Note: This manuscript contains minor amendments made to the published version (as included in the conference proceedings) and which do not appear therein.

Integrated Design and Process Technology, **IDPT-2002**
Printed in the United States of America, June, 2002
© 2002 Society for Design and Process Science

A VISUAL FORMALISM FOR OBJECT-ORIENTED ARCHITECTURE

Amnon H. Eden

Department of Computer Science, Concordia University, Montreal, Canada

eden@acm.org

ABSTRACT

We present a visual formalism for the specification of object-oriented design and architectures. LePUS diagrams effectively capture and convey a concise description of design patterns. LePUS diagrams also offer lucid "roadmaps" ("architectural schemata") for elaborated libraries and frameworks, and make a documentation tool that is more powerful, concise, and comprehensive than existing techniques. We demonstrate the formalism's merit in representing the *Observer* design pattern, as well as manifesting idioms in *Enterprise JavaBeans*™ and Java™ *Swing*.

Keywords: Software design theory, object-oriented programming, visualization.

1. INTRODUCTION

Progress was made in the understanding of object-oriented (O-O) software design and architectures through the introduction of *patterns* [Gamma et. al 94]. Few results, however, manifest the fundamental building blocks of O-O design and architecture [Eden & Hirshfeld 01]. Formalisms proposed for the specifications of design patterns [Lauder & Kent 98; Mikkonen 98] do not account for recurring motifs in O-O design and do not offer a sufficiently expressive vocabulary to manifest the majority of the accepted patterns.

Having been designed to support the development process of a *specific* program, Object notations [Rumbaugh et. al 91; Booch et. al 99; Firesmith et. al 99] invariably suffer from three prominent shortcomings:

- ♦ ***Informal specifications.*** Not one of the aforementioned notations is defined formally, nor provided with denotational semantics, nor founded on axiomatic semantics. As their interpretation founded exclusively on idiosyncratic intuitions[i], tools' support are of limited use and little reasoning can be performed on their contents.
- ♦ ***Abstraction level***. Diagrams represent low-level abstractions: specific classes and their members, and inheritance or other class-to-class relations. Hence, *class diagrams* are useful for illustrating details of small components of limited scope. Such diagrams cannot capture the roles and collaborations among components and modules of lower granularity levels.

  For the same reason, class diagrams do not provide hierarchically abstract decomposition of software except in terms of distinct modular units, which cover only one aspect of complexity.
- ♦ Lack of Generality. Object notations were designed to support software development for a program that has a given set of requirements. Consequently, each diagram contains constant symbols, not variables, and can depict a concrete plan, not a generic schema. In contrast, architectural specifications express *constraints* on desirable properties, not the specifics of their implementation [Perry & Wolf 92]. Clearly, such expressions require variables and quantifiers as well.

Finally, *architectural styles* [Garlan & Shaw 93] and formalisms thereof [Allen & Garlan 97; Luckham et. al 95] are yet to capture O-O idiosyncrasies, such as *higher dimension classes* (Definition I), *class hierarchies* (Definition II), *isomorphic relations* (Definition III), and *clans* (Definition V). These constructions are explained and demonstrated in Section 2, and formal definitions appear in the Appendix.

LePUS

LePUS [Eden 00; Eden 01] is a formal specification language for O-O design and architecture. LePUS is based on the observation of an underlying ontology, which consists a small number of well defined, necessary and sufficient "building blocks" that are ubiquitous to the design of every object oriented program.

In this article, we demonstrate the following uses for LePUS:

1. ***Generic Specifications***: Formal specifications of generic formations, such as *design patterns*, disambiguate informal, verbal descriptions and prevent confusions as to their actual intent. Generic specifications are illustrated using the *Observer* pattern.
2. ***Documentation*** (e.g., of frameworks and libraries): LePUS diagrams effectively capture and convey concise, lucid "roadmaps" to the architecture of O-O programs, allowing for "system browsing" at the appropriate level of abstraction. This aspect is illustrated via the specification of the *Enterprise JavaBeans*™ framework and in Section 4.
3. ***Reasoning*** on specifications, for the verification of implementations and on relations among them.

---

[i] That is, if source code is interpreted at all. Rational Rose, for instance, is a CASE tool which can interpret UML class diagrams as class declarations in Java and C++. The numerous additional parameters required by its *round trip engineering* process demonstrate the extent of which class diagrams are open for interpretations.

Having well-defined semantics, LePUS diagrams can be used as a foundation for tool support in architectural specifications, as demonstrated by Eden and Jahnke [02].

