

FORMAL SPECIFICATION OF OBJECT-ORIENTED DESIGNⁱ

AMNON H. EDEN

Department of Computer Science, Concordia University

Montreal, Quebec H3G 1M8, Canada

Abstract. Software architectures and designs "notations" are typically based on bubble-and-arc diagrams. Embellishing these diagrams may increase their information content but does not make a language. A formal language has syntax, semantics, and inference rules, so that reasoning and automatic manipulation are made possible.

We describe LePUS, a formal language for describing and reasoning about object oriented software architectures, designs, and patterns. A LePUS formula imposes constraints on the software at an appropriate level of abstraction but does not unnecessarily restrict the implementation. LePUS is not an ad hoc collection of loosely related concepts but instead originates from an insight on a small number of necessary and sufficient basic "building blocks" that are ubiquitous in object oriented design. A LePUS specification can be expressed as either a formula or a semantically equivalent diagram. We provide examples of LePUS descriptions ranging from simple design patterns, such as *FACTORY METHOD*, to popular current architectures, such as Enterprise JavaBeans™.

Keywords: Software architecture, object oriented programming, design patterns, formal methods

1. Overview

In a pioneering discussion of software architecture [1], Perry and Wolf suggested some of the desirable properties that an architectural specification language should possess:

- (req. i) Generality. The language should allow the expression of constraints at the necessary level of detail. "What is not constrained by the architect may take any form desired by the implementer" [1] (also *the principle of least constraint*).
- (req. ii) Abstraction. Instead of "an assembly-level" architectural language that allows the specification of all conceivable configurations, yet is too verbose, we are interested in a language that restricts our discussion to topologies of interest. Such a language should incorporate constructions that allow for specifications that are appropriately concise.
- (req. iii) Elementary building blocks. The language is expected to provide a relatively small set of design elements from which complex expressions can be constructed.

ⁱ Presented in the *International Conference on Multidisciplinary Design in Engineering CSME-MDE 2001*, November 21-22, 2001, Montreal, Canada.

Further desirable properties that are expected from a specification language include the following:

- (req. iv) Formal. The language should have well defined, unambiguous semantics, and be founded on a formalism that is of an adequate reasoning power.
- (req. v) Concise [4]. The expressions that specify commonly occurring motifs should be “shorter” than the specifications of rare constructions.
- (req. vi) Compact. “Less is more” [2], and among specification languages of similar expressiveness, less “words” make a more powerful languageⁱ.

In Section 2, we introduce an example that illustrates each of these properties and also the weaknesses of current techniques.

Our first contribution, presented in detail in Section 3, is the observation of the fundamental building blocks of design patterns and of object oriented (OO) architecture, including classes and functions of higher orders; function families (*clans* and *tribe*); and isomorphisms and other relations of special interest.

In Section 4 we describe our second contribution: LePUS (*Language for Patterns Uniform Specification*) [3], a specification language that has been proved to be compact, concise, and expressive [4], and which meets the conditions specified in (req. i) to (req. vi). LePUS is specifically tailored to OO architectures and has been used so far mainly to describe the constructions that appear in the design patterns literature [5, 6, 7, 8].

Section 5 demonstrates that specifications in LePUS can be given in either visual or symbolic form. Both forms have the same underlying semantics and each form can be systematically converted into the other.

In Section 6 we demonstrate various applications of LePUS, including –

- ◆ *validation* of architectural specifications;
- ◆ *reasoning* and proving relations between design patterns;
- ◆ *implementation* of patterns by support tools.
- ◆ *documenting* concrete architectures of class libraries and of application frameworks.

In Section 7, we conclude the paper with a discussion of the related work.

2. Motivation

To motivate the discussion, consider the *Structure* diagram of the *FACTORY METHOD* pattern [9], shown in Figure 1. Seeking to express the *general* properties of the pattern, the authors had to resort to informal hints in order to overcome the limitations of *class dia-*

ⁱ And more is less, as in the case of UML.

grams [19]. Many patterns are presented in precisely this way: as a diagram supported by informal comments. Table 1 lists some of the information the authors sought to expressⁱ.

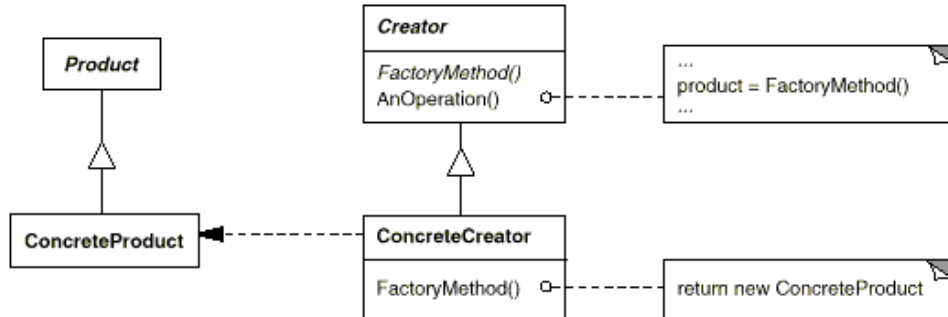


Figure 1. The “Structure” (OMT class diagram) of the *FACTORY METHOD* pattern

Table 1. Properties of the *FACTORY METHOD* pattern

Diagram element	Implied property
1. ConcreteCreator	Any number of concrete classes inherit (possibly indirectly) from <i>Creator</i>
2. ConcreteProduct	Any number of concrete classes inherit (possibly indirectly) from <i>Product</i>
3. FactoryMethod	An abstract method is defined in the <i>Creator</i> base class and in each class that derives from it
4. Note on FactoryMethod	Each concrete version of <i>FactoryMethod</i> creates an instance of a respective <i>ConcreteProduct</i>

LePUS incorporates explicit abstractions for each of the properties listed in Figure 1. More specifically, the elements of the *ABSTRACT FACTORY* pattern can be expressed as follows:

- (i) Creator, ConcreteCreator. LePUS defines a *hierarchy* as a set of classes containing one that is abstract such that all other classes inherit (possibly indirectly) from it. Hence, the representation of the *Creator*, together with the *ConcreteCreator* classes, takes the form of declaring a set of classes that is a *hierarchy*.

$$\begin{aligned} \text{Creators} &: \mathbf{P}(\mathbb{C}) \\ \text{Hierarchy}(\text{Creators}) \end{aligned} \quad (1)$$

- (ii) Product, ConcreteProduct. Similarly, the joint representation of the “Product” classes is in the form of a hierarchy:

$$\begin{aligned} \text{Products} &: \mathbf{P}(\mathbb{C}) \\ \text{Hierarchy}(\text{Products}) \end{aligned} \quad (2)$$

LePUS *hierarchies* have a distinguished representation in the visual formalism,

ⁱ As concluded from the complete specification of the pattern. Note, though, that adherence to the original descriptions was compromised in favor of simplicity.

depicted in Diagram 1.

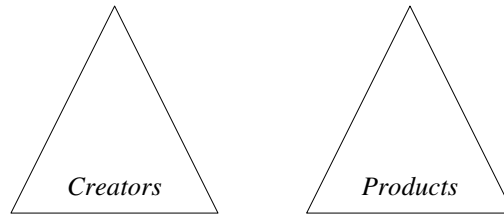


Diagram 1. Two class hierarchies

- (iii) FactoryMethod. It is implied from that all **FactoryMethod** functions share the same signature and each is defined in a different class of the *Creators* hierarchy. A set of functions with this property is called a *clan* in LePUS. In this case, we say that *FactoryMethods* is a *clan* in *Creators* (3). Clans in fact are groups of functions that are connected by dynamic binding.

$$\begin{aligned} \text{FactoryMethods} &: \mathbf{P}(\mathbb{F}) \\ \text{Clan}(\text{FactoryMethods}, \text{Creators}) \end{aligned} \quad (3)$$

Clans also have a special visual representation, depicted in Diagram 2.

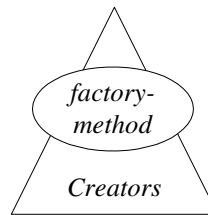
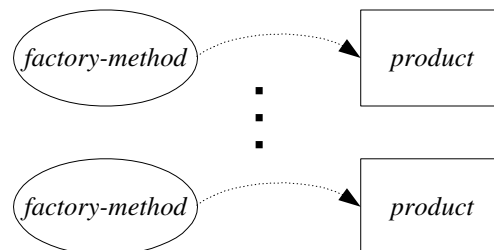


Diagram 2. A clan named *factory-method* in *Creators*

The correlation between the textual and the visual representation will be clarified in the next section.

- (iv) Note on FactoryMethod. implies that each factory method creates instances of exactly one class of the *Products* hierarchy. Figure 2 illustrates this property.



We say that the relation between the sets *FactoryMethods* and *Products* is an *isomorphism*, namely, *Creation* is a bijective function between the two sets. Isomorphisms are indicated in LePUS by a double arrow \leftrightarrow as follows:

$$Create^{\leftrightarrow}(FactoryMethods, Products) \quad (4)$$

The sets and isomorphism in (4) can be represented in the visual formalism as follows:



Diagram 3. An isomorphic *Creation* between two sets

Diagram 4 summarizes the formal representation for the elements of \cdot . (Note that the representation of *FactoryMethods* is in the form of a *clan* in *Creators*, as in Diagram 2, and not merely as a set of functions, as in Diagram 3.)

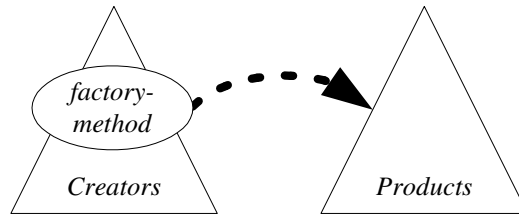


Diagram 4. Towards the *FACTORY METHOD*

3. The Computational Model

Expressions in LePUS specify patterns in terms of constraints on the properties of programs. We say that a program *implements* (or incorporates an instance of) a pattern if it conforms to the respective set of constraints. LePUS can therefore be used in a number of different ways: to describe patterns, to recognize programs that incorporate given patterns, to decide whether two patterns are related, to derive patterns from other patterns, and so on. Although LePUS was introduced in the context of a discussion of patterns, we demonstrate how it is a generally useful language for the specification of software design and architecture, with pattern description being only one of its possible applications. Section 5 demonstrates some of these applications.

Since LePUS is intended primarily for the description of OO systems, its elementary concepts are *methods* and *classes*, called *ground entities*. Primarily, LePUS expressions characterize properties of sets of functions and classes and relations thereof. A *uniform set* (

Definition II) is simply a set whose members are all of the same kind. For example, one uniform set might consist of functions and another uniform set might consist of classes, but a uniform set would not contain both functions and classes.

