VISUALIZATION OF OBJECT ORIENTED ARCHITECTURE

Amnon H. Eden

Dept. of Computer Science, Concordia University, Montreal, Canada

eden@acm.org

http://www.cs.concordia.ca/~faculty/eden

Abstract

We present architectural schemata, *visual "roadmaps" that provide a lucid picture of the design of object oriented programs. Construed in a formal language, architectural schemata can contribute significantly to the comprehensibility of large systems and provide a rigorous framework to their analysis.*

We present LePUS, a formal language for the specification of object oriented architectures. We demonstrate how LePUS schemata can document the design, architecture, and usage of prevalent architectures at the appropriate abstraction level. Finally, we show how to verify conformance to architectural specifications in LePUS.

1. Introduction

The development of large-scale software manifests a constellation of difficulties, which have been discussed extensively in the software engineering literature. Particularly thorny is the problem of *software invisibility*:

> "*In spite of progress in restricting and simplifying the structures of software, they remain inherently unvisualizable, and thus do not permit to use some of the most powerful conceptual tools.*" [1]

In agreement with Brooks, we maintain that the lack of visual representations or "roadmaps" to the architecture of software systems not only impedes the design process but also severely hinders communicating it after completion.

Certain 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 a primary goal of architectural schemata. Hence, allowing the architectural specifications to be made in a visual notation is most desirable.

Generic Vs. Concrete Specifications

It is useful to distinguish *generic* from *concrete* architectural specifications:

- *Concrete* specifications give a clear picture of the gross organization of a specific software system. Hence, concrete specifications consist of constant symbols, which represent extant, specific architectural elements.
- *Generic* specifications are similar to *architectural style*: "a family of such systems in terms of a pattern of structural organization." [2] Generic specifications abstract the essential aspects of many different concrete specifications. Thus, generic specifications must incorporate variables, which may be matched against different elements of different programs.

Hybrid Specifications

Some specifications do not fall strictly under one of the two but incorporate a combination of generic and concrete specifications. In particular, these include the specifications of O-O application frameworks and class libraries.

A *framework* [3] is a reusable, "semi-complete" application that can be specialized to produce custom applications. While some of the application logic is provided with the framework classes, its functionality is meant to be extended by its users. In conclusion, the specification of O-O frameworks requires a combination of generic and concrete clauses.

Similarly, the documentation of class libraries need illustrate their collaboration with elements in the context of their use. Hence, it is expected to reflect interaction between specific (library) and projected (user-defined) elements.

Despite the intended focus on reusability and extendibility, application programmers are required to have intimate knowledge of the internal structure of the framework or library they use. Hence, their accurate and comprehensible documentation is crucial.

Desiderata

In conclusion, an ideal specification language for O-O architectures should –

1. be well-defined and provided with formal semantics;
2. provide means for analysis and deduction;
3. allow specifications at the appropriate level of abstraction;
4. allow for generic, concrete, and hybrid specification by providing both constant and variable symbols;
5. be visual.

2. LePUS

LePUS (*LanguagE for Patterns Uniform Specification*) was defined [4] as a formal language for the specification of O-O patterns and architectures. It is based on higher order logic, and is defined both as a visual and as symbolic language. Below, we explain the visual specifications and the formal model behind them. We use the terms *diagram* and *schema* interchangeably for this purpose.

A complete definition of LePUS is provided in [5, 6]. Additional information can be obtained online from [7].

2.1 Models

In comparison with the host of details which the source code of programs normally contain, architectural specifications take only to a fraction in "volume". Aiming for *architectural abstraction*, this complicated picture of the source code is simplified by "distilling" it. Hence, we define the concept of *architectural model*, a logic *structure* which consists entirely of –

1. *Ground entities*, each of which is of type *class* or *method* (also *function*), and
2. *Relations* defined amongst the entities.

Properties of classes and methods, as well as the correlations amongst them, are manifested through a small set of *relations*. A *relation* is simply a set of tuples pf ground entities that satisfy a particular property. For example, the set of entities c such that "c is a class", or the set of entities (d,b) such that "class d inherits from class b."

For example, the program depicted in Table 1 is mapped to the model in Diagram 1. This model includes, among others, the relations –

Inherit(`BorderDecorator`, `Decorator`)

DefinedIn(`Decorator::Draw`, `Decorator`)

Invoke(`BorderDecorator::Draw`, `Decorator::Draw`)

Uniform Sets

A *uniform set* is a set of entities of the same type and dimension (see below). 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.

