Computing Interfaces in Java

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

Abstract

We investigate the separation between types and classes by putting to use a special feature of Java regarding classes, interfaces and inheritance. We propose an original method which, from a single inheritance class hierarchy*, extracts a multiple inheritance* interface hierarchy*, which contains all the types of the original hierarchy, each class being linked to the interface representing its type. In the resulting structure, interfaces are well organized, and follow a natural multiple specialization, which would not have been possible using only the single inheritance which comes with Java. Our method is based on the use of a* Galois lattice*, which, as previous works have shown, is a reference for the elaboration of hierarchies. We introduce and justify the need for a new algorithm that efficiently builds an essential part of the Galois lattice.*

1. Introduction

In Object-oriented approaches, there are two main relations regarding hierarchy structures: *specialization* that comes from the application domain and *sub-typing* that corresponds roughly to the inclusion between the sets of public services offered by a pair of types, when (as in Java) the only redefinitions admitted for methods are invariant (same parameter types, same return type). Mixing these very different sorts of relations in the same class hierarchy might cause problems.

We believe that a good approach presents several views on a class hierarchy. Thus, when using Java, the interfaces, which are types of the language, can be used to present a hierarchy from the sub-typing point of view. The syntactic nature of the notion of type justifies constructing them in an automatic fashion.

In the Java language, two kinds of “concepts” are provided: *classes* that contain specification and implementation features, and *interfaces* which contain only specification features as method headers (also called *signatures*), and constant *static data*. A class may *implement* several interfaces, and an interface may be implemented by several classes. When a class implements an interface, the class contains data and code to implement the services promised by the interface. Java classes are allowed only tree-like organization (single inheritance hierarchies), while Java interfaces may be organized into partial orders (multiple inheritance hierarchies). The main reason for this is that single class inheritance precludes conflicts between inherited properties. For example, a class cannot inherit two different bodies for a given method. Regarding conversely interfaces, as only signatures are inherited, and as static data can be referred to using the interface name, there is no possibility for conflict.

Our aim in this work is to use as systematically as possible the wealth of possibilities offered by this special Java feature. More specifically we shall address the problem of extracting an interface hierarchy automatically from a set of Java classes.

The paper is organized as follows. In Section 2 we first give an example to illustrate the extraction of an interface hierarchy from a class hierarchy. Section 3 explains how this extraction is done, by building the Galois sub-hierarchy associated with the relation mapping classes to their features. Experimental results are described in Section 4, and several perspectives are given in Section 5.

2. Interfacing a polygon hierarchy

We use a convex polygon hierarchy, adapted from the Yafool system [5]. As imposed by Java, the hierarchy is a single inheritance hierarchy, limited to classes $Polygon$, $Quadrilateral$ ($Rectangle$, $Square$, $Rhombus$) and $Triangle$ ($Equilateral$, $Isosceles$). Components of the public part of this class hierarchy are given in Figure 1. Two kinds of features are used. Methods are represented by their signature; method codes and internal data are omitted. The only data which are preserved are *constant static data*, i.e., data common to all the instances of a class. For example, $sidesNb$ for $Quadrilateral$ which is equal to 4 for all quadrilateral instances.

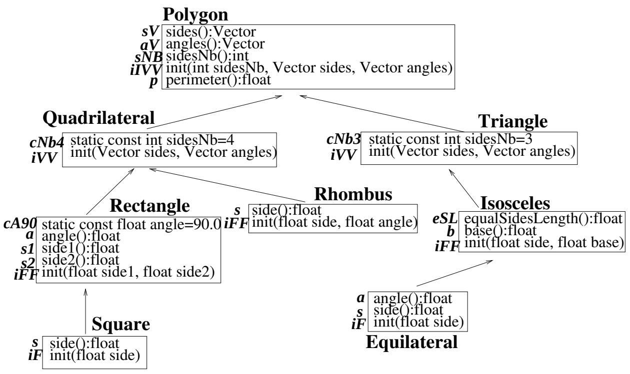

Figure 1. The public part of a polygon Java class hierarchy.