The expressiveness of LePUS is enhanced by its use of higher order (also *dimension*) sets and relations. High order entities, such as sets of sets and relations between sets of sets, provide the principle abstraction mechanism of LePUS. We show below, for example, that the *ABSTRACT FACTORY* pattern can be obtained from the *FACTORY METHOD* pattern by a simple LePUS abstraction operation.

In this section, we deliver a concise exposition of the language. Formal definitions of terms appear in the Appendix.

3.1 *The Universe of Discourse*

We assume a universe of *functions* and *classes*ⁱ, called *ground entities* or *individuals*. We can draw elements from this universe to form uniform sets according to various rules. For example, we can construct the set of functions that are defined in a given class.

A set constructed according to a rule can also be considered as a relation. In LePUS, we consider such sets as relations. A *unary relation* is a set of individuals. For example, *Abstract* is the unary relation of abstract functions: $Abstract(f)$ is true if f is an abstract function and false otherwise. The interpretation of what it means for a function to be “abstract” depends on the implementation language.

A *binary relation* is a set of ordered pairs of individuals. For example, the relation *Inherits* consists of all the pairs $(c1, c2)$ such that class $c1$ inherits from $c2$. Other frequently-used binary relations in LePUS are the following:

- ◆ The relation *SameSignature* contains the pairs (f, g) such that the functions f and g have the same signature, where the term *signature of a function* is taken from programming languages (typically comprising of the function name and parameter types).
- ◆ The relation *Calls* contains the pairs (f, g) such that function f calls, or invokes, function g . Since the relation represents a static property, it abstracts the details of control information (loops, branches, etc.).
- ◆ *DefinedIn* is a binary relation consisting of the pairs (f, c) such that function f is defined in class c .
- ◆ The relation *Create* contains the pairs (f, c) such that a side-effect of some expression within the body of the function f is that an instance of class c is created. Note that f is not necessarily a constructor but that it must directly or indirectly invoke a constructor.

ⁱ The terms “class” and “function” are taken from OO programming and should not be interpreted in their mathematical sense.

LePUS is not constrained to a fixed set of ground relations. Different contexts warrant different sets of relations. The interpretation of other relations used in this article is intuitively clear from their names. A more detailed description of the relations used in the specification of the GoF patterns [9] can be found in [3].

Furthermore, while we have only mentioned unary and binary relations, there is no objection in principle to introducing ternary or higher arity relations if they turn out to be useful.

An important question is how constructs of a programming language map into the relations in LePUS specifications, and vice versa. The answer to this question has direct implications on all the applications of LePUS, such as reasoning and tool support, which are discussed in section 5. What complicates the answer to this question is the fact that different programming languages may have significantly different mappings. For instance, consider the *Inherit* relation: In C++ we may choose to “map” it to *public inheritance*, while in Java it may be mapped to *implementation*. The situation may be even more complicated with some relations that do not necessarily map 1:1 to a syntactic construct. Mapping the *Create* relation to C++ is an example to such a case, where expressions of varying types result in the generation of temporary objects, not always in an explicit form.

We have consider the intuitive interpretation of some of the relations we use in this article. A more detailed answer to the mapping question is beyond the scope of this paper. At this point it is enough to observe, as demonstrated in [3], that each one of the small set of relations used in the representation of the GoF patterns [9] maps directly to a well defined set of syntactic constructs in statically typed languages such as C++, Java™, and Eiffel.

3.2 *Building Blocks*

Certain sets are of more interest to us than others, and LePUS incorporates a concise representation for them. The first two domains are \mathbb{C} and \mathbb{F} , designating the collections of *class* entities and *function* entities respectively. Thus, instead of using a relation to describe an entity, we can specify its *domain*. For example, instead of writing *Class(c)*, we may write $c \in \mathbb{C}$.

Constructions and sets with properties of particular interest are expressed as *domains* or as *predicates*. In this section we describe the constructions, or “building blocks”, which we find of special interest.

- (i) A *class hierarchy* (Definition IX) or simply “*hierarchy*” h is a set of classes with the following property: there is a unique class `root` in h , called “the root of h ”, which is abstract, and such that all other classes in h inherit from it. The sets *Creators* and *Products* of the *FACTORY METHOD* (Diagram 1) are examples of *hierarchies*.

If this condition holds for a set of classes h , we indicate it with the *Hierarchy* predicate as follows:

$$\textit{Hierarchy}(h) \tag{5}$$

We denote the domain of all hierarchies with the symbol \mathbb{H} . Thus, (5) is equivalent to proposition: $h \in \mathbb{H}$.

- (ii) There are many situations in which we have to indicate not just that a relation applies to certain entities but that it has specific properties. One such property is *totality* (Definition V). A relation \mathcal{R} is *total* on sets U and V if and only if, for every u in U , there is some v in V such that $(u, v) \in \mathcal{R}$.

We explain the *total* predicate in terms of an example. Consider the relation *Reference*, where $\textit{Reference}(c, d)$ is true if class c holds a “data member” or “attribute” of type d . This relation applies to many different classes in our universe. However, we are usually interested not in all classes but only in particular sets of classes. For example, we might be interested in expressing the correlation that *Reference* holds between the two sets of classes `Iterators` and `Elements`. In particular, that for every class i in `Iterators` there is some class e in `Elements` such that $\textit{Reference}(i, e)$. In such case we write

$$\textit{Total}(\textit{Reference}, \textit{Iterators}, \textit{Elements}) \tag{6}$$

LePUS actually uses an abbreviated form of this notation, in our example –

$$\textit{Reference}^{\rightarrow}(\textit{Iterators}, \textit{Elements}) \tag{7}$$

Generally, if a relation \mathcal{R} is total in sets U and V , we write $\mathcal{R}^{\rightarrow}(U, V)$.

- (iii) *Isomorphic* is a stronger property of a relation than *totality* but it is defined in a similar way. We illustrate the *isomorphic* property through the following example: Suppose that the class `EnchantedFactory` has a method `MakeRoom` that constructs an instance of `EnchantedRoom`; and that the class `BombedFactory` has a method `MakeRoom` that constructs an instance of `BombedRoom` (source code given in Table 2). We can name the sets

$$\begin{aligned} \textit{RoomFactoryMethods} &\equiv \{ \textit{EnchantedFactory}::\textit{MakeRoom}, & (8) \\ & \quad \textit{BombedFactory}::\textit{MakeRoom} \} \\ \textit{RoomProducts} &\equiv \{ \textit{EnchantedRoom}, \\ & \quad \textit{BombedRoom} \} \end{aligned}$$

If we restrict *Create* to the sets `RoomFactoryMethods` and `RoomProducts`, it is a one-to-one correspondence, or a bijective function with domain `FactoryMethods` and range `Products`. In LePUS, we refer to such correspondences as *isomorphisms* (Definition VI), which can be written as $\textit{Isomorphic}(\textit{Create}, \textit{RoomFactoryMethods}, \textit{RoomProducts})$. The abbreviated notation for the *Isomorphic* property uses a double arrow \leftrightarrow as follows:

$$Create^{\leftrightarrow}(\text{RoomFactoryMethods}, \text{RoomProducts}) \quad (9)$$

as illustrated in Figure 2.

Similarly, the *isomorphic* property can express the correspondence between classes and their functions. Thus,

$$DefinedIn^{\leftrightarrow}(\text{RoomFactoryMethods}, \text{RoomFactories})$$

means that:

- ◆ for every f in `RoomFactoryMethods` there is a unique c in `RoomFactories` such that $DefinedIn(f, c)$, and
- ◆ for every c in `RoomFactories` there is a unique f in `RoomFactoryMethods` such that $DefinedIn(f, c)$.

- (iv) We extend the concept of *isomorphism* to sets of higher order. Consider, for example, the C++ example given for the *ABSTRACT FACTORY*¹ pattern [9, pp. 92-94], quoted in Table 2.

Table 2. Source code for the *ABSTRACT FACTORY* example

```

// Hierarchy of product #1: Room:
class Room {};
class RoomWithBomb : public Room {};
class EnchantedRoom : public Room {};

// Hierarchy of product #2: Door:
class Door {};
class BombedDoor: public Door {};
class DoorNeedingSpell : public Door {};

// Factory interface:
class MazeFactory { /* interface definition omitted */ };

// Factory class for Enchanted objects
class EnchantedFactory : public MazeFactory {
    virtual Room * MakeRoom()
        { return new EnchantedRoom(); }
    virtual Door * MakeDoor()
        { return new DoorNeedingSpell(); }
};

// Factory class for Bombed objects
class BombedFactory : public MazeFactory {
    virtual Room * MakeRoom()
        { return new RoomWithBomb(); }
    virtual Door * MakeDoor()
        { return new BombedDoor(); }
};

```

This example is in fact an extension of (9). It includes two explicit isomor-

phisms:

$$\begin{aligned} Create^{\leftrightarrow}(\text{RoomFactoryMethods}, \text{RoomProducts}) \\ Create^{\leftrightarrow}(\text{DoorFactoryMethods}, \text{DoorProducts}) \end{aligned} \quad (10)$$

Naturally, the *ABSTRACT FACTORY* is not limited to two product hierarchies; instead, the intent of this example is to say: However many “product” hierarchies are defined, each one is created by exactly one set of factory methodsⁱⁱ (and vice versa).

Now we can define sets of higher order that group together sets in (8) by their roles as follows:

$$\begin{aligned} \text{FactoryMethods}^2 &\equiv \{ \text{RoomFactoryMethods}, \\ &\quad \text{DoorFactoryMethods} \} \\ \text{Products}^2 &\equiv \{ \text{RoomProducts}, \\ &\quad \text{DoorProducts} \} \end{aligned}$$

where the subscript designates that the sets contain sets rather than ground elements.

Given the arbitrary levels of abstractions in LePUS, we now may express (10) in one *isomorphic* expression on sets of sets:

$$Create^{\leftrightarrow}(\text{FactoryMethods}^2, \text{Products}^2) \quad (11)$$

Figure 3 illustrates the isomorphic correlation between the higher order sets that is expressed in (11). This illustration demonstrates that complex relations can be described by highly compact LePUS expressions.

