these circumstances "induces" a pattern on the query's actual arguments by adding the required facts to the database.

6. Discussion

Many of the behavioral aspects of design patterns can be, surprisingly enough, represented faithfully through static *relations* as part of the logic framework of LePUS. No difficulties were found in the complete representation of *behavioral* design patterns [GoF 95 pp. 9-10], including patterns that were not presented here (*COMMAND*, *STATE*, *STRATEGY*, and *TEMPLATE METHOD*).

The specification of some design patterns is to fuzzy or teleologic (see an elaboration of the categories of formal specification in the [GoF 95] in [Eden, Hirshfeld & Yehudai 95d]). For instance, we doubt whether *ITERATOR* ("iteration operation"?) and *MEMENTO* ("capturing the internal state" [GoF 95 p. 283]).

LePUS is less expressive with respect to other patterns of the [GoF 95] catalog. The reasons are either that their specification is too language specific (such as the *SINGLETON* or *PROTOTYPE*), or involves more details on the dynamic properties than the scope of LePUS permits (*PROTOTYPE*).

Acknowledgments

Many thanks to Yonat Sharon and Eric Ernst for their insightful and very detailed comments. We thank Yaron Katzir for contributing his ideas.

Appendix A

Below appears an intuitive interpretation to the relations symbols mentioned throughout the article:

Assignment (f, c_1, c_2) designates that the reference from c_1 to c_2 is assigned a value within the flat structure of the body of function *f*.

Creation (f, c) designates a creation of an object of class c within the flat structure of f ("f may create c").

Invocation (f_1, f_2) designates a function call to f_2 within the flat structure of f_1 ("f_1 may call f_2").

Inheritance (c_1, c_2) designates that c_1 "inherits" from the c_1. Among the various techniques and uses of inheritance in different OOPLs (Taivalsaari [96]), we use "inheritance" as in the [GoF 95] i.e., delivering *substitutability* and allowing dynamic dispatch of overridden procedures ("virtual functions").

Forwarding (f_1, f_2) designates a function call to f_2 within the flat structure of f_1, using the formal arguments of f_1 as actual arguments in the same order, and that the signatures of the two functions are identical. We generally assume that *Forwarding* (f_1, f_2) implies *Invocation* (f_1, f_2).

Production (f, c) indicates that an object of class c is created within the flat structure of f and is returned as the result of f. We generally assume that *Production* (f, c) implies *Creation* (f, c).

Reference-To-One/Many designate both compositional and associative relations between classes. Thus, class attributes (also "data members" or "instance variables") are not distinguished from other associations.

Appendix B

A *total* relation applied to variables of different dimensions is defined as follows:

$$\beta^{\rightarrow}(V,w) \overset{def}{=} \forall v \in V : \beta(v,w)$$
$$\beta^{\rightarrow}(v,W) \overset{def}{=} \exists w \in W : \beta(v,w)$$

An *H-total* relation combining hierarchy variables with single classes is defined as follows:

$$\beta^{\rightarrow\mathrm{H}}(h,w) \overset{def}{=} \beta^{\rightarrow}(Nodes(h),w)$$
$$\beta^{\rightarrow\mathrm{H}}(v,h) \overset{def}{=} \beta^{\rightarrow}(v,root(h))$$

References

Alencar P. S. C., D. D. Cowan, C. J. P. Lucena (1996). “A Formal Approach to Architectural Design Patterns”. *Proceedings of the 3rd International Symposium of Formal Methods Europe*, pp. 576-594.

Barwise J. (1976, ed.) *Handbook of Mathematical Logic*, pp. 4-56.

Booch G. (1994). *Object Oriented Analysis and Design With Applications*. 2nd edition. Benjamin/Cummings.

Booch G., J. Rumbaugh, I. Jackobson (1997). “UML: A Unified Modeling Language”. `http://www.rational.com/uml`

Bosch J. (1996). *Language Support for Design Patterns.* TOOLS Europe '96.

Budinsky F. J., M. A. Finnie, J. M. Vlissides, P. S. Yu (1996). “Automatic code generation from design patterns”. *Object Technology* Vol. 35, No. 2.

Buschmann F., R. Meunier, H. Rohnert, P. Sommerlad, M. Stal (1996). *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons.

Coplien J. O., D. C. Schmidt (1995, eds.) *Pattern Languages of Program Design*. Addison Wesley.

