Inheritance Hierarchies and View Hierarchies —Position Paper—

Werner Nutt

Department of Computing and Electrical Engineering
Heriot-Watt University, Edinburgh EH14 4AS
nutt@cee.hw.ac.uk

1 Introduction

In many areas of computing, such as programming languages, databases, conceptual modeling, and knowledge representation we encounter formalisms that allow one to group *objects* into *classes* and to organize the classes into hierarchies. The intuitive understanding is that classes further down in the hierarchy are subsets of the classes further up. The following are examples for such class hierarchies.

Programming Languages: data types in a subtype/supertype relationship (e.g., positive integers as a subtype of integers, or nonempty lists as a subtype of all lists) and classes in object oriented programming;

Databases: queries contained in other queries (e.g., the set of employees with a salary above GBP 30,000 contained in the set of employees with a salary above GBP 20,000) and classes in object-oriented databases;

Conceptual Modeling: entity sets in ER-modeling or classes in object-oriented modeling related to each other by is-a relationships;

Knowledge Representation: frames or "concepts" in description logics with is-a relationships.

We argue that, in spite of the apparent similarity of all these formalisms, there are *two* essentially different *ways* in which such class hierarchies are employed (see also [BDNS98]).

1.1 Class Hierarchies for Inheritance

One way is to use the hierarchy as a structure along which to organize inheritance, for instance of class attributes, methods, or frame slots. The goal is to avoid duplication of code and thus to create more abstract and modular software. Examples for this first kind of usage are class hierarchies in object-oriented (oo) programming languages or oo databases, hierarchies in conceptual modeling and is-a relationships between so-called "primitive" frames and concepts in knowledge representation.

1.2 Hierarchies of Classes Defined as Views

The second kind of usage is to start with some primitive classes and there properties and then to specialize them to yield new classes by imposing constraints upon the elements of the primitive classes. An example would be to define, based on the primitive class "employee," the class of "well-paid employees" by constraining the attribute `salary` to values $\geq 30,000$. This approach is similar to defining a new relation as a view on top of base relations. Among the reasons for introducing such classes are all the classical ones for introducing views in relational databases.

Once a collection of such view classes is defined, it is natural to ask whether the definition of one class, say C_1, is more general than the definition of another one, say C_2. For instance, if the class of "medium-paid employees" were defined by imposing the constraint `salary` $\geq 20,000$ on employees, this definition would be more general than the one of "well-paid employees." Whenever the primitive classes are populated by objects, the view class C_1, due to its more general definition, will be a superset of the view class C_2. The more general/less general relationship based on definitions has been studied in several areas of computer science, most notably in databases, where it is called *containment* [CM77], and in description logics, a branch of knowledge representation, where it is called *subsumption* [DLNS96]. A major motivation behind this work is that knowledge about containment and subsumption can be exploited to optimize the execution of queries.

Clearly, what cannot be achieved by introducing views is inheritance of properties. Here, the superclass/subclass relationship is a *consequence* of the properties of classes. In the case of inheritance, it is a *precondition.*

In the rest of the paper, we illustrate the differences between the two kinds using class hierarchies by way of examples from the area of programming. We will argue that specialization by constraint does not support inheritance and discuss difficulties in updating instances of view classes.

2 Class Hierarchies and Inheritance in Programming

A premier motivation for introducing class hierarchies in programming is to support the writing of more abstract and therefore more easily reusable code.

Suppose we have the class of objects of type person, where every person has an attribute dob for their date of birth. For such persons we can write a method `age` that computes their current age from their dob and the actual date. In an application, there may be many person objects around, e.g., employees, managers, and customers. If they have a `dob`, we can apply our method `age` to them as well.

How can we find out whether they have a `dob`? One way would be to look up their definition: whenever there is an attribute `dob`, we happily compute the age using `age`. However, what if there is a class `gradStudent` with an attribute `dob` standing for the date at which they graduated with their Bachelor's degree? Obviously, it makes little sense to apply the `age` method to this class.

The approach in programming is therefore not to *decide* subclass relationships according to the properties by which classes are defined, but, conversely, to *postulate* subclass relationships and to let the subclasses *inherit* properties (and methods) from their superclasses.

This is in principle a simple and straightforward mechanism. Difficulties arise only if the subtype relationship becomes complex, or if the inheritance is complicated. For instance, if a class is the direct subclass of two classes, and the two classes have two methods with the same name, then it is not clear a priori which of the two methods is to be inherited, and a protocol for multiple inheritance is needed. In other situations, a programmer may even want to preempt the inheritance of a certain attribute or method.

3 Specialization by Constraint

Subclasses inherit from superclasses, e.g., in a natural company hierarchy, manager inherits from employee and employee from person.