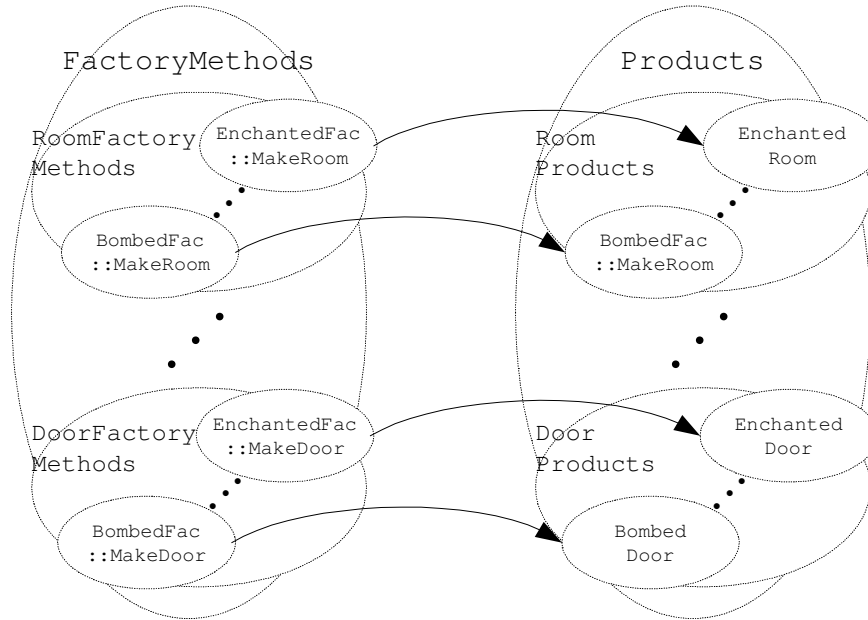


Figure 3. Illustration to (11)

- (v) Suppose that `FactoryMethods` is a set of methods with the same signature; `factories` is a set of classes; and that each function in `FactoryMethods` is defined in a different class in `factories`. In other words, there is a one-to-one correspondence between factories and `FactoryMethods`. Using the *Isomorphic* concept, we can write $DefinedIn^{\leftrightarrow}(factories, FactoryMethods)$. Then we say that `FactoryMethods` is a *clan* (Definition X) in `factories`, and we write:

$$Clan(FactoryMethods, factories) \quad (12)$$

Clans are often associated with class hierarchies: an abstract function is defined in the root class and implemented in each of the concrete classes. In the *FACTORY METHOD* example in section 20, `FactoryMethods` is indeed a clan in the `factories` hierarchy. Clans are fundamental to OO programming as they are the static mechanism which allows dynamic dispatch.

In the trivial case of a clan in a single class, Definition X simply indicates that the specified function is defined in the respective class. For example, given ground function `attach` and ground class `subject`, the clause

$$Clan(attach, subject) \quad (13)$$

simply indicates that `attach` is defined in `subject`.

- (vi) In many cases, there will be a number of such methods, that is, a set of related

clans. Consider, for example, a set of classes `visitors`, a clan in it called `VisitElm-1`, another clan named `VisitElm-2`, and so forth. Let us designate the set of clans as `VisitElements`. More specifically, we say that $Clan(VisitElm-1,visitors), \dots, Clan(VisitElm-N,visitors)$. In this case we say that `VisitElements` forms a *tribe* in `visitors` (Definition XI), written as:

$$Tribe(VisitElements,visitors)$$

- (vii) Consider the following excerpt from the specification of the *FLYWEIGHT* pattern [9]:

Clients should not instantiate ConcreteFlyweights directly. Clients must obtain ConcreteFlyweight objects exclusively from the FlyweightFactory object to ensure they are shared properly.

We seek to represent the constraint on the creation of *ConcreteFlyweight* such that only `FlyweightFactory` may instantiate `ConcreteFlyweight`. Alternatively we can say that if the relation

$$Create(x,ConcreteFlyweight) \tag{14}$$

holds for some x , than x can only be `FlyweightFactory`. In LePUS we represent this information by the exclusive property (Definition VII), indicated by means of an exclamation mark:

$$Create(FlyweightFactory!,ConcreteFlyweight) \tag{15}$$

- (viii) We may need to express the property that two relations “overlap” in a given domain and range. In category theory, we would draw a diagram with arrows from the domain to the range and we would say the diagram “commutes” (Definition VIII). For instance, in conjunction with (9), the statements

$$\begin{aligned} &Return\ Type^{\leftrightarrow}(FactoryMethods,Products) \\ &Commutate_{Create,Return\ Type}(FactoryMethods,Products) \end{aligned}$$

imply that if a method f instantiates $c1$, and has the return type $c2$, then $c1 = c2$.

The relations and generalizations have been defined in such a way that we can describe most of the key properties of design patterns.

4. The Language

By the *principle of least constraint* [1], architectural specification should describe constraints at the desired level rather than supply a specific solution. Accordingly, LePUS expressions manifest constraints on variables representing sets of elements in programs

and their properties. To test whether these properties are satisfied by a certain program, for example, we assign constructions in the program to the free variables and check if they satisfy the formula. If so, we say that the particular sequence of program constructions *validates* the formula.

This section presents the syntax of LePUS. A *well formed formula* (Definition III) is a conjunction on sequence of *clauses*. Each clause consists of a combination of the following elements:

- ◆ Variables
- ◆ Predicates
- ◆ Operators

Below we discuss each element separately.

4.1 Variables

Variables range over specific domains. Other than \mathbb{F} and \mathbb{C} (the domains of ground functions and ground classes, respectively), our domains of interest consist of *uniform sets* (Definition II). A *uniform set* is simply a set whose members are all of the same kind. For example, one uniform set might consist of functions and another uniform set might consist of classes, but a uniform set would not contain a mixture of functions and classes. Similarly, a uniform set does not contain elements of different dimensions.

For example, a variable of type *function* and dimension 2 ranges over sets of sets of functions. Below appears a declaration of such variable (where $\mathbf{P}(\mathbb{F})$ stands for the power set of \mathbb{F}):

$$Visit : \mathbf{P}^2(\mathbb{F}) \tag{16}$$

By convention, the lower case names f, f_1, f_2 designate function variables of dimension 0; upper case F, F_1, F_2 stand for function variables of dimension 1; and F^d, F_1^d, F_2^d designate function variables of dimension d .

Finally, we refer to a ground class (function) as a *class (function) of dimension 0*. A uniform set consisting of classes (functions) of dimension $d-1$ is referred to as a *class (function) of dimension d* . This definition allows for additional flexibility in the terms *function* and *class*, which is desirable for our discussion. For example, we say that *Products* (2) stands for a class of dimension 1, and that *FactoryMethods* (3) stands for a function of dimension 2.

4.2 Predicates

Instead of connectives and quantifiers, LePUS specifications express architectural properties using a small set of *predicates* (Definition IV). In this section, we present informal descriptions of the predicates of LePUS; the Appendix contains formal definitions.

The predicate *Hierarchy*(\mathbb{C}), as used in (5), is true for a set of classes \mathbb{C} if it forms an inheritance hierarchy (Definition IX). The property expressed by the predicate is that

there is a distinguished member of \mathbb{C} called the root class and all other elements of \mathbb{C} inherit from the root.

Predicates may be implied by a declaration. For example, the domain \mathbb{H} represents the set of all x such that x is a hierarchy. Hence, a declaration in the form

$$\textit{Creators} : \mathbb{H} \tag{17}$$

is in fact equivalent to (1).

The predicate $\textit{Clan}(\mathbb{F}, \mathbb{C})$ is true of a set of functions \mathbb{F} and a set of classes \mathbb{C} if the members of \mathbb{F} share a common signature and each member of \mathbb{F} is defined in exactly one member of \mathbb{C} . $\textit{Clan}(\mathbb{F}, \mathbb{C})$ is read as "the functions in \mathbb{F} form a clan in \mathbb{C} ". The set \mathbb{C} is usually a hierarchy.

The predicate $\textit{Tribe}(\mathbb{F}, \mathbb{C})$ generalizes \textit{Clan} to sets of sets of functions such that each set forms a clan in \mathbb{C} . Like \textit{Clan} , this predicate expresses a common programming technique.

Other predicates of LePUS define properties of relations. The ground relations in LePUS are defined over the domains \mathbb{F} and \mathbb{C} . However, they are almost always employed in subsets of these domains. A relation that is restricted to a particular context often manifests interesting properties which are expressed by predicates.

One of these predicates is *Total* (Definition V), where $\textit{Total}(\mathcal{R}, \text{Dom}, \text{Ran})$ expresses the fact that every element of Dom is related to at least one element of Ran by the relation \mathcal{R} . LePUS provides the abbreviated notation

$$\mathcal{R}^{\rightarrow}(\text{Dom}, \text{Ran})$$

The predicate *Isomorphic* (Definition VI) is stronger than the predicate *Total*. If $\textit{Isomorphic}(\mathcal{R}, \text{Dom}, \text{Ran})$ holds then there is a one-to-one correspondence between Dom and Ran in \mathcal{R} . The corresponding LePUS notation is

$$\mathcal{R}^{\leftrightarrow}(\text{Dom}, \text{Ran})$$

Both *Total* and *Isomorphic* predicates extend to sets of dimensions greater than one. However, it is always the underlying relation on individuals of \mathbb{F} and \mathbb{C} that determines the meaning of a property.

Two relations \mathcal{R} and \mathcal{S} may be equivalent to one another with respect to a particular domain dom and range ran . In this case, we say that they *commute* (in the sense of category theory). The formal notation for commuting relations is

$$\textit{Commute}_{\mathcal{R}, \mathcal{S}}(\text{dom}, \text{ran})$$

4.3 Operators

Consider the specification of the "skeleton" class in the Enterprise JavaBeans™ framework (EJB is discussed in detail in Section 6.1):

“When the skeleton receives a network message from the stub, it identifies the method invoked and the arguments, and then invokes the corresponding method on the actual instance.”

This description can be expressed in LePUS as follows:

$$\begin{aligned} \textit{Skeleton}, \textit{Bean} &: \mathbb{C} \\ \textit{skMethods}, \textit{bMethods} &: \mathbf{P}(\mathbb{F}) \end{aligned} \tag{18}$$

$$\begin{aligned} &\textit{Tribe}(\textit{skMethods}, \textit{Skeleton}) \\ &\textit{Tribe}(\textit{bMethods}, \textit{Bean}) \\ &\textit{Forward}^{\leftrightarrow}(\textit{skMethods}, \textit{bMethods}) \\ &\textit{SameSignature}^{\leftrightarrow}(\textit{skMethods}, \textit{bMethods}) \\ &\textit{Commute}_{\textit{Forward}, \textit{SameSignature}}(\textit{skMethods}, \textit{bMethods}) \end{aligned}$$

(The last two clauses indicate that each method in *skMethods* forwards the call to one in *bMethods* by the same signature.)

One may consider (18) verbose. The reason is that, fundamentally, there is only one set of “function signatures” which has a separate implementation on each one of the method sets *skMethods* and *bMethods*. The very distinction between *skMethods* and *bMethods* is artificial.

By referring to functions through their signatures rather than directly, LePUS offers yet another level of abstraction. We do so by defining the selection operator \otimes between a signature *s* and a class *c* as follows: $s \otimes c$ returns the unique (by Axiom 1) function that is defined in *c* with signature *s*. Similarly, for a set of signatures *S* we can define $S \otimes c$ as the set of all functions defined in *c* whose signature is in *S*. It is easy to prove that $s \otimes c$ gives a clan in *c*, and that $S \otimes c$ gives a tribe in *c*. Definition XIII extends this definition to classes of any dimension.