Eden A. H., J. Gil, A. Yehudai (1997c). “Precise Specification and Automatic Application of Design Patterns”. The 12th IEEE International Automated Software Engineering Conference - ASE 1997. `http://www.math.tau.ac.il/~eden/bibliography.html#ase`

Eden A. H., Y. Hirshfeld, A. Yehudai (1998a). “LePUS - A Declarative Pattern Specification Language”. Technical report 326/98, department of computer science, Tel Aviv University. `http://www.math.tau.ac.il/~eden/bibliography.html#lepus`

Eden A. H., Y. Hirshfeld, A. Yehudai (1998c). “Multicast - Observer ≠ Typed Message”. Submitted: *C++ Report*, SIGS Publications. `http://www.math.tau.ac.il/~eden/bibliography.html#multicast_observer_typed_message`

Eden A. H. (1998d). “Giving ‘The Quality’ A Name”. *Journal of Object Oriented Programming*, guest column, June 98. SIGS Publications. `http://www.math.tau.ac.il/~eden/bibliography.html#giving_the_quality_a_name`

Florijn G., M. Meijers, P. van Winsen (1997). *Tool Support in Design Patterns.* In: Askit M., S. Matsuoka (1997, eds.) *Proceedings of the 11th European conference on Object Oriented Programming - ECOOP’97.* Lecture Notes in Computer Science no. 1241. Berlin: Springer-Verlag.

GoF (1995): Gamma E., R. Helm, R. Johnson, J. Vlissides. *Design Patterns*: *Elements of Reusable Object Oriented Software*. Addison-Wesley.

Hedin G. (1997). *Language Support for Design Patterns using Attribute Extensions*. In Bosch J., S. Mitchell (1997, eds.) *Object-Oriented Technology - ECOOP'97 Workshop Reader*. Lecture Notes in Computer Science no. 1357. Berlin: Springer-Verlag.

Helm R., I. M. Holland, D. Gangopadhyay (1990). *Contracts: Specifying Compositions in Object Oriented Systems.* Proceedings of OOPSLA 1990, SIGPLAN Notices, vol.25 no.10.

Kim J. J. K. M. Benner (1996). *Implementation Patterns for the Observer Pattern*, in [Vlissides, Coplien & Kerth 96].

Klarlund N., J. Koistinen, M. I. Schwartzenbach (1996). "Formal Design Constraints". *Proceedings of ACM SIGPLAN Conference On Object-Oriented Programming Systems, Languages and Applications* 1996.

Lauder A., S. Kent (1998). "Precise Visual Specification of Design Patterns". *ECOOP'98*.

Martin R., D. Riehle, F. Buschmann (1997, eds.) *Pattern Languages of Program Design 3*. Addison-Wesley

Minski N. H. (1994). "Law Governed Regularities in Software Systems". Technical report LCSR-TR-220, Rutgers University, LCSR, Jan. 94.

Pal P. P. (1995). "Law-Governed Support for Realizing Design Patterns". *TOOLS USA* 17.

Pierce, C. B. (1993). *Basic Category Theory for Computer Scientists*. MIT Press.

Pree W. (1994). *Metaprogramming Patterns – A Means for Capturing the Essentials of Reusable Object-Oriented Design.* Proc. of the 8th European conference of Object Oriented Programming. Springer-Verlag.

Quintessoft Engineering, Inc. (1997). *C++ Code Navigator* 1.1. `http://www.quintessoft.com`

Rohnert H. (1996). *The Proxy Design Pattern Revisited*, in: [Vlissides, Coplien & Kerth 96]

Rumbaugh J., M. Blaha, W. Premerlani, F. Eddy, W. Lorensen (1991). *Object Oriented Modeling and Design*. Prentice Hall.

Taivalsaari A. (1996). "On the Notion of Inheritance". *ACM Computing Surveys* 28 (3) pp. 438-479.

Visual Prolog 5.0 (1997). Prolog Development Center A/S. `http://www.pdc.dk`

Vlissides J. (1997a). "Multicast". *C++ Report*, Sep. 97. SIGS Publications.

Vlissides J. (1997b). "Multicast - Observer = Typed Message". *C++ Report*, Nov.-Dec. 97. SIGS Publications.

Vlissides J. M., J. O. Coplien, N. L. Kerth (1996, eds.) *Pattern Languages in Program Design 2*. Addison-Wesley.

Zimmer W. (1995). *Relationships Between Design Patterns.* In: [Coplien & Schmidt 95]