From Java classes to Java interfaces through Galois lattices *

Marianne Huchard, Hervé Leblanc
LIRMM,
161 rue Ada, 34392 Montpellier Cedex 5, France,
email: {*huchard,leblanc*}*@lirmm.fr*

July 16, 1999

Abstract

We develop here an application of the Galois lattice theory in the object-oriented systems framework. These systems are centered on the notion of *classes*, which gather data and procedures. The Java language also provides the concept of *interface*, which is like a shop window of classes. The classification of interfaces may be any partial order, while the classification of classes is restricted to a tree-like structure. We take advantage of these features to propose a technique to build an ordered set of interfaces giving a consistent view of a class set. This algorithm is based on the construction of a special sub-order of the Galois lattice associated with the relation mapping the classes to their properties.

1 Introduction

Object systems have a great succes in several computer science domains as knowledge representation, databases and software engineering. This succes is due to the proximity between the computer representation and the real world, as well as to the facility to develop and maintain such systems. The particularity of object systems is that they are built under the *class* notion. A class is an aggregation of data and procedures (or methods) acting on these data. One of the most important concept is *inheritance*, which organizes classes into a *hierarchy*, that is a partial order corresponding more or less to a real world classification. The problematic point is that such inheritance hierarchies are not so easy to build, and the object community is very interested by all methodologies and tools that could help object designers and programmers in this task. Previous works [GM93, DDHL96] have used the Galois lattice structure giving some keys to address the problem of building well factorized inheritance hierarchies. Unfortunately, some object-oriented languages, like Java, restrict the inheritance hierarchies to tree-like structures. As a consequence, they have some organization problems which can not be solved by the Galois lattice techniques without adaptations not very satisfactory.

Another main concept of object systems is the notion of *encapsulation*. Encapsulation is a necessary condition for an easy maintenance and evolution of programs [Mey88]. According this principle, classes have to be considered with two separate points of view:

- a *specification* point of view, describing the operations or services available on objects of the class; these services are said *public*, and constitute the *interface* of a class.
- an *implementation* point of view, describing how services are implemented; this part is usually said *private*, and is hidden by a protection mechanism.

*This work is supported by the CNET, project 971B.

In [Coo92], a sort of interface hierarchy for Smalltalk classes is built. But the proposal suffers from several problems: an algorithm is given which is not characterized, and the result is not really used in the language itself, only as a form of documentation.

We propose here to carry on this idea and to go deeply. Our work is based on the Java programming language which is becoming one of the most popular object-oriented programming languages. Java has a special policy concerning classes and encapsulation. Two kinds of objects are provided: *classes*, which can contain specification and implementation features, and *interfaces*, which only contain specification features as methods headers, also named *signatures*, and data common to all instances of a class, also named *static data*[1]. Java interfaces are a kind of *shop window* for classes and give directions for use. Java classes can only be organized into tree-like structures (single inheritance hierarchies), while Java interfaces may be organized into partial orders (multiple inheritance hierarchies). A class may *implement* several interfaces, while an interface may be implemented by several classes.

Netherveless, interfaces are not well exploited by programmers, essentially for cost development reasons: writing interfaces, as writing a program documentation is often considered as a waste of time. Here we address the problem of extracting automatically an interface hierarchy from a set of Java classes. Due to the single inheritance constraint on classes, Java class hierarchies are often not well organized, while Java interface hierarchies may be. Galois lattices give us an elegant solution to the problem.