Let *Signatures* vary over sets of signatures. Using the definition of the selection operator, we may rephrase (18) as follows:

$$\textit{Forward}^{\leftrightarrow}(\textit{Signatures} \otimes \textit{Skeleton}, \textit{Signatures} \otimes \textit{Bean}) \tag{19}$$

Since our work on operators is still in progress we shall not discuss them any further in this article.

5. The Visual Formalism

Some kinds of information can be conveyed more efficiently by diagrams than by text. Diagrams are especially appropriate for expressing relationships between entities, which is also the primary goal of LePUS.

We have already used our motivational example (Section 2) as the basis for a visual formalism in LePUS. In this section we define the visual formalism in detail and describe how it correlates to the symbolic formalism. Since people vary in their preferences, the

choice of text or diagrams is very much a matter of taste. By providing both textual and diagrammatic vocabularies, we expect to increase the number of people who find LePUS comfortable to work with.

Variables

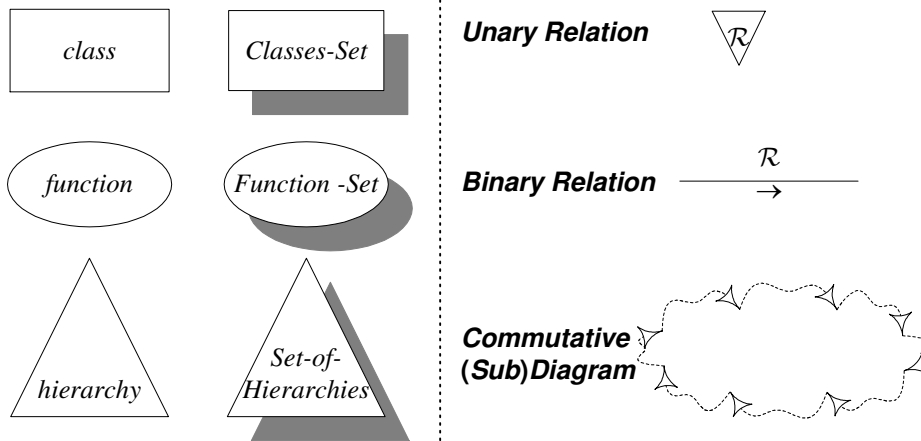


Figure 4. The basic set of graphic symbols in LePUS

The advantages of diagrams are well-known and widely advertised; the drawbacks are not often discussed. Diagrams are useful in engineering activities to the extent that they are precise. A vague diagram may do more harm than good by suggesting precision that it does not in fact possess [10]. In the seventies, dataflow diagrams were precisely defined, effective, and widely-used; subsequently, they were employed without a full understanding of the underlying methodology, degenerated into vagueness, and were eventually abandoned.

LePUS diagrams are precise. There is a direct correspondence between the visual and textual forms of an architecture, design, or pattern. Diagrams provides a quick and accurate overview of the system, while text supports detailed reasoning. In many cases, a simple visual transformation corresponds to a simple textual transformation. For example, turning a two-dimensional shape into a three-dimensional shape by the simple expedient of providing a shadow effect corresponds to increasing the "dimension" of a term, as in going from a clan to a tribe.

A *well formed diagram* in LePUS is equivalent to a *well formed formula* (Definition III). Figure 4 lists the figures used the visual notation and the linguistic construct delineated by each, where:

- ◆ *icons* represent variables
- ◆ *unary* relation marks represent unary relations
- ◆ *edges* represent binary relations
- ◆ *commute* designation(s) circumscribe the relations and domains of commuting

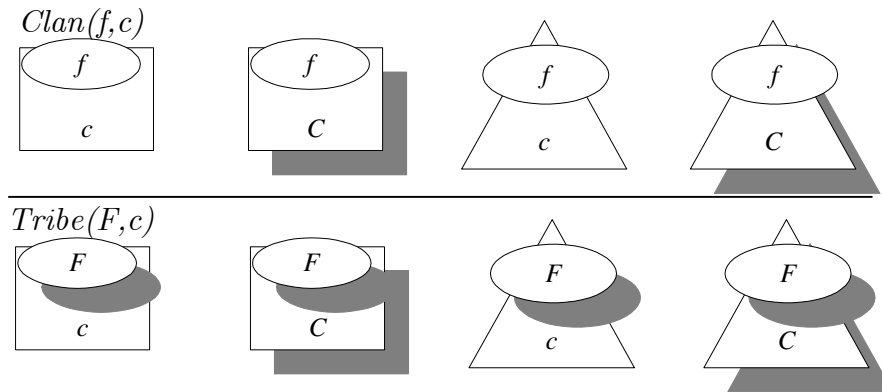


Figure 5. Superimpositions represent *clans* and *tribes*

Clans and tribes are represented by combinations of function and class icons, as defined in

Figure 5. For example, clause (12), which states that `FactoryMethods` is a clan in the `FactoryMethods` hierarchy, is equivalent to Diagram 2.

A *diagram* is a graph whose vertices consist of variable *icons*, possibly adorned with a *unary relation* marks, and whose *edges* are each labeled by a binary relation symbol. An edge labeled \mathcal{R} , connecting vertices v_1, v_2 represents the predicate $\mathcal{R}(v_1, v_2)$. For instance, *Inheritance* edges must connect class icons, or else the diagram is not well formed.

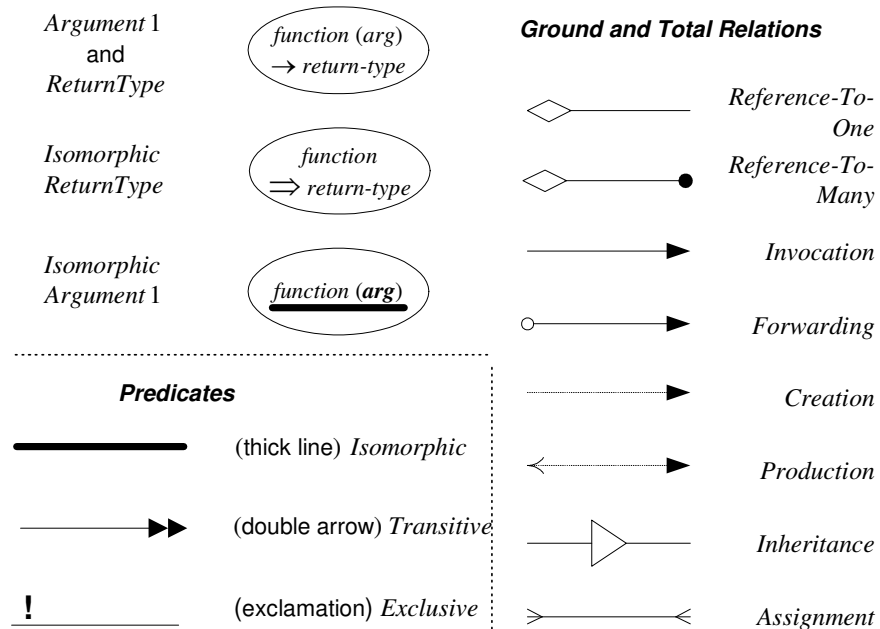


Figure 6. Graphic synonyms for predominant relations. The direction of the binary relation edges is left to right

Special edge styles are defined for prevailing relations (e.g., *Inheritance*) and adornments representing the different predicates (Definition IV). Thus, for instance, the isomorphism illustrated in Figure 2 is formally specified in Diagram 3, where *Creation* is represented as an arrow, whose shaft is thick to indicate that it is *isomorphic*.

A *diagram* may be circumscribed by the *commutative* designator, indicating that the isomorphisms thereof *commute* over the indicated variables.

$$\begin{aligned}
 \textit{Visit} &: \mathbf{P}^2(\mathbb{F}) & (20) \\
 \textit{Accept} &: \mathbf{P}(\mathbb{F}) \\
 \textit{Visitors, Elements} &: \mathbb{H}
 \end{aligned}$$

$\textit{Tribe}(\textit{Visit}, \textit{Visitors})$
 $\textit{Clan}(\textit{Accept}, \textit{Elements})$
 $\textit{FirstArg}^{\leftrightarrow}(\textit{Accept}, \textit{Visitors})$
 $\textit{FirstArg}^{\leftrightarrow}(\textit{Visit}, \textit{Elements})$
 $\textit{Invocation}^{\leftrightarrow}(\textit{Accept}, \textit{Visit})$
 $\textit{Invocation}^{\leftrightarrow}(\textit{Visit}, \textit{Elements})$
 $\textit{Commute}_{\textit{FirstArg}, \textit{Invocation}}(\textit{Visit}, \textit{Elements}, \textit{Visitors}, \textit{Accept})$

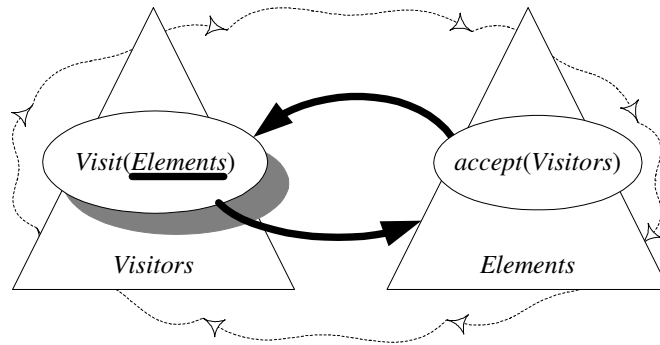


Diagram 5. *VISITOR* pattern

The visual formalism conveys powerful abstractions in a single picture. Consider, for example, the symbolic specification of the *VISITOR* pattern (20) vs. its visual representation (Diagram 5). While both representations convey the same information, the symbolic specification focuses separately on each properties, while the visual representation offers a broader “roadmap” of the pattern in one glance.

6. Applications

The proof of the pudding is in the eating and the proof of a specification language is in the applications. In this section, we discuss architectural and design problems to which

LePUS has already been applied and problems to which we anticipate LePUS will be applied in the future.

The main advantage of formalizing in logic is that we can reason about specifications and make formal deductions about their properties. In particular, we can validate designs by showing that they conform to particular patterns, as shown in Section 6.1. Other forms of reasoning, such as the analysis of relations between patterns, are demonstrated in the remaining subsections.

A crucial advantage of a formal language is that it provides a firm foundation for the construction of software tools. Our next major step will be the development of LePUS tools and, in the final part of this section, we outline a prototype for such a tool.

6.1 Reasoning

Perhaps the most significant advantage of a formal language is that it facilitates precise reasoning. Without a rigorous inference mechanism, arguments may be more or less convincing, but they will inevitably lack the certainty that is required in science and engineering.