A ground class is also called a class of dimension 0, and a set of classes of dimension d-1 is called a class of dimension d. For example, Diagram 2 depicts the set of methods called `ReadAndWrite` as a method of dimension 1. Following is an explanation for the way sets and their relations are specified in LePUS.

2.2 Fundamentals

LePUS formulae consist of the following symbols:

- *Terms (Constants* and *Variables)*
- *Relations*
- *Predicates*

We use three examples in the descriptions that follow: *Microsoft Foundation Classes* (Diagram 2); the *MODEL-VIEW-CONTROLLER* pattern in Java™ *Swing* library (Diagram 3); and a more elaborate example of the Enterprise JavaBeans™ (Diagram 5).

Terms

LePUS terms include variables and constants. Variables range over specific domains, such as $\mathbb{F}$, the domain of methods, and $\mathbb{C}$, the domain of classes. Thus, for instance, $model$ (Diagram 3) is a class variable, namely, it ranges over the domain of 0-dimensional classes, while $Views$ ranges over 1-dimensional classes.

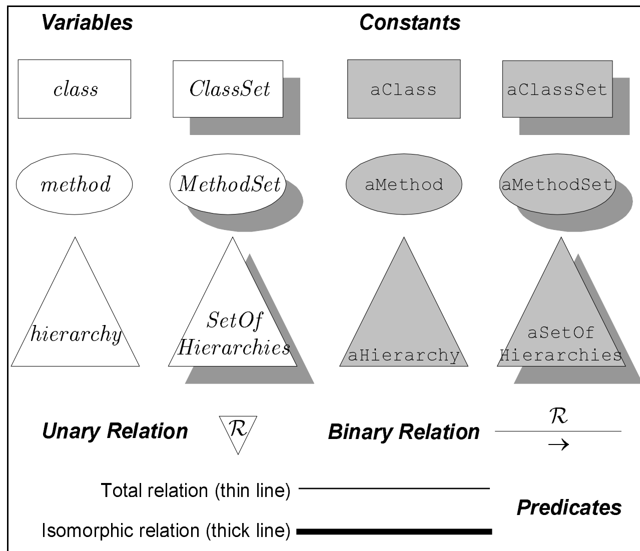

***Figure 1**. Legend for generic LePUS symbols*

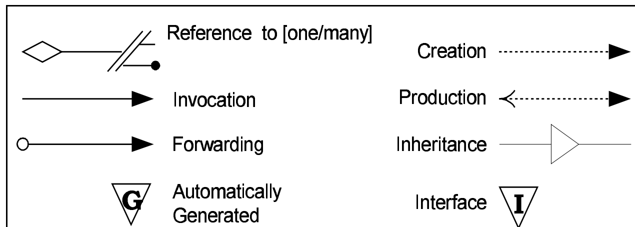

***Figure 2**. Legend for commonly used relations*

***Table 1**. Java implementation of the Decorator pattern*

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

class BorderDecorator
   extends Decorator
{
   void Draw() {
      Decorator::Draw(); //...
   }
   int BorderWidth;
}
```

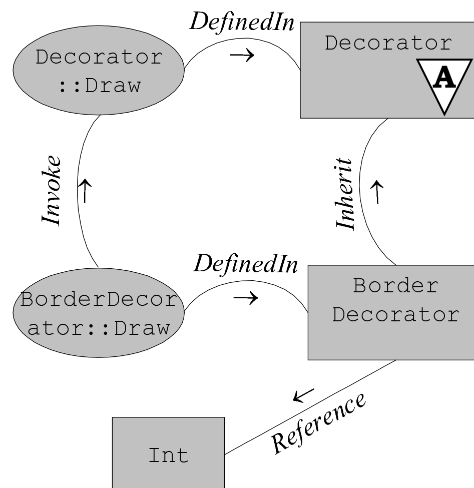

Diagram 1. Model of the program in *Table 1*

In contrast, *constant* symbols represent specific entities, such as the set of methods `ReadAndWrite` (Diagram 2). Constant symbols appear in `fixed size` typeface printed over solid icons, while variable names appear in *italics* and whose shapes have a white fill (Figure 1).

Constants and variable allow us to distinguish between extant and projected entities. Thus, for instance, in specifying how to use the Java™ *Swing* library (Diagram 3), entities that are part of *Swing* are indicated as constants while the parts that the user should write appear as variables.

Relations

LePUS diagrams contain unary relations (inverted triangles) and binary relations, depicted as edges connecting the two entities involved. Thus, for instance, Diagram 3 indicates that the unary relation *Interface* applies to class `observer`. 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 1.

2.3 Predicates

In LePUS, predicates are used to express properties of relations and sets.

Isomorphic relations

Consider the following description [8] of the "call forwarding" relation between the methods defined in the implementation of the "Home Interface" and the ones defined in the "bean" class in EJB:

> *... 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.*

We conclude that for every method bh in the set $BeanHomeImp::Create$ there is exactly one method b in the set of methods $Bean::Create$ such that $Forward(bh,b)$ is true. In other words, we say that the relation $Forward$ is an isomorphism between the sets $BeanHomeImp::Create$ and $Bean::Create$. Stated in a symbolic form, we use the predicate $Isomorphic$ to write

$Isomorphic(Forward,$
$BeanHomeImp::Create,$
$Bean::Create)$

Visually, isomorphic relations are indicated by thick edges, such as the edge connecting $Bean::Create$ and $BeanHomeImp::Create$ in Diagram 5.

Total relations

Swing documentation [9] assert that every "mutator" method defined in class "model" must invoke the method `setChanged`, which is defined in class `Observable`. In mathematical terms, we say that the relation $Invoke$ is a *total function* (rather than an isomorphism) between the sets $Mutators$ and `setChanged`[i]. Symbolically, the predicate $Total$ has the following form:

$Total(Invoke, Mutators,$ `setChanged`$)$

Visually, total relations are represented as thin edges. In Diagram 3, for instance, the $Invoke$ arrow connecting $Mutators$ to `setChanged` consists of a thin shaft.

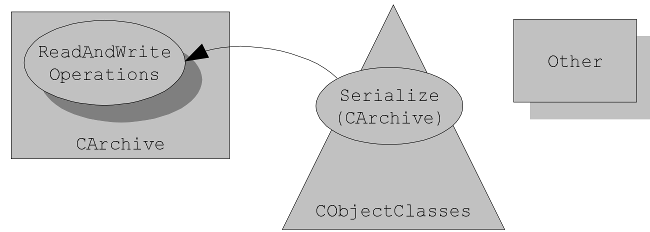

***Diagram 2**. Microsoft's MFC*

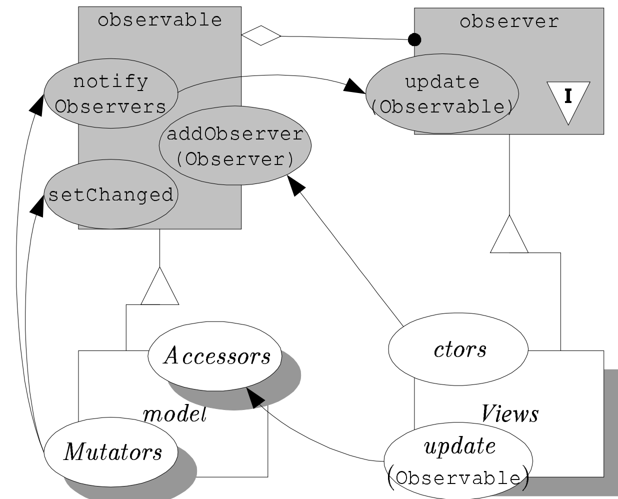

Diagram 3. Swing's *MVC "usage pattern"*

2.4 Compositions

Finally, we observe commonly recurring compounds that deserve special treatment in LePUS.

Hierarchies

Consider the inheritance hierarchy depicted in Diagram 3, consisting of the set $Views$ and the base `observer`. Similar constructions exist in every OOP program. Observing this, we define as a *hierarchy* any set of classes that contains a "root" such that all other classes inherit (possibly indirectly) therefrom. Hierarchies are depicted as triangles, such as `CObjectClasses` in Diagram 2.

Clans and Tribes

Consider the set of methods `Serialize` that are defined in many MFC classes. To enable dynamic binding, all these methods conform to the method defined in `CObject`. Also, observe that each method in `Serialize` is defined in a different class; in our terminology, $DefinedIn$ is an isomorphic relation between `Serialize` and `CObjectClasses`. A set of methods with these two properties (i.e., signatures conformance and isomorphic $DefinedIn$ with a class) is called a *clan*. Figure 3 gives a legend for clan symbols.