Overview

Section 2 introduces the ontology that underlies the notation and its semantics. Section 3 describes the visual formalism employing three detailed sample specifications. Formal definitions for both these chapters were mostly left to the Appendix. In Section 4 we describe the architectural "schema" of the *Enterprise JavaBeans*™ framework. In Section 5 we demonstrate the reasoning power of LePUS specifications with proving that Swing's *MVC* is a special case of the *Observer* design pattern. Section 6 concludes with future research directions.

2. The Underlying Ontology

In this section, we observe a set of the underlying abstractions, which are at the basis of every O-O design. This set of concepts also delivers well-defined semantics for LePUS.

Design Models

In comparison with the host of details that programs normally contain, architectural specifications take only a fraction in "volume." Aiming for abstraction, the complexity of the source code can be greatly simplified by "distilling" a picture of the program's basic properties into a collection of entities and relations, designated *design model*. More specifically, we suggest that programs are represented via a *structure* in mathematical logic [Barwise 77] $\mathbb{M} = \langle \mathbb{U}, \mathbb{R} \rangle$, which consists of –

1. A universe $\mathbb{U}$ of *ground entities* (*atoms*), each of which is of type *class* or *method* (also *function members*), and
2. $\mathbb{R}$ is a set of *ground relations* among the atoms.

Table 1 gives a trivial Java™ program and Figure 1 illustrates the ground entities and relations therein.

Eden and Hirshfeld [01] describe in detail the mapping relation between programs written in an OOP language $\mathbb{L}$ into a *design model*. We call such mapping *denotation*, designated $\mathbb{D}$.

Table 1. Implementation of the *Decorator* pattern

```
abstract class Decorator {
   abstract void Draw();
}

class BorderDecorator extends Decorator {
   void Draw() {
      // do_something();

      Decorator.Draw(); //...
   }
   int BorderWidth;
}
```

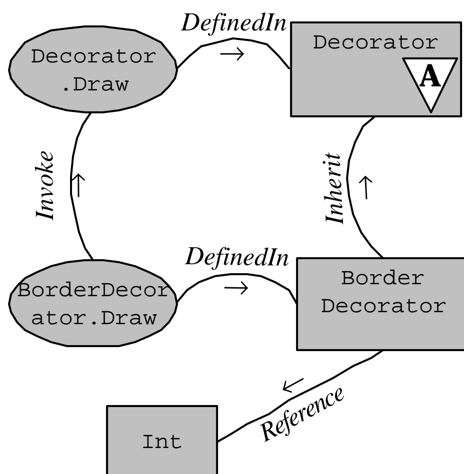

Figure 1. *Design model* of the program in Table 1

Dimensions

A ground class is also called a *class of dimension* $\mathit{0}$, and a set of classes of dimension $d\text{-}1$ is called a class of dimension d (Definition I, see Appendix.)

The simplest example is $\mathbb{C}$, the domain of ground classes, which is also a *1*-dimensional class in itself. Other domains of interest are $\mathbf{P}(\mathbb{C})$, the domain of *1*-dimensional classes, $\mathbf{P}^2(\mathbb{C})$, the domain of *2*-dimensional classes, and so on. By this notation, $\mathbf{P}^n(\mathbb{C}) \in \mathbf{P}^{n+1}(\mathbb{C})$.

We also define the set of *signatures*, $\mathbb{S}$. In programming languages, the *signature* of a method designates its name and argument types. A formal definition for $\mathbb{S}$ is given in the Appendix (Definition VI).

Class Hierarchies

In OOP, classes most often are organized in class hierarchies. In LePUS, we are interested in a specific kind of class hierarchies, defined as follows: A set of classes is a *hierarchy* (Definition II) if it contains an abstract "root" class, such that all other classes inherit (possibly indirectly) therefrom. In Section 3 we illustrate this definition with several examples.

We designate the domain of all *hierarchies* as $\mathbb{H}$. By definition, $\mathbb{H} \subset \mathbf{P}(\mathbb{C})$.

Function Families

Methods in O-O programs (also *function members*) are often organized in "families" (Definition V), related by their signature. The first kind of such families is a group of methods that share the same signature and that are defined in a group of classes. In such case we say that the set of methods F is a *clan in* C. Similarly, a set of clans of the same dimension $\{F_1, \ldots F_n\}$ that also share the same set of classes C is designated a *tribe in* C (Definition V.)

In our formalism, function families are represented using the *selection operator*, defined and illustrated in Section 3.3.

3. THE FORMALISM

LePUS (*LanguagE for Patterns Uniform Specification*) was defined [Eden 00] as a formal language for the specification of O-O patterns and architectures. LePUS is a compact subset of higher order logic, and is defined both as a visual and as symbolic language. In this article, we focus primarily on the visual formalism.