Due to the single inheritance constraint, this hierarchy has several drawbacks. One is that $Square$ can not be considered as a subclass of both $Rectangle$ and $Rhombus$. Another problem is that some features may be duplicated, because they can not be factorized correctly (for example $side()$, which is repeated in classes $Rhombus$, $Square$, and $Equilateral$). These problems are well-known by object designers, and several technical solutions have been proposed [11], but there are not always satisfactory. The fact that some features are not factorized presents two problematic aspects: when an update needs to be done, the programmer has to update all occurrences of the feature that are not rationally gathered under a common "root" (the upper occurrence in the hierarchy); this often causes errors. The second problematic aspect is that this lack of factorization also alters the readability of the hierarchy.

In order to reconsider the polygon class hierarchy, we need to flatten it. A class is associated with all the properties it inherits or declares. A class may redefine an inherited method, giving it a different code, but in Java the signature cannot be changed: two different signatures for a given method name correspond to two different features (note that changing only the return type is not allowed.). The relation obtained is called *owns-property*, and is shown in Table 1 for polygons (property names have been simplified as described in Figure 1).

Because of this single inheritance constraint, it is impossible to reorganize Java classes properly, but the idea we defend is that interfaces can be used to give an external view of classes which more acceptable from a sub-typing point of view (2). For an interface which is close to a class, the name of the class is used to form the name of the interface (for instance $IPolygon$). For interfaces that do not represent any class or which are close to more than one class, the name is chosen by a designer in relation to the meaning of the factorized feature set:

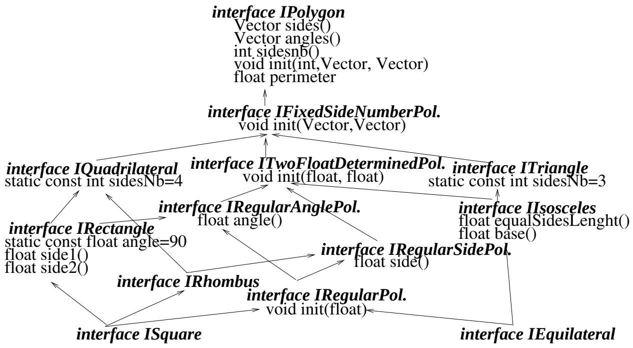

Figure 2. The interface hierarchy

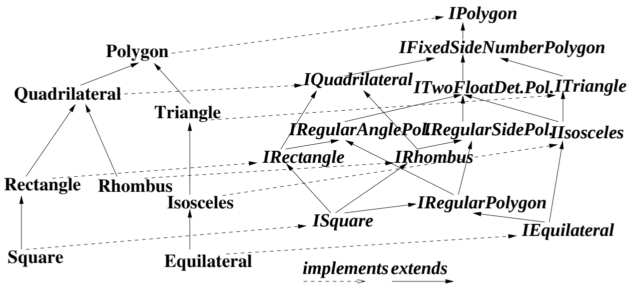

Figure 3. The complete hierarchy.

- *IFixedSideNumberPolygon* represents polygon classes whose instances all have the same side number;
- *ITwoFloatDeterminedPolygon* represents polygons determined by two floats (two sides, or two angles, or an angle and a side);
- *IRegularSidePol.* represents polygons that have equal sides;
- *IRegularAnglePol.* represents polygons that have equal angles;
- *IRegularPolygon* represents polygons that have equal angles and equal sides.

This interface hierarchy is a view on concepts which is in accordance to the usual mathematical classification, and clearly better than the class hierarchy from this point of view. It does not contain feature redundancy and it introduces interesting intuitive concepts about polygons that a designer can recognize and name as shown before. Note also that *Square* may now appear as a subtype of both *Rhombus* and *Rectangle*. Inheritance links furthermore respect the inclusion between feature sets.