LePUS formulae manifest a small subset of predicate calculus and consequently the language provides standard methods of logical inference. In practice, most of the reasoning that we actually perform in LePUS falls into a few categories, the most important of which are outlined below.

6.1.1 A Case Study in Validation

It is only natural that formalization should throw light on ambiguities that arise from verbal specifications. One such application is the *validation* that certain programs indeed implement the pattern as specified.

A generic specification in LePUS is a formula with free variables. If we assign particular values (“constants”) to the variables, we obtain a *proposition*, or a logical statement, which is either true or false. If it is true, the assigned values correspond to a valid instance of the specification.

For example, suppose that the specification contains the clause $DefinedIn(f, c)$, where f and c are free variables of types \mathbb{F} and \mathbb{C} respectively. Suppose we assign the function `f00` to f and the class `bar` to c . The value of the resulting statement is true depending whether the function `f00` actually is defined in the class `bar`.

More generally, when we have a concrete software system consisting of classes and their member functions, we can assign particular classes and functions from the system into a LePUS specification and thereby check whether the respective subsystem satisfies the specification. In such case we say that the specific subprogram *validates* the formula [4]. Following is a detailed example for this process.

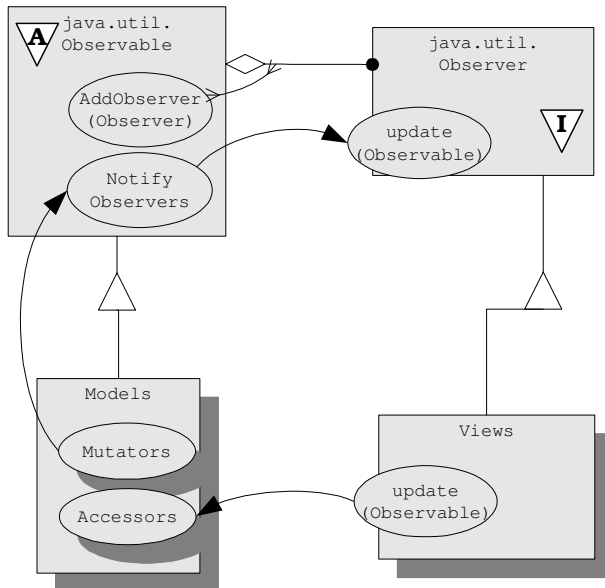


Diagram 6. Observer/Observable classes in Swing [11]

Since LePUS is defined in terms of symbolic logic, *validation* can be rephrased formally as follows: Given a certain implementation (represented by an n -tuple of elements), and a pattern (represented as a formula), we say that the given implementation *validates* the pattern if the assignment of the respective elements satisfies the formula.

Consider for example the Observer/Observable classes in the Swing class library [11], whose specification is expressed in LePUS format in Diagram 6. While the diagram can serve as a design document for the respective classes, it can also be used to prove in two simple steps to satisfy the *OBSERVER* pattern [9], depicted in (Diagram 7).

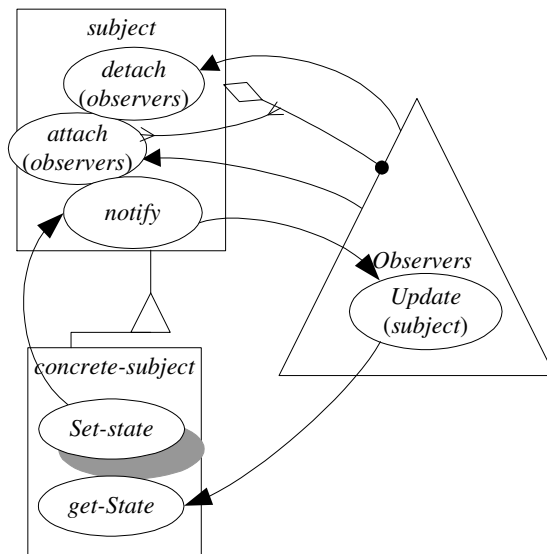


Diagram 7. Observer pattern

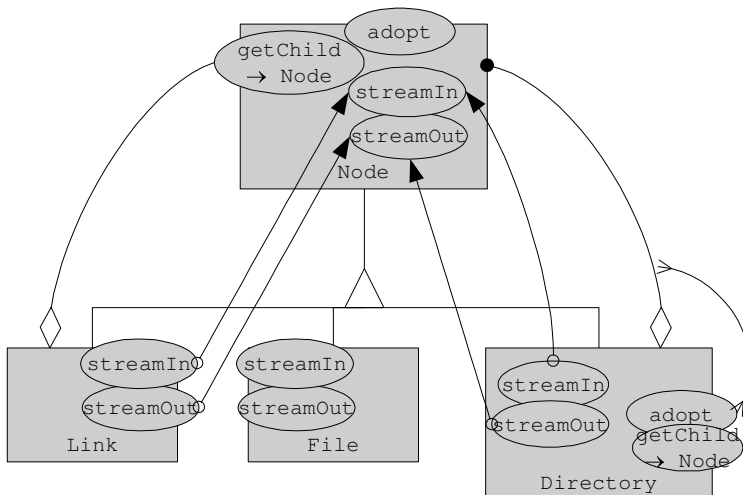


Diagram 8. LePUS Schema for the program in [12, p. 29]

- ◆ The set of Views, jointly with the Observer class, can be unified with the Observers hierarchy;
- ◆ Class Observable and the set Models can be unified with the variables *Subject*, *ConcreteSubject*, respectively.

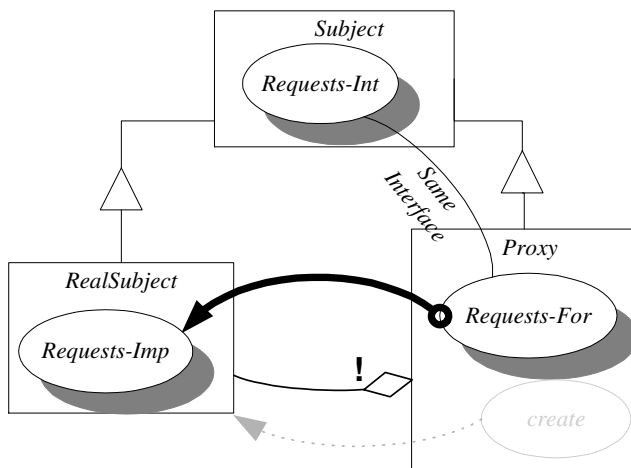


Diagram 9. PROXY pattern

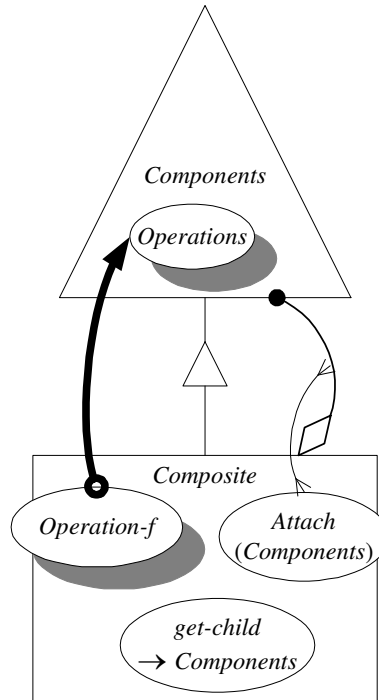


Diagram 10. *RECURSIVE COMPOSITE* pattern

In another example, consider the C++ program [12, p. 29] which demonstrates a combined implementation for two patterns: *RECURSIVE COMPOSITE* (Diagram 10) and *PROXY* (Diagram 9), both from the GoF catalog [9]. A visual depiction of the model for this program is given in Diagram 8. Assignment of the constant names in Diagram 8 to the formulae of the *PROXY* and the *COMPOSITE* is straight forward. From this point, the proof of its conformance to both patterns is trivial.

6.1.2. Relations Between Patterns

The first and simplest form of reasoning about LePUS formulae is strengthening and weakening. We strengthen a specification by adding clauses to it, with the effect that it claims more and applies to fewer actual architectures. Conversely, we weaken a specification by removing clauses from it so that it says less and applies to more architectures. In LePUS, strengthening is also called *refinement*.

We have used [13] strengthening and weakening to resolve a dispute on the following matter: is pattern *X* a special case of pattern *Y*? In LePUS, we can generalize this question in the following terms: can we obtain pattern *Y* by weakening *X* (i.e., removing clauses from its specification)?

The dispute in question was reported by Vlissides [14, 15]. The author proposes a new design pattern, *MULTICAST*, while a colleague maintains it is a special case of the *OBSERVER* [9]. The participants in this debate are the authors of the *OBSERVER*. Their dif-

faculty in reaching an agreement on the “true” meaning of their very own specifications only stresses how vague verbal specifications are.

To resolve this dispute, we formulated the description (Diagram 11) and showed that, in fact, Multicast is *not* a special case of the *OBSERVER* (Diagram 7), which stands in contrast with the agreement reached between the contestants. Put simply, *MULTICAST* is not a special case of the *OBSERVER* because some of the elements in the specification of the *OBSERVER* are missing from the *MULTICAST*.

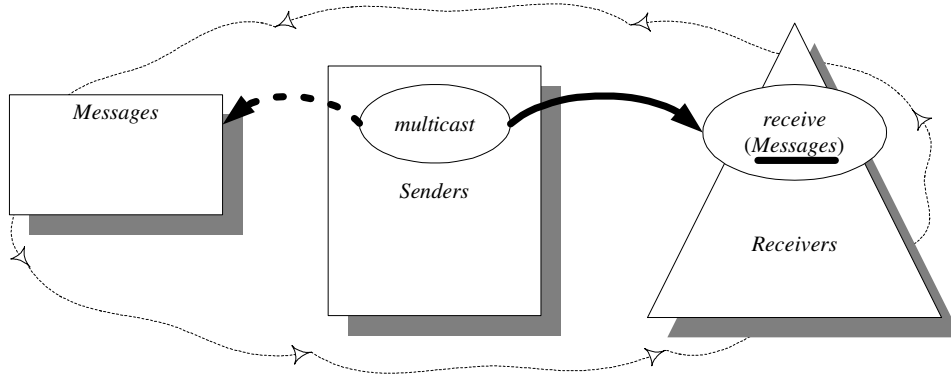


Diagram 11. *MULTICAST* pattern

The next form of reasoning is specifically facilitated by LePUS because it is based on simple manipulation of dimensions. If we replace a set in a specification by one of its elements (that is, an element with dimension one less than the original set, which may itself be a set) we obtain a *projection* of the original. The inverse operation is called *abstraction*, and consists of replacing an individual or set by a set of higher dimension.

As a simple example, abstracting a clan by replacing its set of functions by a set of sets of functions yields a tribe. Conversely, a tribe has several projections, each of which is a clan.