2 Some Java classes and interfaces

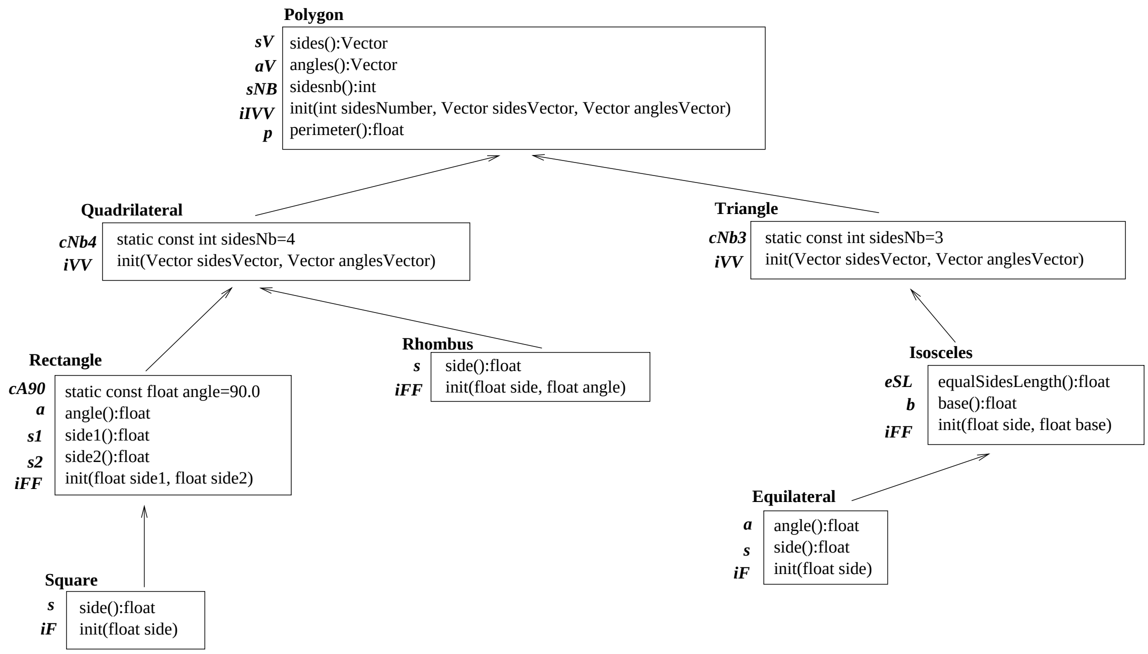

Figure 1: The public part of a polygon Java class hierarchy.

A convex polygon hierarchy is used in order to illustrate our proposal. This hierarchy is adapted from the Yafool system [Duc88]. As Java imposes, the hierarchy is a single inheritance hierarchy, limited here to classes *Polygon*, *Quadrilateral* (*Rectangle*, *Square*, *Rhombus*) and *Triangle* (*Equilateral*, *Isosceles*). Some elements of the public part of this class hierarchy are given in the figure 1. Two kinds of features are used here: methods and

[1]More precisely, interfaces only contain constant static data.

	iIVV	SNB	p	sV	aV	iVV	iFF	CNb4	s	CNb3	a	S2	S1	eSL	cA90	b	iF
Polygon	x	x	x	x	x												
Triangle	x	x	x	x	x	x				x							
Quadrilatéral	x	x	x	x	x	x		x									
Rhombus	x	x	x	x	x	x	x	x	x								
Isoceles	x	x	x	x	x	x	x			x				x		x	
Rectangle	x	x	x	x	x	x	x	x			x	x	x		x		
Equilateral	x	x	x	x	x	x	x		x	x	x			x		x	x
Square	x	x	x	x	x	x	x	x	x		x	x	x		x		x

Table 1: *The relation owns-property for polygon classes.*

constant static data. Methods are represented by their signature. A signature has two parts, the first is a list of the method argument types followed by a name of argument, the second is the method return type if there is one. For example, the class *Polygon* has a method called *perimeter*, without argument, and which returns a float number representing the polygon perimeter. Codes of methods and internal data are omitted. The only data we keep are *constant static data*, that is, data common to all instances of a class. For example, *sidesNb* for *Quadrilateral* which is equal to 4 for all quadrilateral instances.