In Figure 3 we show how this interface hierarchy is connected to the class hierarchy. The result is more complex than class hierarchy, but this disadvantage is counterbalanced both by a better conceptual organization and by the fact that the subsequent programming effort is reduced (interfaces are generated automatically). First, due to factorization, a feature modification will be restricted to the interface declaring the feature, to the classes implementing the interface, and their sub-classes. Second, the development

	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							
Quadrilateral	X	X	X	X	X	X		X									
Rhombus	X	X	X	X	X	X	X	X	X								
Isosceles	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. *"own-property"* **for polygons**

of an operation that applies to all regular polygons is now simplified because $IRegularPolygon$ can be used as a parameter type.

3 A new global algorithm for Galois sub-hierarchy construction

Our method intensively uses Galois lattices, and more specifically a sub-structure we call the *Galois sub-hierarchy*, first studied by [6], that are reference structures for building class hierarchies. We will explain their main principles, and why an algorithm for building a Galois sub-hierarchy was necessary.

3.1 Galois lattice and sub-hierarchy: foundations of hierarchy construction

Galois lattices (or concept lattices [12]) are used in several domains such as knowledge representation, machine learning (conceptual clustering), data mining, classification and software engineering [7]. A Galois lattice is the result of the formation of concepts from a binary relation between *objects* (in a broad sense) and their *features*. In our context, *objects* are Java classes. The concepts are organized into a partial order that may be interpreted as an inheritance relation. In a pragmatic approach introduced by [6], only concepts carrying informations are kept, leading to simpler structures, with a space complexity linear in the size of the input relation (the reader is referred to [3, 7] for more detail). The main advantage of these structures is that they guarantee a maximal factorization with a minimal number of concepts, giving at the same time an organization into a hierarchy. Concepts are organized according to the inclusion of their feature sets. If the factorization is based on the external or public features, we may expect a very interesting result from a subtype point of view.

To restrict to our specific problem, let $\mathcal{C}$ and $\mathcal{F}$ be respectively the class set and the feature set, and R the binary relation $owns - property$ between $\mathcal{C}$ and $\mathcal{F}$. The construction is based on the two fundamental and very symmetrical mappings $feat$ and $class$ defined below.

$$forX \in \mathcal{P}(\mathcal{C}),\ feat(X) = \{y \in \mathcal{F} \mid \forall x \in X, (x, y) \in \mathcal{R}\}$$
$$forY \in \mathcal{P}(\mathcal{F}),\ class(Y) = \{x \in E \mid \forall y \in Y, (x, y) \in \mathcal{R}\}$$

Definition 3.1 *Let O_C be the set of pairs $(class(feat(\{e\})), feat(\{e\}))$, $e \in \mathcal{C}$; let O_P be the set of pairs $(class(\{a\}), feat(class(\{a\})))$, $a \in \mathcal{F}$.*
The Galois sub-hierarchy $\mathcal{S}$ is the set $O_C \cup O_P$, given with $<_{\mathcal{S}}$ which is defined by $(X_1, Y_1) <_{\mathcal{S}} (X_2, Y_2)$ iff $X_1 \subset X_2$ (which is equivalent to $Y_2 \subset Y_1$).

The Galois sub-hierarchy can also be defined by a simplification of the Galois lattice as proposed in [6, 7] (see [4] for a proof of the equivalence between the two definitions). This allows feasibility avoiding the worst case complexity of the whole lattice that is exponential with respect to the number of classes and to the number of features. Another presentation of the Galois sub-hierarchy consists in eliminating a redundancy in the structure that can be observed between the node labels (the closed set pairs) and the order. A feature (resp. a class) belonging to a node belongs to all the nodes below (resp. above), as it was inherited. The simplification is based on the observation that features and classes need be quoted only in their introducing node. More formal definitions can be found in [7, 4]. The left part (or extent) of a simplified pair contains the names of the input classes "defined" by this node (these classes have exactly the features of the complete right part (or intent), they will implement the interface corresponding to this node). When this extent is empty (This pair is necessarily a pair of O_P), a new factorization superclass has been discovered. The intent of a simplified pair contains the features "declared" by the vertex. When this intent is empty (this pair is necessarily a pair of O_C), all the features of the node are declared by upper nodes. The Galois sub-hierarchy preserves the more informative part of the Galois lattice: the nodes that either define at least one class, or declare at least one property; all the other nodes are eliminated.