Due to lack of space, we provide only an overview of LePUS. A complete definition is provided in [Eden 01].

LePUS diagrams manifest terms and constraints thereon. More specifically, each symbol in a LePUS diagram represents a construction that falls under one of the following categories:

- *Terms* (*variables* and *constants*)
- *Relations*
- *Predicates*
- *Operators*

Figure 2 depicts the set of basic symbols used in LePUS diagrams. A legend for particular edge styles that represent relations (Section 3.2) is given in Figure 7 in the Appendix.

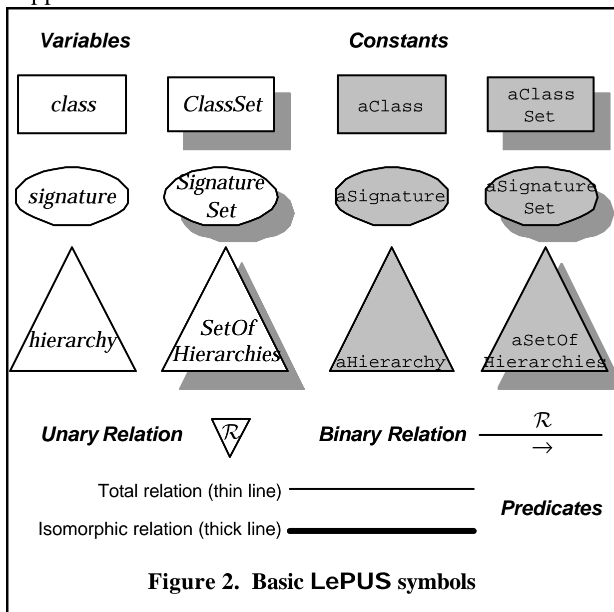

Figure 2. Basic LePUS symbols

In this section, we describe each set of symbols and illustrate their meaning using three examples: the *Model-View-Controller* "usage pattern" in Java™ *Swing* library (Figure 3); the *Observer* design pattern [Gamma et. al 94] (Figure 4); and a more elaborate example of *Enterprise JavaBeans*™ (Figure 6).

3.1 TERMS

LePUS diagrams consist of two kinds of terms: *variables* and *constants*. Each variable ranges over a given domain, such as the set of ground classes $\mathbb{C}$, and the set of *1*-dimension classes $\mathbf{P}(\mathbb{C})$, and so forth.

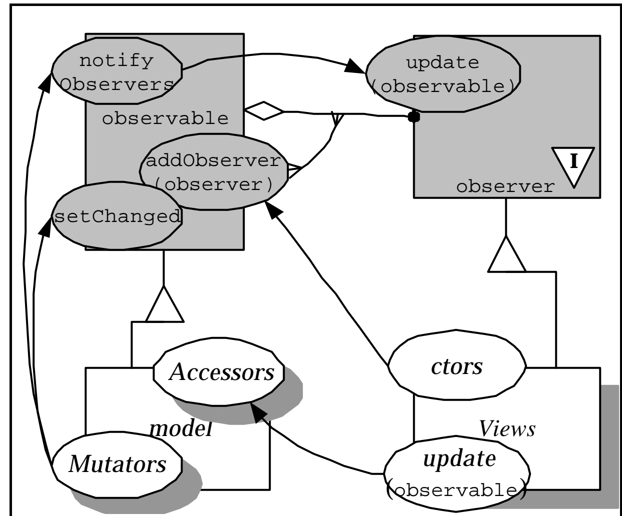

Figure 3. MVC "usage pattern" in Java™ *Swing*

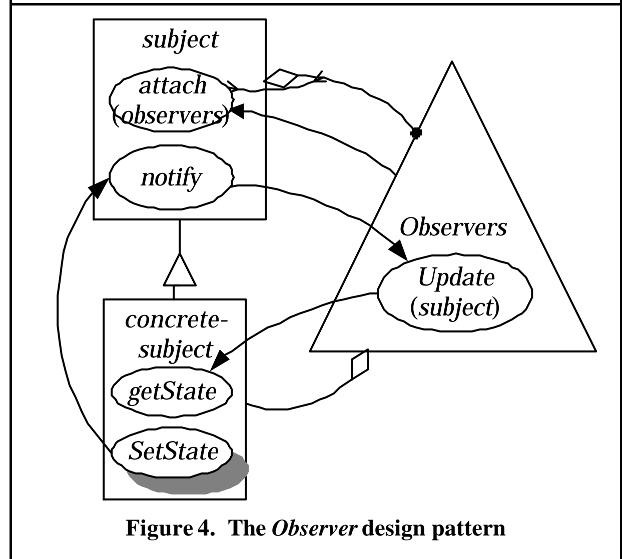

Figure 4. The *Observer* design pattern

For example, Figure 3 depicts a class variable *model*, which stands for a *0*-dimensional class, i.e., it ranges over ground classes. The same diagram also contains the *1*-dimensional class variable *Views*, which ranges over *1*-dimensional (i.e., sets of) classes.