The dimension of a clan is determined not by the ellipse but by the shape that it superimposes. Thus, a clan in a ground class is simply a ground function defined in that class. For instance, $Update$ (Diagram 3) is a *1*-dimensional method, as it superimposes a *1*-dimensional class ($Views$).

A set of clans that share the same set of classes is called a *tribe*. For instance, `ReadAndWriteOperations` (Diagram 2) is a tribe in class `Carchive` (meaning, in this case, simply a set of methods defined therein). The di-

[i] A ground entity such as `setChanged` can also be treated as a singleton set.

mension of a tribe is defined to be *1* plus the dimension of the superimposed shape. Thus, for instance, the dimension of *Methods* (Diagram 5) is *2* (namely, it is a set of sets of methods).

3. Conformance to a Pattern

We demonstrate the reasoning power of LePUS for the verification of compare *Swing*'s MVC (Diagram 3) with the *OBSERVER* design pattern [10], depicted in Diagram 4. This example demonstrates the possibility of *verification* of architectural properties with LePUS schemata.

The similarity between the two schemas is apparent as both manifest a notification mechanism based on the same principles. We say that Diagram 3 *conforms to* Diagram 4, where the *conformance* relation between diagrams Δ and Γ, written

$$\Gamma \models \Delta$$

implies that every program that satisfies Δ also satisfies Γ. A formal definition of *conformance* appears in [6].

Conformance can be proved by unifying elements from Diagram 3 with the variables in Diagram 4. As Table 2 clearly shows, every model that satisfies *Swing's* MVC also satisfies the *OBSERVER* pattern.

4. Architectural Schema of EJB

Enterprise JavaBeans™ (EJB) [8] is a "programming model", supporting the development of distributed, server-side software. EJB specifications includes numerous protocols of communication, conventions, and interface specifications. In particular, "beans", the classes that application developers need develop, must conform to certain requirements such that the software on the EJB server side can host and manage them. Elements of this programming model are illustrated in Diagram 5, which contains entities of three types:

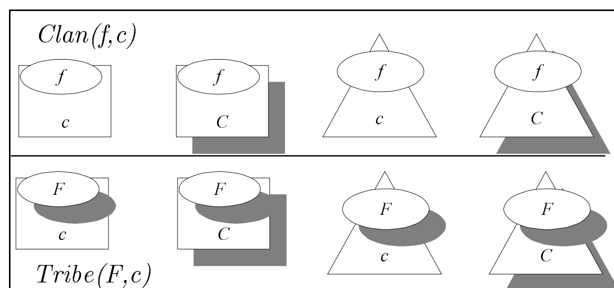

Figure 3. *Legend for* clans *and* tribes

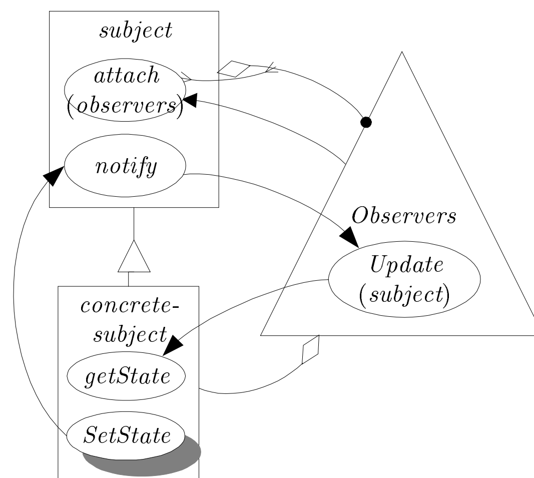

Diagram 4. *The* OBSERVER *design pattern*

Table 2. *Unifications from MVC (Diagram 3) entities to* OBSERVER *variables (Diagram 4)*

MVC	*OBSERVER*
`observer` & *Views*	*Observers*
`update` & *update*	*Update*
`observable`	*Subject*
`addObserver`	*Attach*
`notifyObservers`	*Notify*
`model`	*ConcreteSubject*
Mutators	*GetState*
Accessors	*SetState*

Table 3. *Verbal specifications of elements of the EJB architecture in Diagram 5*