3.2 Towards a more efficient computation

First, as global hierarchy construction algorithms have previously been proposed, the need for a new algorithm is discussed. We proved in a preliminary comparative study [4], that some of the existing global algorithms [2, 1, 10] do not identify the produced structure, but build a hierarchy very close to the Galois sub-hierarchy, missing some of the links and adding a few classes that do not correspond to pairs of closed sets. Another algorithm which has been proposed ([9]) builds the Galois sub-hierarchy in a framework where input classes will always be associated with leaves of the output Galois sub-hierarchy. As a result, we had to propose an algorithm building the entire Galois sub-hierarchy, and with a simplified form for closed set pairs, since it is the form useful to generate Java interfaces.

The constructive definition given in the previous section for the Galois sub-hierarchy can be considered as a basic algorithmic scheme. Nevertheless, a brute force algorithm following this scheme would not be very efficient. Let us remark that, O_C and O_P may intersect, and that several features (resp. classes) can lead to the same closed set pair. The pairs have to be ordered with respect to inclusion in order to define the inheritance links, and without introducing transitivity edges.

The two following ideas are used in order to reduce the complexity. First closed sets generation is optimized, ensuring that elements of O_C, O_P and $O_C \cap O_P$ are produced only once and limiting the examined part of the binary relation when a closed set pair is generated. Second, inheritance link production is optimized: the order the closed sets are generated in respects their top-down order in the structure.

The algorithm is detailed and proved in [8].

4 Experimental results

This algorithm is implemented in Java (JDK 1.2) as part of a platform dedicated to the development of class hierarchies.

The method has been tested with the API of Java and with a Java application developed by FRANCE TÉLÉCOM R&D, the PÉTRA project, that deals with the management of telecommunication networks, and to the API of Java. The resulting interface hierarchies allowed:

- to validate class organization, and have a basis to discuss the interfaces already in the API(s),
- to improve classes: when possible, new factorization classes may be added; interfaces with very few features reveal sometimes features that have been omitted in some classes,
- to give a view independent from implementation; this is appreciated by developers who prefer exchange interfaces to classes;
- to produce more generic code, using interfaces as variables or parameter types.

5 Conclusion

As a perspective in the context of Java, this work could be helpful in guiding the reverse design of classes, in particular using association links to simulate inheritance links when multiple inheritance would be necessary. Another point is doing a similar work in the context of CORBA, to automatically extract IDL interfaces from Java applications. This could be added to the tools that Java proposes for CORBA applications.

Note. This work is done in collaboration with France Telecom R&D.

References

[1] J.-B. Chen and S. C. Lee. Generation and Reorganization of Subtype Hierarchies. *Journal of Object Oriented Programming*, 8(8), 1996.

[2] W. 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.

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

[4] H. Dicky, M. Huchard, and H. Leblanc. Etude d'algorithmes en relation avec la construction globale de sous-hiérarchie de Galois (english version available "The Galois lattice may characterize class hierarchies built by different algorithms). Technical Report 99059, LIRMM, Montpellier, 1999.

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

[6] 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.

[7] R. Godin, H. Mili, G. Mineau, and R. Missaoui. Conceptual Clustering methods based on Galois lattices and applications. *Revue d'intelligence artificielle*, 9(2), 1995.

[8] M. Huchard and H. Leblanc. Présentation d'un algorithme global de réorganisation de hiérarchies de classes (english version available). Technical report, CNET, Décembre 1999. Rapport d'avancement 03.99.

[9] G. Mineau, J. Gecsei, and R. Godin. Structuring Knowledge Bases Using Automatic Learning. *Proceedings of the sixth International Conference on Data Engineering*, pages 274–280, 1990.

[10] I. Moore and T. Clement. A Simple and Efficient Algorithm for Inferring Inheritance Hierarchies. *TOOLS Europe 1996 Proceedings, Prentice-Hall*, 1996.

[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design*. Prentice Hall Inc. Englewood Cliffs, 1991. pages 15–84.

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