In contrast, *constant* symbols represent specific entities. For example, each one of the class constants `observer` and `observable` depicted in Figure 3 represents a concrete class whose implementation is provided with the Java™ *Swing* class library. Class constants are comparable to class symbols in a UML *class diagram* [Booch et. al 99].

Constant symbols appear in `fixed-size` typeface printed over solid icons, while variable names appear in *italics* over transparent icons.

3.2 RELATIONS

LePUS diagrams include unary relations, such as *Interface*, and binary relations, such as *Inherit*. A *unary relation* is simply a set of ground entities that satisfy a particular property, for example, the set of ground entities *c* such that "*c* is an 'interface' class." A *binary relation* is a set of pairs of ground entities that satisfy a particular property, for example, the set of entities (d,b) such that "class d inherits from class b." These relations are essentially independent of any particular programming language (although the precise interpretation of a relation might depend somewhat on language details.)

For example, Figure 3 indicates that the unary relation *Interface* applies to class `observer`. The same diagram also implies that the pair *(model,* `observable`*)* is in the binary relation *Inherit*.

Particular edge styles were defined for some of the most common relations, such as *Inherit* and *Invoke*. A legend for these edge styles is given in Figure 7 in the Appendix.

3.3 OPERATORS

Methods are represented via the superimposition of *signature* symbols over class symbols. For example, consider the signature symbol *notify* in Figure 4. The combination of the *notify* ellipse with the *subject* class represents a single method that is defined in class *subject* and has the signature *notify*. In other words, this combination of symbols represent a *clan* (Definition V) of the size of one method in *subject* with the signature *notify*.

Similarly, consider the "set of signatures" *Methods* in Figure 6, where it occurs in five different places. In each occurrence, the *Methods* ellipse is superimposed on a different class. Each "occurrence" represents a different *tribe* (Definition V), which results from the selection of all methods with a signature in *Methods* that are defined in the respective class.

Symbolically, superimposition is designated using the *selection operator* (Definition VII). The selection operator yields either a *clan* or a *tribe* (Definition V), depending on the dimension of the *signature* term.

3.4 PREDICATES

LePUS restricts the possible relations between sets into two types: Bijective (called *isomorphic*) and *total* functions. Predicates are used to express which one applies. Below we describe two predicates that were incorporated in LePUS.

Isomorphic

Consider the following description [Matena & Hapner 99] pertaining to Enterprise JavaBeans™. The following paragraph describes a "call forwarding" relation between the methods defined in class *BeanHomeImp* and the methods defined in the *bean* class:

> *... when a* `create()` *method is invoked on the home interface, the container delegates the invocation to the corresponding* `ejbCreate()` *and* `ejbPostCreate()` *methods on the bean class.*

Note that this description is not restricted to just one `create` method but to any number thereof. Thus, we conclude that for every method c in the set $Create \otimes BeanHomeImp$ there is exactly one method ec in the set of methods $ejbCreate \otimes Bean$ such that –

$$Forward(c,ec)$$

We say that the *Forward* relation between the sets of methods $Create \otimes BeanHomeImp$ and $ejbCreate \otimes Bean$ is an *isomorphism* (Definition III), written

$$Isomorphic(Forward, Create \otimes BeanHomeImp, ejbCreate \otimes Bean)$$

or, using the shorthand defined in LePUS for *Isomorphic*,

$$Forward^{\leftrightarrow}(Create \otimes BeanHomeImp, ejbCreate \otimes Bean) \quad \textbf{(1)}$$

Similarly, we conclude that the *Forward* relation between the sets $Create \otimes BeanHomeImp$ and $ejbPostCreate \otimes Bean$ is also an isomorphism, or

$$Forward^{\leftrightarrow}(Create \otimes BeanHomeImp, ejbPostCreate \otimes Bean) \quad \textbf{(2)}$$

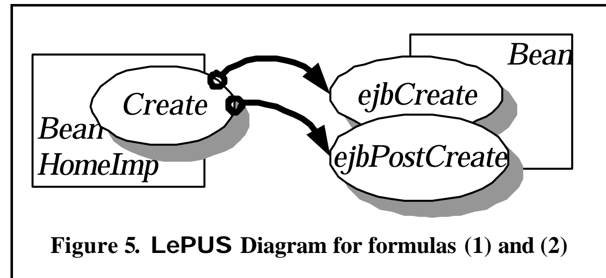

Figure 5. LePUS Diagram for formulas (1) and (2)

Visually, an *Isomorphic* relation is indicated by thick shaft. Thus, Figure 5, which depicts the LePUS diagram for formulas (1) and (2), contains an edge that combines the style for *Forward* relation (circle on left-hand argument, arrow on the right-hand argument) with the *Isomorphic* designation (thick shaft).