(1) "Every bean obtains an `EJBContext` object, which is a reference directly to the container."

(2) "The stub implements the remote interface so it looks like a business object."

(3) "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."

(4) "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."

(5) "A bean's home interface may declare zero or more `create()` methods, each of which must have corresponding `ejbCreate()` and `ejbPostCreate()` methods in the bean class."

(6) "… 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."

- Constants designate classes whose implementation is provided with the Java EJB package, such as `EJBContext` and `java.rmi.remote`;
- Variables that are not marked **G**, such as class *Bean*, designate entities that the application developer should code;
- Variables marked **G**, such as class *BeanHomeImp*, designate entities that are automatically generated by the EJB environment from the users' implementations.

Selected extracts from the corresponding verbal specifications appear in Table 3, indicated by their respective number in Diagram 5.

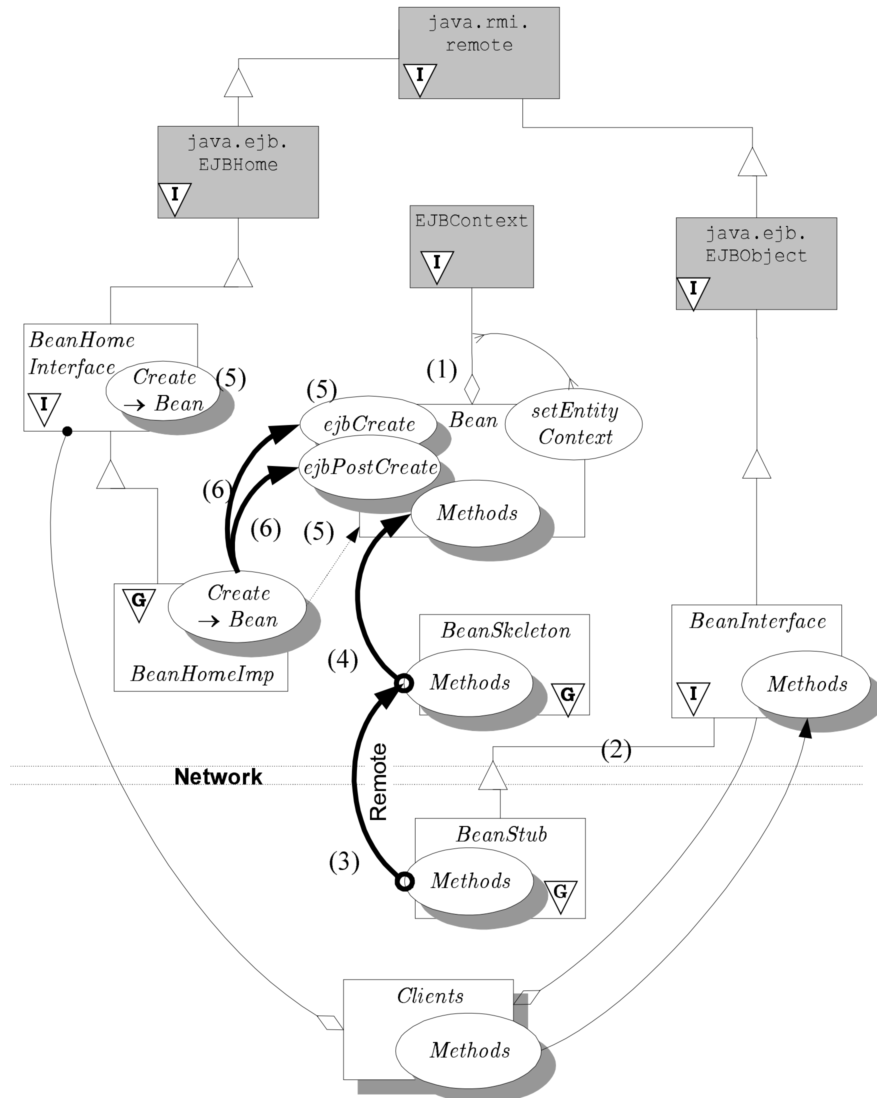

Diagram 5. *The architecture specified by the Enterprise JavaBeans™ "programming model". Numbers in this diagram indicate the respective quote from Table 3.*

References

[1] F. P. Brooks (1987). "No Silver Bullet: Essence and Accidents in Software Engineering". *Computer*, Vol. 20(4), Apr. 1987.

[2] D. Garlan, M. Shaw (1993). "An Introduction to Software Architecture." In V. Ambriola and G. Tortora (ed.), *Advances in Software Engineering and Knowledge Engineering*, Vol. 2, pp. 1-39. New Jersey: World Scientific Publishing Company.

[3] M. Fayad, D. C. Schmidt. "Object-Oriented Application Frameworks." *Communications of the ACM*, Vol. 40, No. 10, October 1997, pp. 32-38.

[4] A. H. Eden (2000). "Precise Application and Automatic Application of Design Patterns", Ph.D. Dissertation. Department of Computer Science, Tel Aviv University.

[5] A. H. Eden, Y. Hirshfeld, A. Yehudai (1999). "Towards a Mathematical Foundation for Design Patterns." *Technical Report 1999-004*: Department of Information Technology, Uppsala University.

[6] P. Grogono, A. H. Eden. "Concise and Formal Descriptions of Architectures and Patterns." Submitted: *The Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2001, Amsterdam, The Netherlands.

[7] `http://www.cs.concordia.ca/~faculty/eden/lepus/`

[8] V. Matena, B. Stearns (2000). *Applying Enterprise JavaBeans(tm): Component-Based Development for the J2EE(tm) Platform*. Reading, MA: Addison Wesley.

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

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