Due to the single inheritance constraint, this hierarchy has several defaults. Firstly, *Square* can not be considered as a subclass of both *Rectangle* and *Rhombus* classes as it would be the case in a natural classification. Secondly, some properties are duplicated, because they can not be factorized correctly (for example *side*() which is repeated in classes *Rhombus*, *Square*, and *Equilateral*). The fact that properties are not factorized is rather problematic, because when an update has to be done to such a property, the programmer has to carry out the update on all occurrences of the property, which is often a source of errors. This lack of factorization also entails the readibility of the hierarchy.

In such inheritance hierarchies, a class inherits all the properties of the classes it specializes, also called super-classes. Except some special cases not developped here, a binary relation mapping classes to their public properties can be built from the hierarchy: to a class are associated all the properties inherited or declared by the class. Such a relation is called *owns-property* and for polygons is shown in table 1 (property names have been simplified).

3 Galois sub-hierarchy, definitions and construction schema

Galois (or concept-) lattices and a special sub-structure, namely the *Galois sub-hierarchy*, are used here because they are a good medium to factorize common properties of an object set, giving at the same time a classification of them [DP90, Wil82]. Futhermore this structure is the minimal one (in space complexity) ensuring a maximal factorization. We recall some basic definitions.

Let E (classes for example) and F (resp. properties) be two finite sets and $\mathcal{R} \subseteq E \times F$. Two mappings, f and g form the basis of the construction. f maps a set of classes to the properties they share, when g maps a set of properties to classes which share all of them. The pair (f, g) yields a Galois connection, and $h_E = gof$ et $h_F = fog$ are closure operators. A set $X \subseteq E$ (resp. $Y \subseteq F$) such that $h_E(X) = X$ (resp. $h_F(Y) = Y$) is called a *closed set* with respect to h_E (resp. h_F).

Definition 3.1 (Galois lattice) *Given a binary relation $\mathcal{R}$, and the closures h_E and h_F associated with $\mathcal{R}$ as shown previously, the Galois lattice is the set $\mathcal{L}$ of pairs of closed sets (X,Y) where X is a closed set w.r.t. h_E and Y is a closed set w.r.t. h_F such that $Y = f(X)$*