The *FACTORY METHOD* pattern (21), presented in Section 2, provides a more interesting example: *abstracting* the variables *FactoryMethods* and *Products* yields the *ABSTRACT FACTORY* pattern [9]. (Since the dimension of *FactoryMethods* has changed, the predicate on the first statement need also change to a *Tribe*.)

$$\begin{aligned} \text{FactoryMethods} &: \mathbf{P}(\mathbb{F}) \\ \text{Creators, Products} &: \mathbb{H} \end{aligned} \tag{21}$$

$$\begin{aligned} &\text{Clan}(\text{FactoryMethods}, \text{Creators}) \\ &\text{Create}^{\leftrightarrow}(\text{FactoryMethods}, \text{Products}) \\ &\text{ReturnType}^{\leftrightarrow}(\text{FactoryMethods}, \text{Products}) \\ &\text{Commute}_{\text{ReturnType}, \text{Create}}(\text{FactoryMethods}, \text{Products}) \end{aligned}$$

For those who are familiar with the specification of both patterns, this may come as a small surprise: While the relation between the patterns is acknowledged in the catalog, it is by no means made clear that it is such a simple relation. The precise nature of their relation is only made by their specification in LePUS.

Another example for the usefulness of a formal language can be made by pinning down informal observations on properties of patterns. For example, consider the following quote from the description of the *ITERATOR* pattern [9]:

CreateIterator is an example of a factory method (see Factory Method (107)). We use it here to let a client ask a list object for the appropriate iterator. ...

To support this statement we may observe the overlap between the *FACTORY METHOD* (21) and the first four clauses in the specification of the *ITERATOR* (22).

$$\begin{aligned}
 \text{Aggregates, Iterators} &: \mathbb{H} & (22) \\
 \text{Elements} &: \mathbf{P}(\mathbb{C}) \\
 \text{CreateIterator} &: \mathbf{P}(\mathbb{F}) \\
 \text{IterationOps} &: \mathbf{P}^2(\mathbb{F})
 \end{aligned}$$

$$\begin{aligned}
 &\text{Clan}(\text{CreateIterator}, \text{Aggregates}) \\
 &\text{Create}^{\leftrightarrow}(\text{CreateIterator}, \text{Iterators}) \\
 &\text{ReturnType}^{\leftrightarrow}(\text{CreateIterator}, \text{Iterators}) \\
 &\text{Commute}_{\text{ReturnType}, \text{Create}}(\text{CreateIterator}, \text{Aggregates}) \\
 &\text{Tribe}(\text{IterationOps}, \text{Iterators}) \\
 &\text{Reference}^{\rightarrow}(\text{Iterators}, \text{Elements}) \\
 &\text{ReferenceToMany}^{\rightarrow}(\text{Aggregates}, \text{Elements})
 \end{aligned}$$

6.2 Architectural Schemata

Garlan and Perry [16] define “Software Architecture” as follows:

The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.

We expect the documentation of the architecture of a concrete system (also *concrete architecture*) to manifest these properties. Of particular interest are architectural specifications of OO application frameworks and class libraries.

Parameterization is a key technique for abstraction. Frameworks, for example, are essentially parametrized architectures. LePUS is particularly suited to describing parametrized architectures because it uses variables in standard mathematical ways, as shown below.

A framework is a reusable, “semi-complete” application that can be specialized to produce custom applications. Documentation of frameworks is particularly challenging since it combines existing and anticipated elements: While some of the application logic is provided with the framework classes, its functionality is extended by the user (application developers), for instance, by inheriting from framework base classes.

Despite the intended focus on reusability, application programmers are required to have intimate knowledge of the internal structure of the framework they use. Hence, correct, accurate and comprehensible documentation is crucial. However, the only current means for documenting frameworks are specific “how-to-use” examples and diagrams in object notations whose flaws were discussed above.

Although class libraries are not as tightly coupled with the program code of client applications (as frameworks are required to), their documentation needs are similar: To depict the interaction between classes that are already provided, and classes that programmers of the client program should write.

In order to depict the relation between extant and anticipated constructs, accurate description of frameworks and libraries requires a combination of constant and variable symbols. This is one of the clear advantages of LePUS over Object notations, which allow only for constant symbols.

Finally, there is a strong case for providing visual specifications for concrete architectures, since graphic abstractions provide powerful mnemonics that can give a concise roadmap to the software system. Following is a sample architectural specification of such a system.

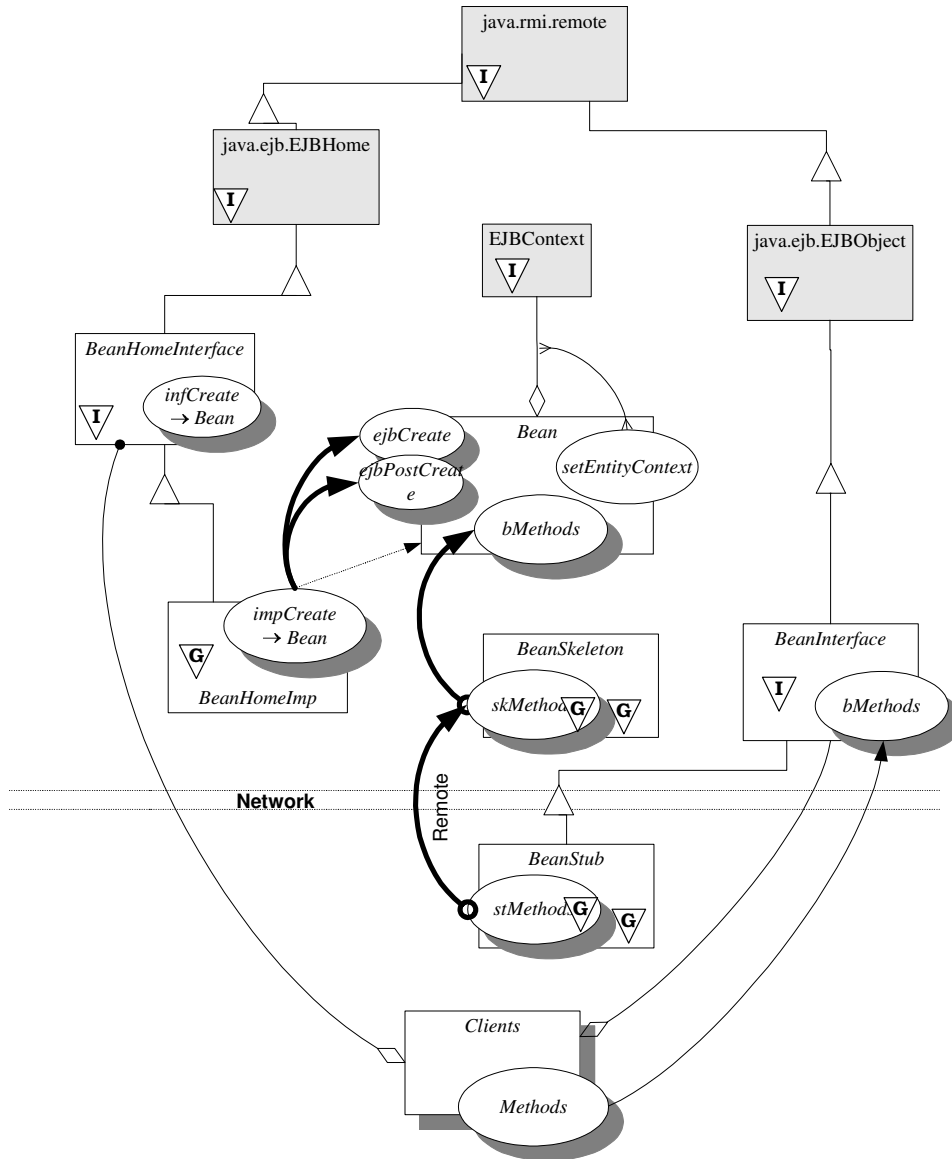


Diagram 12. LePUS schema of Enterprise JavaBeans™

Enterprise JavaBeans™ (EJB) is an environment that supports the development of distributed, server-side software. The server-side components are distributed objects hosted in “Enterprise Java Bean containers” which provide remote services for clients that are distributed throughout the network.

The specification of the EJB [17] “programming model” consists of numerous conventions, protocols and interfaces which make up the EJB API. The classes (“beans”) that application developers should write need conform to certain requirements so that the EJB

“container” can host and manage them. LePUS Schema of the EJB environment appears in Diagram 12. The diagram contains literals of three types:

- ◆ The constants designate concrete classes that are included in the Java EJB package, such as `java.rmi.remote` and `EJBContext`;
- ◆ Variables marked G for the unary predicate *AutoGenerated*, such as *BeanHomeImp* and *BeanStub*, designate classes that are generated by the EJB environment from the user part of the program;
- ◆ The remaining variables designate the classes that the application developer should program, such as class *Bean*.

Table 3 contains selected extracts from the EJB documentation which were used to generate Diagram 12.

Table 3. Verbal specification of the Enterprise JavaBeans™ architecture

“Every bean obtains an <code>EJBContext</code> object, which is a reference directly to the container.”	$Reference(Bean, EJBContext)$
“The stub implements the remote interface so it looks like a business object.”	$Inherit(BeanStub, BeanInterface)$
“But the stub doesn’t contain business logic; ... Every time a business method is invoked on the stub’s remote interface, the stub sends a network message to the skeleton telling it which method was invoked.”	$Forward^{\leftrightarrow}(stMethods, skMethods)$
“When the skeleton receives a network message from the stub, it identifies the method invoked and the arguments, and then invokes the corresponding method on the actual instance.”	$Forward^{\leftrightarrow}(skMethods, bMethods)$
“A bean’s home interface may declare zero or more <code>create()</code> methods, each of which must have corresponding <code>ejbCreate()</code> and <code>ejbPostCreate()</code> methods in the bean class.”	$Tribe(InfCreate, EjbHomeInterface)$ $Tribe(ejbCreate, Bean)$ $Tribe(ejbPostCreate, Bean)$
“These creation methods are linked at runtime, so that when a <code>create()</code> method is invoked on the home interface, the container delegates the invocation to the corresponding <code>ejbCreate()</code> and <code>ejbPostCreate()</code> methods on the bean class.”	$Invokes^{\leftrightarrow}(impCreate, ejbCreate)$ $Invokes^{\leftrightarrow}(impCreate, ejbPostCreate)$

6.3 Tool Support

The usefulness of specification languages, as well as of formalisms, is ultimately tested in their pragmatics. One important use is the automation and semi-automation of pattern related activities.

The logical substructure of LePUS makes a logic language such as PROLOG a very natural medium for tool implementation. In fact, developing a prototype in PROLOG of a tool support for LePUS specification has proved to be most straightforward [3]. More specifically, LePUS specifications straightforwardly transcribe into structural statements, defining the constructs that a tool manipulates. Furthermore, since LePUS formulae are

purely symbolic logic statements, they easily translate into PROLOG *predicates*. Table 4 demonstrates the straightforward translation of the *FACTORY METHOD* pattern (Diagram 4) into PROLOG.