Total

The second kind of binary relations between sets is characterized as *total functions* (Definition IV). Consider, for example, the following description of the *MVC* "usage pattern" in *Swing* [Walrath & Campione 99]: Each one of the "mutator" methods defined in class "model" must invoke the method `setChanged`, which is defined in class

`Observable`. Formally, we say that the relation *Invoke* between the sets *Mutators*⊗*model* and `setChanged`⊗`observable` is a *total function*, written[i]

Total(Invoke,
Mutators⊗*model,*
`setChanged`⊗`observable`*)*

or using the shorthand defined in LePUS,

$$Invoke^{\rightarrow}(Mutators \otimes model, \quad \texttt{setChanged} \otimes \texttt{observable}) \qquad \textbf{(3)}$$

Visually, edges of *total* relations have thin shaft, as demonstrated in Figure 3, which contains the visual depiction of expression (3).

4. DOCUMENTATION

LePUS can be used to create *architectural schemata*, which provide "road-maps" for complicated systems. In this section, we demonstrate LePUS schemata in documenting an elaborate application framework.

Enterprise JavaBeans™ (EJB) is an environment that supports the development of distributed, server-side software. EJB "containers" provide remote services for clients that are distributed throughout the network. The framework is described as a "programming model" [Matena & Hapner 99] that encompasses several programming conventions and calling sequences (also *Application Programming Interface*.) Users of this library are programmers that need write classes ("beans") which must conform to certain requirements.

EJB constructions can be divided into three kinds: Parts that are already implemented, parts that need be written by the user/programmer, and code that is generated from the users' definitions. These parts need to cooperate closely in order to accomplish the intended tasks. Thus, the requirements from the programmer's code must be followed meticulously or else the program will not achieve the desired effect. The Extensive EJB documentation includes detailed descriptions in natural language and several examples in Java™ source code and class diagrams.

Figure 6 consists of a formal and comprehensive specification of the essentials of the EJB architecture. The diagram combines the parts that are already implemented with the parts that the programmer should write, as well as the parts that are generated from the programmer's specifications, in the following way:

- *Constants* represent classes whose implementation is provided with the Java EJB package, such as `EJBContext` and `java.rmi.remote`;

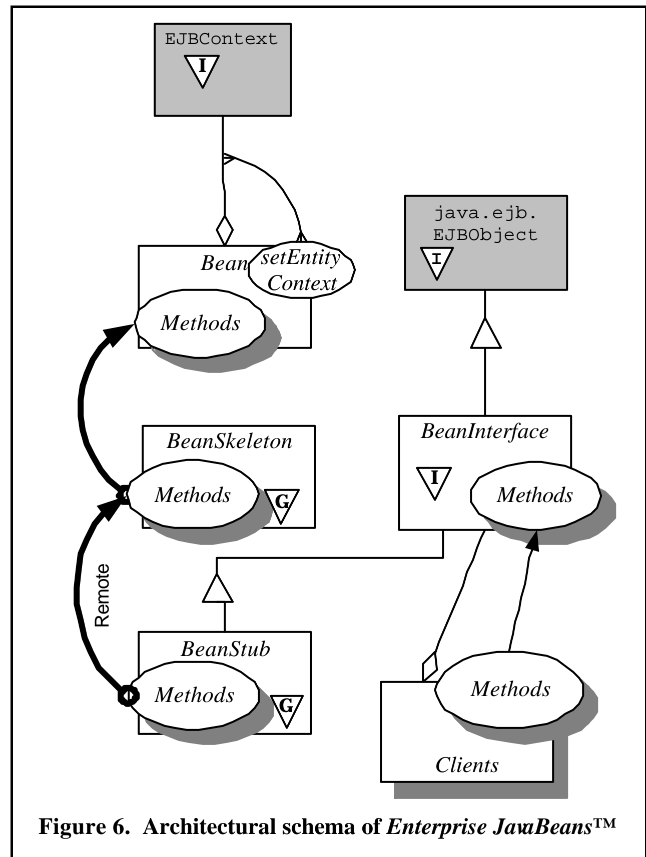

Figure 6. Architectural schema of *Enterprise JavaBeans*™

- *Variables* that are not marked with the unary predicate **G**, such as class *Bean*, designate entities that the application developer should implement him/herself;
- *Variables* marked with the unary predicate **G**, such as class *BeanSkeleton*, designate entities that are generated from the programmer's specifications.