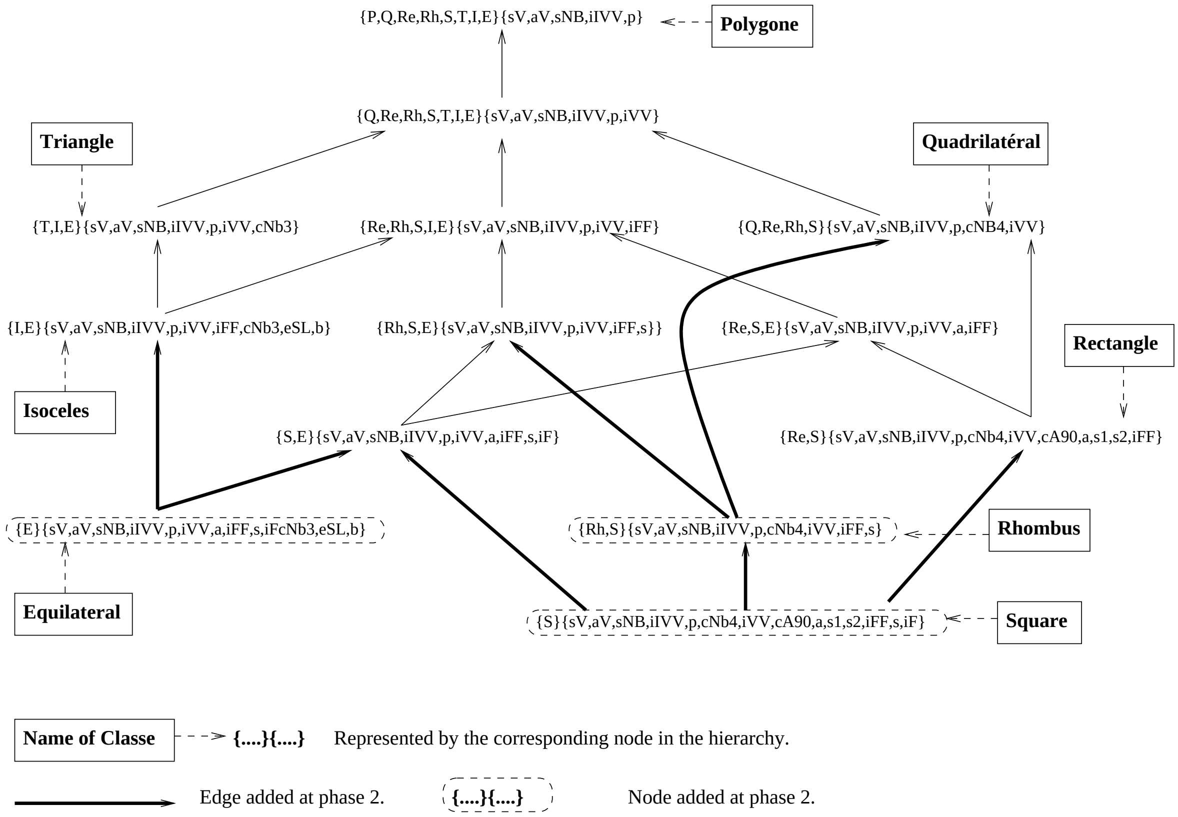

Figure 2: The Galois sub-hierarchy.

(or $X = g(Y)$), provided with the order $<_{\mathcal{L}}$ defined by $(X_1, Y_1) <_{\mathcal{L}} (X_2, Y_2)$ if $X_1 \subset X_2$ (or $Y_2 \subset Y_1$).

The Galois sub-hierarchy was first defined in the object framework by [GM93].

Definition 3.2 (Galois sub-hierarchy) *Let $\mathcal{L}$ be a Galois lattice, the Galois sub-hierarchy is the order obtained removing from $\mathcal{L}$ the pairs (X, Y) satisfying the two following conditions:*
- *$\forall x \in X$, there is a pair $(X', Y') \in \mathcal{L}$ with $(X', Y') <_{\mathcal{L}} (X, Y)$ and $x \in X'$*
- *$\forall y \in Y$, there is a pair $(X', Y') \in \mathcal{L}$ with $(X, Y) <_{\mathcal{L}} (X', Y')$ and $y \in Y'$*

In terms of representation, the removed vertices correspond to a possible factorization, but are often useless because they do not represent any element of E (there is no element $e \in E$ such that $f(\{e\}) = Y$) and they do not represent any element of F (there is no element $e \in F$ such that $g(\{e\}) = X$). In terms of complexity, each retained vertex being representative of at least an element of E or an element of F, the structure has a vertex number less or equal to $|E| + |F|$, which is an important reduction compared with the exponential vertex number of the whole lattice.

Figure 2 shows the Galois sub-hierarchy for polygons. A vertex as disappeared from the lattice, corresponding to the pair $(\{Re, Rh, S\}, \{sV, aV, sNB, iIVV, p, cNB4, iVV, iFF, \})$.

Up to now, we have not found a complete construction algorithm in the literature, only algorithms, as [MGG90], that build a major part of the sub-hierarchy, apparently due to application domain constraints. Extending these approaches, we propose a construction schema based on the following property [DHH99].

Property 3.1 *The vertex set of $\mathcal{L}$ is the union of the two following sets (not necessary disjoint):*
$O_P = \{(g(f(\{e\})), f(\{e\}))$ s.t. $e \in E\}$ and $O_C = \{(g(\{a\}), f(g(\{a\})))$ s.t. $a \in F\}$

The resulting vertices, ordered by inclusion, are shown in figure 2. We have developped an algorithm that works in two phases. The first phase is concerned by the computation and ordering of all closed set pairs of O_P. These pairs are useful to factorize properties. At the same time, the classes which are represented by such pairs are marked. In the second phase, only closed set pairs of $O_C \setminus O_P$ are obtained and classified in the hierarchy.