Table 4. PROLOG definition of the *FACTORY METHOD* pattern

```
FactoryMethod(hierarchy_structure(Creators_root, Creators_Leaves),
             hierarchy_structure(Products_root, Products_Leaves),
             FactoryMethods):-
    clan(FactoryMethods,
         [Creators_root|Creators_Leaves]),
    isomorphic(production, FactoryMethods,
              [Products_root|Products_Leaves]).
```

Since set relations in LePUS are either *total* or *isomorphic*, it is sufficient to code one rule that generalizes any given ground relation to its *isomorphic* counterpart, and another rule that defines a given ground relation to *total*. Table 5 delivers the PROLOG rule for *isomorphic*.

Table 5. PROLOG definition of *isomorphic* (Definition VI)

```
isomorphic(_, [], []) :- !.

isomorphic(Relation, L1, L2) :-
    member(E1, L1),
    member(E2, L2),
    Relation(E1, E2),
    Remainder1 = minus(L1, [E1]),
    Remainder2 = minus(L2, [E2]),
    isomorphic(Relation, Remainder1, Remainder2),
    !.
```

In our implementation, we used a statically typed, object-oriented version of PROLOG [18]. More details on the implementation are provided in [3].

It is important to note that a significant issue in tool support is the question of the representation of programs as a set of *facts* that can be manipulated by PROLOG predicates. More generally, it is the question of mapping constructs of programming languages into a logic *model* (Definition I), discussed in section 3. However, as demonstrated in [3], the problem proves to have a simple solution, as each ground relation in the representation of the GoF patterns [9], for instance, map directly to a well defined set of syntactic constructs in statically typed languages such as C++, Java™, and Eiffel.

To summarize its contribution, our tool supports the following activities:

- ◆ *Validation* tests whether a particular piece of code is indeed an instance of a given formula. By representing a program as a model (Definition I), and a pattern as a LePUS formula, a *validation* question is equivalent to the result of a PROLOG query which attempts to unify the atoms with the respective PROLOG predicate.
- ◆ *Recognition* is a task defined as the search for the presence of atoms that will validate a formula. In PROLOG this search is equivalent to the search in a database containing *facts*. For this purpose we have employed another aspect of PROLOG –

execution of a *query* with variables rather than atoms. A variable in a PROLOG *query* is interpreted as a request to match all elements of the database that satisfy the rule that may successfully replace this variable through *unification*.

- ◆ *Application* of a formula means a modification of a program in such a way as to satisfy the formula. As a result of application of a pattern ϕ we expect the program to incorporate an instance of ϕ . *Application* involves a mechanism that is only a slightly different from the ones used for *validating* or *recognition*. While the PROLOG *goal* in *application* tasks incorporates a *predicate*, some of the actual arguments passed to the query may designate *atoms* and *facts* that do not exist in the database. These indicate that the program must be changed to incorporate the missing atoms, and to add the missing facts.

A formal definition of *validation*, *recognition*, and *application* operations with respect to a logic structure are described in detail in [4].

7. Related Work

In this section we discuss the relevant notations, formalisms, and tools, and show which of the desirable properties of a specification language (Section 1) are not satisfied.

Object Notations. Since object notations [19, 20, 21, 22] are popular means for OO design, they are the first potential candidate for architectural specifications. Object notations, however, and UML [23] in particular, suffer from a number of shortcomings in this respect:

- (i) According to the *principle of least constraint* (req. i), architectural specification describe the constraints to the desired level rather than point to a specific solution. However, object notations were designed specifically for that purpose, namely, for describing the design of a concrete software system.
Consequently, object notations incorporate only constant symbols, no variables or quantifiers. For example, a constraint such as *the first formal argument of f is of class c* cannot be expressed by object notations.
- (ii) Object notations can only account for first order sets and their members and relationships among them, while the specification of patterns must account for the occurrence of relations between sets of higher dimension (Section 3.2).
- (iii) Object notations incorporate representations for syntactical elements of object languages, such as objects, classes, and attributes, while architectural relations are at a higher level of abstraction.
- (iv) Object notations do not represent procedures (also *function members*, *methods*) as first-class entities but instead as elements of classes. Consequently, procedures cannot be characterized as necessary and their key properties can only be specified informally. For instance, the only way to express the following sample facts in an Object notation is by means of informal notes: “Method f_2 invokes another method f_1 ”, or “Method f_2 incorporates a statement which creates an object of class c ”.

Design patterns' formalisms. Formal Languages for the specification of design patterns are described in several publications. Helm, Holland and Gangopadhyay [24] defined “Contracts”, an extension to first order logic with representations for function calls, assignments, and an ordering relation between them. The “behavioral compositions” described do not address structural relations but only run time (“behavioral”) characterizations.

Tool Support. Florijn, Meijers, and van Winsen [25] propose a tool that supports the application of design patterns. They use the “Fragments Model” for the representation of patterns, a graph with labeled arcs, whose nodes stand for the *participants* in the pattern's leitmotif, and its arcs describe the roles of the connecting nodes. The *patterns' wizard* [26] supports the *application* of patterns, represented as metaprogramming algorithms which describe the sequence of steps in the application of the respective pattern. Other tools supporting the application of design patterns are described in [27, 28, 29, 30, 31, 32, 33, 34], most of which are reviewed in [26].

Architecture Description Languages. It is generally accepted [35] that coherent architectural specifications warrant the recognition of underlying constructs, elementary building blocks, and repeating abstractions. In the case of OO programs, natural candidates for such constructs are *inheritance hierarchies* and *function families* (Section 3.2). Nonetheless, research in software architecture and in *Architecture Description Languages* (ADLs) [36] largely ignored the OO idiosyncrasies. Few works observe a sufficiently expressive yet compact set of elementary building blocks, or capture the microarchitectures observed in *design* and *architecture* patterns. As a result of this oversight, any attempt to capture the architecture of OO systems using contemporary architectural specification languages is destined to neglect key regularities in their organization.

Observations of Ground Relations. Other works have observed “elementary” relations between entities that can be used to document OO architecture. Keller et. al [37] list *Call Actions* and *Create Actions* (corresponding to *Invoke* and *Create* relations discussed in Section 3, respectively) among the information their reverse engineering environment maintains for the representation of OO programs, as well as information on the relations between classes (e.g., *Inheritance*, *Reference*). Chiba [38] describes a metaprogramming environment for manipulating C++ programs whose features are classified under *Function Invocation* and *Object Creation*, among other features.

Acknowledgements

We thank Prof. Yoram Hirshfeld of the Department of Mathematics, Tel Aviv University, for his significant contribution to this work.

Appendix

We include the formal definitions for concepts mentioned in this article. Constant names appear in `fixed font`. \mathbb{C} and \mathbb{F} designate the domains of ground classes and ground functions respectively. We use the names C^d, C_1^d, C_2^d, \dots to denote classes of dimension d , and the names $F^d, F_1^d, F_2^d \dots$ represent functions of dimension d . The transitive closure of a binary relation \mathcal{R} , is denoted as \mathcal{R}^+ .

Axiom 1

For all classes c , functions $f_1 \neq f_2$, one of the following conditions much be false:

- ◆ $DefinedIn(f_1, c)$
- ◆ $DefinedIn(f_2, c)$
- ◆ $SameSignature(f_1, f_2)$

Axiom 2

- (i) $Inherit^+$ (the transitive closure of the binary relation $Inherit$) is antisymmetric. This means that there cannot exist classes c and d such that both $Inherit^+(c, d)$ and $Inherit^+(d, c)$ are true.
- (ii) $SameSignature$ is an equivalence relation, i.e., it is –
 - ◆ Reflexive: $SameSignature(f, f)$
 - ◆ Symmetric: $SameSignature(f_1, f_2) \Rightarrow SameSignature(f_2, f_1)$
 - ◆ Transitive: If $SameSignature(f_1, f_2)$ and $SameSignature(f_2, f_3)$ then $SameSignature(f_1, f_3)$

Definition I: Model

A *model* is a pair $m=(\mathbb{U}, \mathbb{R})$ such that –

- (i) \mathbb{U} is a set of *ground entities*, partitioned into the domains \mathbb{C} and \mathbb{F} .
- (ii) \mathbb{R} is a set of ground relations $\{\mathcal{R}_1, \mathcal{R}_2, \dots\}$ that include the relations:
 - ◆ $Abstract \subset \mathbb{U}$
 - ◆ $SameSignature \subset \mathbb{F} \times \mathbb{F}$
 - ◆ $Inherit \subset \mathbb{C} \times \mathbb{C}$
 - ◆ $DefinedIn \subset \mathbb{F} \times \mathbb{C}$
- (iii) m satisfies Axiom 1 and Axiom 2.

Definition II: Uniform Set

- (i) A set of ground classes (functions) is designated a *uniform set of dimension 1*.
- (ii) A set whose elements are uniform sets of classes (functions) of dimension d is a uniform set of dimension $d+1$.

A uniform set of classes (functions) of dimension d is also called a *class (function) of dimension d* . When it is clear from the context, the term *set* is short for *uniform set*.

$\mathbf{P}^d(\mathbb{C})$ designates the domain of uniform sets of classes of dimension d and, symmetrically, $\mathbf{P}^d(\mathbb{F})$.

Definition III: Well-Formed Formula

Formulae may contain the following:

- ◆ *Variables* which range over *uniform sets* (Definition II)
- ◆ *Predicates* (Definition IV)
- ◆ *Operators* (Definition XIII)

A formula is well formed if the arguments of the predicate and operator symbols are of the appropriate type and dimension.