We maintain that LePUS' *Schemata* offer a documentation scheme that is much more powerful, precise, and comprehensive than existing means.

5. REASONING

In this section, we demonstrate the capacity of reasoning with LePUS diagrams by proving that the MVC "usage pattern" (Figure 3) is a "special case" of (also: *refines*) the *Observer* pattern (Figure 4).

Consider the symbolic formulae *MVC* and *Obs*, which correspond to the LePUS diagrams in Figure 3 and Figure 4 respectively. Then:

Corollary 1.

$$MVC \vdash_{pc} Obs$$

which uses the symbol for *syntactic entailment*, representing the application of some proof theory for predicate calculus [Barwise 77].

---

[i] A ground term such as `setChanged`⊗`observer` is also considered as a singleton set.

In contrast with the richness of predicate calculus, we seek to introduce a more straightforward reasoning mechanism that will allow us to work directly with LePUS diagrams. Thus, we introduce the *unification* technique as a simpler entailment relation.

Informally, *unification* (Definition VIII) matches every element in one diagram with an element of a second diagram in a manner that must conform to the second. Table 2 demonstrates this mapping between Figure 3 and Figure 4.

Given the formal definition of *unification*, we can define a simpler form of syntactic entailment, written

$$\Delta \vdash_{u} \Gamma$$

which means that diagram Δ can be *unified* with diagram Γ using some *unification function* $\cup$.

Corollary 2.

$$MVC \vdash_{u} Obs$$

Proof: We define $\cup$ as the function that maps each element of the left column of Table 2 to the respective elements of the right column.

Table 2. Unifications from MVC into Observer

MVC	*Observer*
`observer`, *Views*	*Observers*
`update`, *update*	*Update*
`observable`	*Subject*
`addObserver`	*Attach*
`notifyObservers`	*Notify*
`model`	*ConcreteSubject*
Mutators	*SetState*
Accessors	*getState*

We show that $\cup$ is a valid unification function, by Definition VIII:

(i) Since the range of $\cup$ includes all the terms in *Obs*, $\cup$ is an *onto* function.

(ii) The only variables of *Obs* whose dimension is greater than *0* are *Observers* and *SetState*, whose dimension is *1*, and the dimension of the respective terms in *MVC* (namely, `observer`, *Views,* and *Mutators*) is *0* or *1*.

(iii) No constants are defined in *Obs*.

(iv) It is straightforward to observe that this condition holds for all predicates in *Obs*. For example, the predicate

$$Invoke^{\rightarrow}(SetState \otimes ConcreteSubject, notify \otimes subject)$$

is satisfied by the respective term (which is depicted in expression (3).)

(v) It is straightforward to observe that this condition holds for the two such predicates in *Obs*. For example, the predicate

$$ReferenceToMany^{\rightarrow}(subject, Observers)$$

is satisfied by the *MVC* predicate

$$ReferenceToMany^{\rightarrow}(\texttt{observable}, \{\texttt{observer}\} \cup Views)$$

□

In [Eden, Hirshfeld & Yehudai 98] we use a similar technique to resolve a debate reported by Vlissides [97]. We show that the pattern *Multicast* proposed by the author is not, as his opponent suggested, a "specialization" of the *Observer* pattern [Gamma et. al 94].

6. SUMMARY AND FUTURE DIRECTIONS

We have described a well-founded visual formalism for the specification of object-oriented design and architecture. We demonstrated the use of LePUS in the specification of design patterns and frameworks and with reasoning thereon. Below we describe the theorem that must be proven in order to Establish the soundness of the *unification* relation, which can be effectively used by visual tools in support of software specifications with LePUS.

Eden and Hirshfeld [01] define the concept of *satisfaction* of a formula by a design model. Concisely, a LePUS formula j is satisfied by a program p if the design model π of p contains an n-tuple of ground entities $(t_1, \ldots t_n)$ such that their respective assignment to the terms in j is true in π. This concept can easily be extended to LePUS diagrams.

Given a diagram Δ, let us designate the set of *design models* (Section 2) that satisfy it as $[\![\Delta]\!]$. Thus, we may define the semantic relation

$$\Delta \vDash \Gamma$$

iff $[\![\Delta]\!] \subset [\![\Gamma]\!]$.