We give now a rough time complexity of this algorithm. To reduce complexity, rows of the relation table are ordered with respect to an increasing number of properties, while columns are ordered with respect to a decreasing number of classes (as in table 1). In that way, upper nodes are discovered first. Let M be the number of pairs of the relation, obtaining the ordered table is in $max(\mathcal{O}(M), \mathcal{O}(|E|.Log(|E|)), \mathcal{O}(|F|.log(|F|)))$. Sets being represented by bit vectors, we assume that the set operations are in constant time. In first phase, there are at most $|F|$ vertices which are created. These vertices are compared together to determine the transitive reduction of the inclusion order between their left part (the associated class set). This step is in $\mathcal{O}(|F|^2)$. In the second phase, at most $\mathcal{O}(|E|)$ vertices are created and inserted. For that, they have to be compared (using the right part, i.e. the associated property set) together, and with the vertices of previous phase. The step is in $\mathcal{O}(|E|.(|E|+|F|))$. For more details about the algorithm and its complexity, see [HL99].

4 Getting the interface hierarchy for polygons

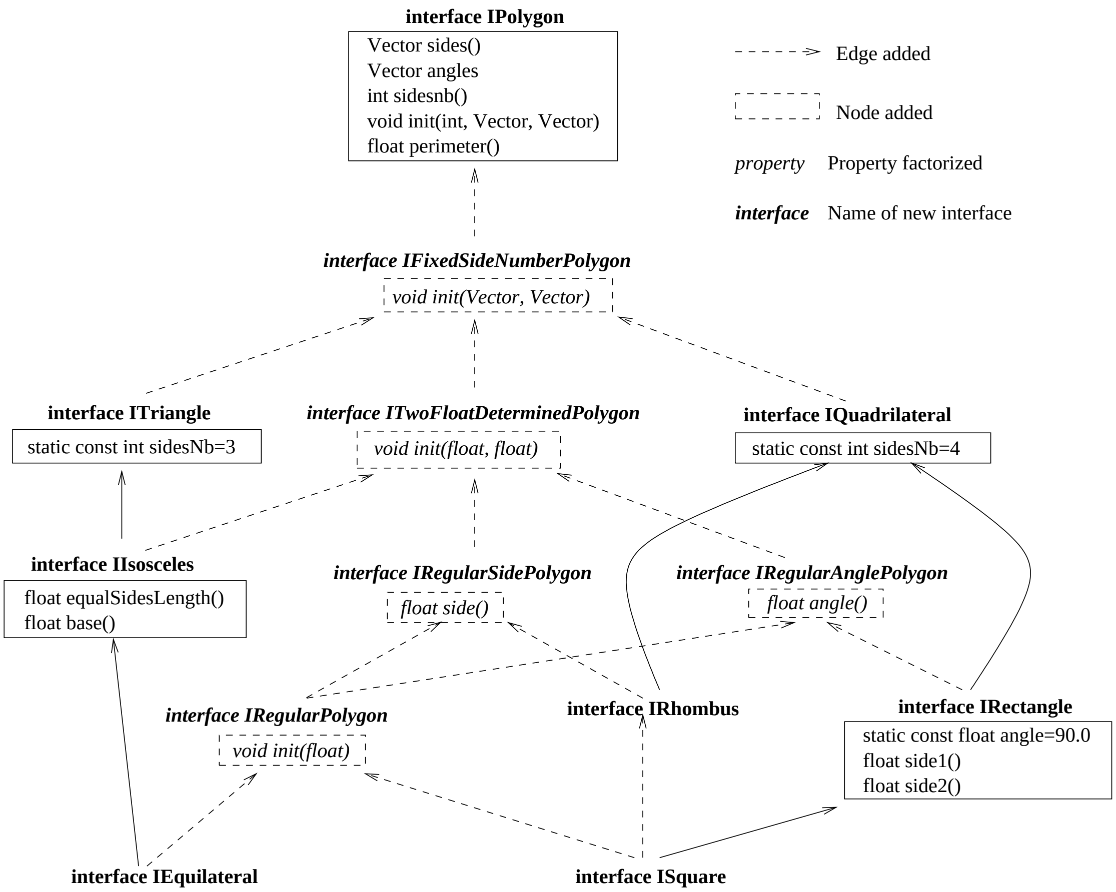

Figure 3: The interface hierarchy.

The interface hierarchy is deduced from the Galois sub-hierarchy by a simplification of the label vertices. A class is removed from the label if it appears in a vertex below, and symmetrically, a property is removed if it appears in a vertex above. The resulting interface hierarchy is shown figure 3. The property names are in rectangle boxes. The class names have disappeared, and replaced by interface names.

This interface hierarchy is a view on classes which is more conform to the usual mathematical classification. Thanks to the Galois sub-hierarchy construction, the interface hierarchy does not contain property redundancies and it introduces interesting intuitive concepts about polygons that a designer can recognize and name, like the concept $IRegularPolygon$. This example shows also the limit of the method: some new concepts may seem less interesting like $ITwoFloatDeterminedPolygon$, the interface that represents polygons which are determined by only two float numbers (a side and an angle, or two sides, or a side and a diagonal, etc.). This interface hierarchy is connected to the class hierarchy: a class *implements* the interface which corresponds to the lower vertex that contain the class, (see [HL99]).