Definition IV: Predicates

Predicates are subsets defined by their properties. LePUS defines the following predicates:

Symbol	Domain	Definition
<i>Hierarchy</i>	$\mathbf{P}(\mathbb{C})$	Definition IX
<i>Clan</i>	$\mathbf{P}^n(\mathbb{F}) \times \mathbf{P}^n(\mathbb{C})$	Definition X
<i>Tribe</i>	$\mathbf{P}^{n+1}(\mathbb{F}) \times \mathbf{P}^n(\mathbb{C})$	Definition XI
<i>Total</i>	$\mathbb{R} \times \mathbf{P}^n(\mathbb{U}) \times \mathbf{P}^n(\mathbb{U})$	Definition V
<i>Isomorphic</i>	$\mathbb{R} \times \mathbf{P}^m(\mathbb{U}) \times \mathbf{P}^n(\mathbb{U})$	Definition VI
<i>Exclusive</i>	$\mathbb{R} \times \mathbf{P}^m(\mathbb{U}) \times \mathbf{P}^n(\mathbb{U})$	Definition VII
<i>Commute</i>	$\mathbb{R} \times \mathbb{R} \times \mathbf{P}^m(\mathbb{U}) \times \mathbf{P}^n(\mathbb{U})$	Definition VIII

Definition V: Total Predicate

Given a binary relation \mathcal{R} , and two sets V^m and W^n of dimensions m, n respectively, we define the predicate $\mathcal{R}^{\rightarrow}(V^m, W^n)$, read “ \mathcal{R} is total in V^m, W^n ”, as follows:

- ◆ $m=n=0$ $\mathcal{R}(V^m, W^n)$
- ◆ $m>0, n=0$ $\forall V^{m-1} \in V^m \bullet \mathcal{R}^{\rightarrow}(V^{m-1}, W^n)$
- ◆ $m=0, n>0$ (Symmetrically)
- ◆ $m, n>0$ $\forall W^{n-1} \in W^n \quad \exists V^{m-1} \in V^m \bullet \mathcal{R}^{\rightarrow}(V^{m-1}, W^{n-1})$
(i.e., $\mathcal{R}^{\rightarrow}$ is a total function in V^m, W^n)

Definition VI: Isomorphic Predicate

Given a binary relation \mathcal{R} , and two sets V^m and W^n of dimensions m, n respectively, we define the predicate $\mathcal{R}^{\leftrightarrow}(V^m, W^n)$, read “ \mathcal{R} is isomorphic in V^m, W^n ”, as follows:

- ◆ $m=n=0$ $\mathcal{R}(V^m, W^n)$
- ◆ $m>0, n=0$ $\forall V^{m-1} \in V^m \bullet \mathcal{R}^{\leftrightarrow}(V^{m-1}, W^n)$
- ◆ $m=0, n>0$ (Symmetrically)
- ◆ $m, n>0$ $\forall V^{m-1} \in V^m \quad \exists! W^{n-1} \in W^n \bullet \mathcal{R}^{\leftrightarrow}(V^{m-1}, W^{n-1})$ and
 $\forall W^{n-1} \in W^n \quad \exists! V^{m-1} \in V^m \bullet \mathcal{R}^{\leftrightarrow}(V^{m-1}, W^{n-1})$
(i.e., $\mathcal{R}^{\leftrightarrow}$ is a bijective function in V^m, W^n)

Definition VII: Exclusive Predicate

Given a binary relation \mathcal{R} and variables V, W , we define the predicate *left-exclusive*, written $\mathcal{R}(V!, W)$ as follows:

$$\mathcal{R}(V, W) \wedge [\forall v \bullet \mathcal{R}(v, w) \wedge (w \in W) \rightarrow (v \in V)]$$

And symmetrically for the predicate *right-exclusive*: $\mathcal{R}(V, W!)$.

Definition VIII: Commute predicate

We say that the isomorphic relations \mathcal{R}, \mathcal{S} commute in the sets dom, ran , written as $\text{Commute}_{\mathcal{R}, \mathcal{S}}(\text{dom}, \text{ran})$, to express that for all d in dom and r in ran , $(d, r) \in \mathcal{R}$ iff $(d, r) \in \mathcal{S}$.

Definition IX: Hierarchy predicate

A uniform set of classes h is said to be a *hierarchy* if it contains a class r such that:

$$Abstract(r) \wedge \forall c \in h, c \neq r \bullet Inherit^+ \rightarrow (c, r)$$

Definition X: Clan predicate

F^d is a clan in C^d if the following condition holds:

$$DefinedIn^{\leftrightarrow}(F^d, C^d) \wedge (\forall F_1^{d-1}, F_2^{d-1} \in F^d \bullet SameSignature^{\rightarrow}(F_1^{d-1}, F_2^{d-1}))$$

Definition XI: Tribe predicate

F^{d+1} is a tribe in C^d iff all F^d in F^{d+1} are clans in C^d .

Definition XII: Pre-Method, Signature

Since *SameSignature* is an equivalence relation (Axiom 2), it induces a partition on \mathbb{F} , and we may define the following:

- (i) \mathbb{P} designates the quotient set $\mathbb{F} / SameSignature$. The sets in \mathbb{P} are called *pre-methods*.
- (ii) A *representative* s for a pre-method $p \in \mathbb{P}$ is called a *signature*.

Definition XIII: Selection Operator

Given a pre-method p and a set of classes C^d , we define the expression $p \otimes C^d$ as the following set of functions:

- ◆ $d=0$ $\{ f \mid f \in p \wedge DefinedIn(f, C^d) \}$
and by Axiom 1 there can be at most one such function
- ◆ $d>0$ $\{ p \otimes C^{d-1} \mid C^{d-1} \in C^d \}$

Given a set of pre-methods P and a set of classes C^d , we define $P \otimes C^d$ as the following set of functions:

$$P \otimes C^d \equiv \{ F^d \mid p \otimes C^d \wedge p \in P \}$$

References

-
1. D. E. Perry, A. L. Wolf. "Foundation for the Study of Software Architecture." *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No. 4, Oct. 1992, pp. 40-52.
 2. L. Bass, P. Clements, R. Kazman (1998). *Software Architecture in Practice*. Reading, MA: Addison Wesley Longman, Inc.
 3. A. H. Eden (2000). "Precise Specification of Design Patterns and Tool Support in Their Application," PhD diss., Department of Computer Science, Tel Aviv University.
 4. A. H. Eden, Y. Hirshfeld. "Towards a Formal Foundation for Object Oriented Architecture." *Submitted: Formal Methods Europe*, Mar. 12-16, 2001. Humboldt Universität zu Berlin.
 5. J. Coplien, D. Schmidt, eds. (1995). *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley.
 6. J. M. Vlissides, J. O. Coplien, N. L. Kerth (1996). *Pattern Languages in Program Design 2*. Reading, MA: Addison-Wesley.
 7. R. Martin, D. Riehle, F. Buschmann, eds. (1997). *Pattern Languages of Program Design 3*. Reading, MA: Addison-Wesley.
 8. D. Schmidt, M. Stal, H. Rohnert, F. Buschmann (2000). *Pattern-Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons, Ltd.
 9. E. Gamma, R. Helm, R. Johnson, J. Vlissides (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, MA: Addison-Wesley.
 10. M. Jackson (1995). *Software Requirements and Specifications*. Reading, MA: Addison Wesley.
 11. K. Walrath, M. Campione (1999). *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Reading, MA: Addison-Wesley.
 12. J. Vlissides (1998). *Pattern Hatching*. Addison-Wesley.
 13. A. H. Eden, Y. Hirshfeld, A. Yehudai. "Multicast - Observer <> Typed Message." *C++ Report*, Vol. 10, No. 9, October 1998, pp. 33-39.
 14. J. Vlissides (1997). "Multicast". *C++ Report*, Vol. 9, No. 8, Sep. 97. New York: SIGS Publications.
 15. J. Vlissides (1997). "Multicast - Observer = Typed Message". *C++ Report*, Vol. 9, No. 9, Nov.-Dec. 97. New York: SIGS Publications.

-
16. D. Garlan, D. E. Perry. "Introduction to the Special Issue on Software Architecture." *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, April 1995, pp. 269-274.
 17. V. Matena, M. Hapner (1999). *Enterprise JavaBeans™ Specification*, v1.1. Palo Alto, CA: Sun Microsystems.
 18. Visual Prolog 5.0 (1997). Prolog Development Center A/S.
<http://www.pdc.dk>
 19. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen (1991). *Object Oriented Modeling and Design*. New Jersey: Prentice Hall.
 20. G. Booch (1994). *Object Oriented Analysis and Design with Applications*, 2nd edition. Redwood City, CA: Benjamin/Cummings.
 21. K. Walden, J. M. Nerson (1995). *Seamless Object-Oriented Software Architecture*. New Jersey: Prentice Hall.
 22. D. G. Firesmith, I. Graham, B. Henderson-Sellers (1999). *Open Modeling Language (OML) Reference Manual*. Cambridge: Cambridge University Press.
 23. G. Booch, I. Jacobson, J. Rumbaugh (1999). *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley.
 24. R. Helm, I. M. Holland, D. Gangopadhyay (1990). "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems." *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 21 - 25, 1990, pp. 169-180. Ottawa, Canada.
 25. G. Florijn, 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.
 26. A. H. Eden, J. Gil, A. Yehudai (1997). "Precise Specification and Automatic Application of Design Patterns." *Proceedings of the Twelve IEEE International Automated Software Engineering Conference (ASE 1997), Lake Tahoe, Nevada*, Nov. 3-5, 1997, pp. 143-152. Los Alamos: IEEE Computer Society Press.

-
27. P. P. Pal. "Law-Governed Support for Realizing Design Patterns." *Proceedings of 17th Conference on Technology of Object-Oriented Languages and Systems (TOOLS 17)*, 1995. New Jersey: Prentice Hall.
 28. F. J. Budinsky, M. A. Finnie, J. M. Vlissides, P. S. Yu (1996). "Automatic code generation from design patterns". *Object Technology*, Vol. 35, No. 2.
 29. Quintessoft Engineering, Inc. (1997). C++ Code Navigator 1.1. <http://www.quintessoft.com>
 30. K. Brown (1996). "Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk." M. Sc. thesis, University of Illinois.
 31. P. Alencar, D. Cowan, J. Dong, C. Lucena. "A Pattern-Based Approach to Structural Design Composition." *IEEE 23rd Annual International Computer Software and Application Conference*, October 1999, pp. 160-165.
 32. C. Kramer, L. Prechelt. "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software." *Proceedings of the Working Conference on Reverse Engineering*, November 8-10, 1996, Monterey, CA, pp. 208-215.
 33. J. Bosch. "Relations as Object Model Components." *Journal of Programming Languages*, Vol. 4, No. 1, 1996, pp. 39-61.
 34. M. O'Cinnéide, P. Nixon. "A Methodology for the Automated Introduction of Design Patterns." *Proceedings of the IEEE International Conference on Software Maintenance*, 30 August - 3 September, 1999.
 35. G. Odenthal, K. Quibeldey-Cirkel (1997). "Using Patterns for Design and Documentation". *Proceedings of the European Conference of Object Oriented Programming 1997*. Lecture Notes in Computer Science. Berlin: Springer.
 36. N. Medvidovic, R. N. Taylor (1997). "A Framework for Classifying and Comparing Architecture Description Languages". In *Proceedings of the Sixth European Software Engineering Conference*, together with *Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 60-76, Zurich, Switzerland, September 22-25, 1997.
 37. R. K. Keller, R. Schauer, S. Robitaille, P. Page. "Pattern-based Reverse Engineering of Design Components." In: *Proceedings of the Twenty-First International Conference on Software Engineering*, pages 226-235, Los Angeles, CA, May 1999. IEEE.

-
38. S. Chiba (1995). "A Metaobject Protocol for C++." *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 15 - 19, 1995, pp. 285-299. Austin, TX USA.