This definition allows us to formulate the following hypothesis:

Soundness of *Unification.*

Given two diagrams Δ, Γ, than

$$\Delta \vdash_{u} \Gamma$$

is true only if

$$\Delta \vDash \Gamma$$

is true.

APPENDIX

Nomenclature

$\mathbb{U}$	The domain of all *ground entities* (*atoms*)
$\mathbb{C}$, $\mathbb{F}$	The domains of *ground classes* and *ground methods*, respectively
$\mathbb{R}$	The domain of *ground relations*
$\mathbb{H}$	The domain of *class hierarchies*
$\mathbb{S}$	The set of *signatures*
$\mathcal{R}_1, \mathcal{R}_2$	Relation symbols
c^d	Constant of dimension d
V^d	Variable of dimension d
$\mathbf{P}(S)$	The power set of set S
$dim(t)$	The *dimension* of term t
L	An OOP language
D	A *denotation*
U	A *unification function*
Δ, Γ	LePUS diagrams

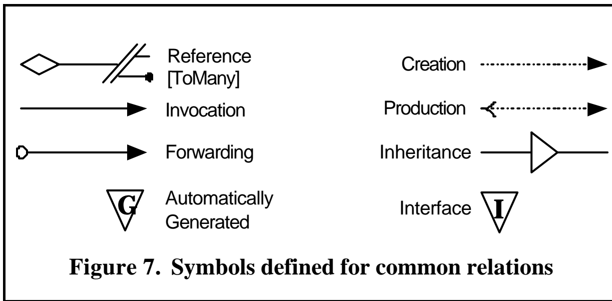

Figure 7. Symbols defined for common relations

Definition I: *Dimension*

(i) A ground class (method) is designated a *class (method) of dimension 0*.

(ii) A set of classes (methods) of dimension d is *a class (method) of dimension $d+1$*.

$\mathbf{P}^d(\mathbb{C})$ designates the domain of classes of dimension d.

Definition II: *Hierarchy*

A uniform set of classes $h \in \mathbf{P}(\mathbb{C})$ is a *hierarchy* if it contains a class r such that:

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

where $Inherit^{+}$ stands for the transitive closure of the binary relation $Inherit$.

Definition III: *Isomorphic*

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

- $m=n=0$ $\quad \mathcal{R}(\mathtt{V}^m, \mathtt{W}^n)$
- $m>0,\ n=0$ $\quad \forall V^{m-1} \in \mathtt{V}^m \bullet \mathcal{R}^{\leftrightarrow}(V^{m-1}, \mathtt{W}^n)$
- $m=0,\ n>0$ $\quad$ (Symmetrically)
- $m,n>0$ $\quad \forall V^{m-1} \in \mathtt{V}^m \ \exists! W^{n-1} \in \mathtt{W}^n \bullet \mathcal{R}^{\leftrightarrow}(V^{m-1}, W^{n-1})$ and $\forall W^{n-1} \in \mathtt{W}^n \ \exists! V^{m-1} \in \mathtt{V}^m \bullet \mathcal{R}^{\leftrightarrow}(V^{m-1}, W^{n-1})$

where (i.e., $\mathcal{R}$ is a bijective function in $\mathtt{V}^m, \mathtt{W}^n$).

Definition IV: *Total*

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

- $m=n=0$ $\quad \mathcal{R}(\mathtt{V}^m, \mathtt{W}^n)$
- $m>0,\ n=0$ $\quad \forall V^{m-1} \in \mathtt{V}^m \bullet \mathcal{R}^{\rightarrow}(V^{m-1}, \mathtt{W}^n)$
- $m=0,\ n>0$ $\quad$ (Symmetrically)
- $m,n>0$ $\quad \forall W^{n-1} \in \mathtt{W}^n \ \exists V^{m-1} \in \mathtt{V}^m \bullet \mathcal{R}^{\rightarrow}(V^{m-1}, W^{n-1})$

(i.e., $\mathcal{R}$ is a total function in $\mathtt{V}^m, \mathtt{W}^n$).

Definition V: *Clans and Tribes*

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

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

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

Definition VI: *Pre-Method*, *Signature*.

We assume that $SameSignature$ is an equivalence relation. Hence, it induces a partition on $\mathbb{F}$, and we may define the following:

- An *element* of the quotient set $\mathbb{F}/SameSignature$ is called a *pre-method*.
- A *representative* s for a pre-method is called a *signature*.
- The set of all the representatives is designated $\mathbb{S}$.