This makes sense intuitively, since every manager is an employee and every employee is a person. Still, if we assume that manager objects have additional slots, e.g., the departments they manage, manager objects are not exactly like other employee objects, since they differ in structure.

Orthogonally to managers, we may want to introduce the class of senior employees, which are employees that are at least 50 years old. Differently from managers, no additional assertion is needed for an employee object to become a senior employee. All we need to do is to apply the age method and check the value. Also differently from managers, the fact that an employee object is senior does not entail that it inherits attributes or methods.

Let us consider another example, which has been suggested by Hugh Darwen. An ellipse has two axes with two lengths. For the sake of simplicity we assume that we only consider ellipses with axes parallel to the x- and y-axes of the plane, centered at the origin. (In the general case, we would also need a pair of numbers to describe the center of the ellipse and an angle to describe the orientation of the ellipse in the plane.) We can model such ellipses by a class `ellipse`, which has two real-valued attributes, say `xlength`, `ylength`.

A circle can be seen as a special ellipse where both axes have the same length. If we are able to define classes by constraints, we can introduce the class `circle` as the set of all ellipses that satisfy the constraint

$$\texttt{xlength} = \texttt{ylength}. \tag{1}$$

For a circle, it does not make sense to distinguish between two axes. A circle only has a radius. We therefore would like to suppress the attributes `xlength` and `ylength` and to replace them with a new attribute `radius`. Differently from the two length attributes of an ellipse, which are the data that constitutes an ellipse, `radius` would have to be *defined* in terms of `xlength` and `ylength`, because the radius is derived from the length of the axes. One of the several possible definitions would be `radius` := `xlength`.

Now, suppose we have one `ellipse` object `e101` with `e101.xlength` = 5 and `e101.ylength` = 5. Then, via specialization by constraint, `e101` is also an instance of `circle`.

We want to discuss what should be the effect of *updating* `e101`. We consider two cases.

– Updating a length of the ellipse:

$$\texttt{e101.xlength} := 10. \tag{2}$$

Then `e101` doesn't satisfy any more the constraint defining circles, and consequently, `e101` fails to be a circle.

– Updating the radius:

$$\texttt{e101.radius} := 10. \tag{3}$$

Here, the question is how to understand the update. If we understand `radius` as something like a method, then the update doesn't make sense and has to be rejected.

If we understand `radius` as an alias for the attribute `xlength`, then (3) is equivalent to (2). Under this interpretation, however, the strange effect of updating the radius of a circle would be that the circle ceases to be a circle. Alternatively, we may require that updating an ellipse as a circle maintains the properties that turn it into a circle. Since the definition of a circle is unusually simple, there is an easy way out. If we update the attribute `xlength`, then taking into account the defining constraint `xlength` = `ylength`, we have to apply the same update to `ylength`, so that (3) is equivalent to

$$\begin{aligned}\texttt{e101.xlength} &:= 10\\ \texttt{e101.ylength} &:= 10.\end{aligned}$$

The result is again a circle.

In a more general setting, we may allow to update an object in a view class only, if that update translates unambiguously into an update of the underlying primitive class. Clearly, whether this is possible or not depends on how the view class has been defined. The question, under which conditions and in which ways objects in views can be modified has been investigated under the heading of the "view update problem" [FGM96].

4 Conclusion

Whether a system should support predefined class hierarchies or hierarchies computed according to the definition of classes (or both) depends on the purpose to be served.

We have argued, that the main purpose for predefined hierarchies is to allow for inheritance of properties and thus to facilitate better software design. Computed hierarchies occur when classes are defined as views. Knowledge about such a hierarchy can be used to optimize query execution or, more generally, search.

Finding out the hierarchy between classes induced by their definition is considerably more difficult than keeping track of explicitly given subclass/superclass relationships. Obviously, if arbitrary relational queries were allowed in definitions, then the relationship is undecidable because of the undecidability of entailment in first order predicate calculus. If only select-project-join queries are allowed, then entailment would still be NP-hard. Thus, a language for defining class hierarchies that is both expressive and where computation of hierarchies is feasible, requires a careful design.

References

[BDNS98] M. Buchheit, F.M. Donini, W. Nutt, and A. Schaerf. A refined architecture for terminological systems: Terminology = Schema + Views. *Artificial Intelligence*, 99(2):209–260, 1998.

[CM77] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proc. 9th Annual ACM Symposium on Theory of Computing*, 1977.

[DLNS96] F. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. In G. Brewka, editor, *Principles of Knowledge Representation and Reasoning*, Studies in Logic, Language and Information, pages 193–238. CLSI Publications, 1996.

[FGM96] Jose Alberto Fernandez, John Grant, and Jack Minker. Model theoretic approach to view updates in deductive databases. *J. Automated Reasoning*, 17(2):171–197, 1996.