It is more complex, comparatively to the class hierarchy, but this disadvantage is counterbalanced by a better organization and by the fact that the subsequent programming effort is reduced. Firstly, due to the factorization, a property modification will be confined to the interface declaring the property and the classes implementing this interface. Secondly, the development of an operation that applies on all regular polygons is now simplified because $IRegularPolygon$ can be used as a type.

5 Conclusion

We have outlined a method to deduce an interface hierarchy for a Java class hierarchy. Netherveless, the algorithm has a more general scope, as it gives a global method to build a Galois sub-hierarchy. The method is implemented on a Java 1.2 platform and is currently tested on two kinds of class hierarchies. Class hierarchies which are components of the Java environment are studied to compare the interfaces actually proposed by the environment and the interface hierarchy built by our method. In the framework of a CNET[2] project, this method is intended to help developpers of several parts of applications dedicated to the network management in sharing class hierarchies built separately, giving them a good way to exchange views of their classes, and above all, these views may be used as components of programs.

References

[Coo92] William R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'92.*, 10(1):1–15, 1992.

[DDHL96] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On automatic class insertion with overloading. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'96.*, 31(10):251–267, 1996.

[DHH99] H. Dicky, M. Huchard, and H.Leblanc. Etude d'algorithmes en relation avec la construction globale de sous-hiérarchie de galois. Technical Report 99059, LIRMM, Montpellier, 1999.

[DP90] B.A. Davey and H.A. Priestley, editors. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[Duc88] R. Ducournau. *Y3, version 3.* Sema Group, 1988.

[GM93] R. Godin and H. Mili. Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'93*, 28(10):394–410, 1993.

[HL99] M. Huchard and H. Leblanc. From Java classes to Java interfaces through Galois lattices. Technical Report 99088, LIRMM, Montpellier, 1999.

[Mey88] B. Meyer. *Object-oriented Software Construction.* Englewood Cliffs NJ: Prentice Hall, 1988.

[MGG90] Guy Mineau, Jan Gecsei, and Robert Godin. Structuring Knowledge Bases Using Automatic Learning. *Proceedings of the sixth International Conference on Data Engineering*, pages 274–280, 1990.

[Wil82] R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. *Ordered Sets, in I. Rivals (Eds)*, 23, 1982.

[2] Centre National d'Etudes des Télécommunications