Definition VII: *Selection Operator*

Given a signature s and a class C^d, we define the expression $s \otimes C^d$ as the following set of methods, depending on the *dimension* d of C^d:

- $d=0$ $\quad \{ f \mid SignatureOf(f,s) \wedge DefinedIn(f, C^d) \}$

♦ $d>0$ { $s \otimes C^d\text{-}1$ } for all $C^{d-1} \in C^d$

Given a set of signatures S and a class C^d, we define $S \otimes C^d$ as the following set of functions:

$S \otimes C^d \equiv \{ s \otimes C^d \}$ for all $s \in S$

Definition VIII: ***Unification Function***

Given two diagrams Δ, Γ, and a function $\cup$ that maps terms in Δ to terms in Γ, we say that $\cup$ is a *unification function* if the following conditions hold for any terms t, t_1, t_2 in Δ:

(i) $\cup$ is an onto function (i.e., its range covers all the terms in Γ)

(ii) If $\cup(t)$ is a variable then $dim(t) \geq dim(\cup(t))$

(iii) If $\cup(t)$ is a constant then at least one of the following holds:
- $t = \cup(t)$
- $dim(t) \geq dim(\cup(t))$

(iv) If $\mathcal{P}(\mathcal{R}, \cup(t_1) \otimes \cup(t_2), t)$ then $\mathcal{P}(\mathcal{R}, t_1 \otimes t_2, t)$

(v) Omitted ([i]).

where $\mathcal{R}$ is a binary relation symbol and $\mathcal{P}$ is a predicate symbol.

REFERENCES

R. Allen, D. Garlan. "A Formal Basis For Architectural Connection." *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 3, July 1997, pp. 213-249.

J. Barwise, ed. (1977). *Handbook of Mathematical Logic*. Amsterdam: North-Holland Publishing Co.

G. Booch, I. Jacobson, J. Rumbaugh (1999). *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley.

A. H. Eden, Y. Hirshfeld, A. Yehudai (1998). "Multicast – Observer ≠ Typed Message." *C++ Report*, Vol. 10, No. 9, pp. 33-39, October 1998.

A. H. Eden (2000). "Precise Specification of Design Patterns and Tool Support in Their Application," PhD dissertation., Department of Computer Science, Tel Aviv University.

A. H. Eden (2001). "Formal Specification of Object-Oriented Design." *International Conference on Multidisciplinary Design in Engineering CSME-MDE 2001*, November 21-22, 2001, Montreal, Canada.

A. H. Eden, Y. Hirshfeld (2001). "Principles in Formal Specification of Object Oriented Design and Architecture." *CASCON 2001*, November 5-8, 2001, Toronto, Canada.

A. H. Eden, J. Jahnke (2002). "Coordinating Software Evolution Via Two-Tier Programming". *Fifth International IEEE Conference on Coordination Models and Languages – Coordination 2002*. York, UK, 8-11 April 2002.

D. G. Firesmith, I. Graham, B. Henderson-Sellers (1999). *Open Modeling Language (OML) Reference Manual*. Cambridge: Cambridge University Press.

E. Gamma, R. Helm, R. Johnson, J. Vlissides (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, MA: Addison-Wesley.

D. Garlan, M. Shaw (1993). "An Introduction to Software Architecture." *Advances in Software Engineering and Knowledge Engineering*, Vol. 2, pp. 1-39.

A. Lauder, S. Kent (1998). "Precise Visual Specification of Design Patterns." In: E. Jul (ed.): *Proceedings of the 12th European Conference on Object Oriented Programming, Brussels, Belgium*. Lecture Notes in Computer Science 1445, pp. 114-134. Berlin: Springer-Verlag.

D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, W. Mann (1995). "Specification and Analysis of System Architecture Using Rapide." *IEEE Transactions on Software Engineering*, Special Issue on Software Architecture, Vol. 21, No. 4, pp. 336-355, April 1995.

V. Matena, M. Hapner (1999). *Enterprise JavaBeans™ Specification, v1.1*. Palo Alto, CA: Sun Microsystems.

T. Mikkonen (1998). "Formalizing Design Patterns." *Proceedings of the International Conference on Software Engineering*, April 19 - 25, 1998, Kyoto, Japan.

M. Pawlan (1998). "Enterprise JavaBeans™: Working with Entity and Session Beans." Technical report, Sun Microsystems.

D. E. Perry, A. L. Wolf (1992). "Foundation for the Study of Software Architecture." *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No. 4, Oct. 1992, pp. 40-52.

J. Rumbaugh, et. al (1991). *Object Oriented Modeling and Design*. Prentice Hall.

J. M. Vlissides (1997). "Multicast." *C++ Report*, Vol. 9, No. 8, September 1997.

K. Walrath, M. Campione (1999). *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley.

---

[i] The conditions described here are only listed for illustrative purposes. A more complete description of a function of real use remains the subject of future research.