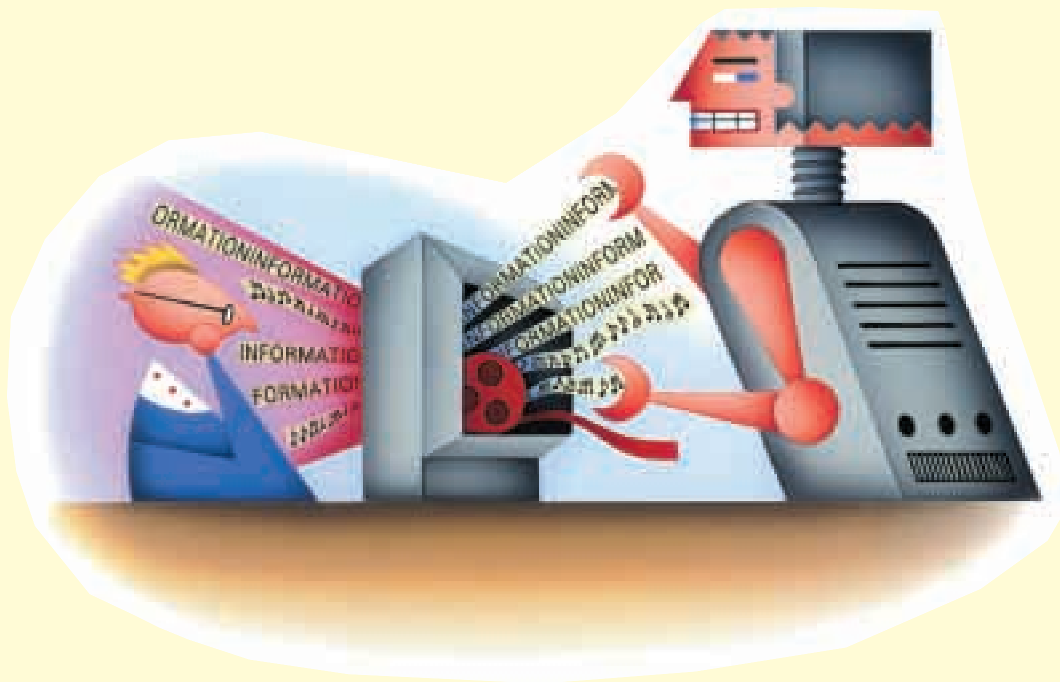

Instances and Classes in Software Engineering

Christopher A. Welty

David A. Ferrucci

Over the past decade or so, one of the many areas that artificial intelligence has influenced is software engineering. Although artificial intelligence cannot lay a particular claim to such notions as taxonomies, relational models, activity/state diagrams, inheritance, or other techniques that are now part of mainstream software engineering, CASE, and database design, it is within the purview of AI that they were developed. Along the way from artificial intelligence to software engineering, some of these notions lost their meaning.

Christopher A. Welty
Vassar College
Computer Science Dept.
Poughkeepsie, NY
weltyc@cs.vassar.edu

David Ferrucci
IBM Watson
Research Center
Yorktown Heights, NY
ferrucci@us.ibm.com

Within the past 10 years, object-oriented analysis and modeling has led to significant improvements in productivity in the earlier phases of software engineering (such as requirements, design, and implementation). These improvements have been facilitated in part by the emergence of tools to support the process and languages such as UML for supporting partly standardized diagrams.

The foundations of object-oriented modeling derive from philosophy through artificial intelligence to software engineering. In philosophy, the fields of epistemology and ontology have been considering for centuries what is needed to represent things in the world and what it means to do so. Within the past century, the development of first-order logic and, later, computers has significantly sped up and formalized many of these ideas.

Knowledge representation (KR) is one of the original subfields of AI, and the study of modeling knowledge in a computer has always been its central theme. The modeling facilities provided by object-oriented languages, and the diagrammatic representation offered by UML and its predecessors, were developed and studied by KR researchers in the 1970s.

The purpose of this article is not to claim credit for the success of object-oriented analysis and modeling, but to show how the transfer of this technology from KR to software engineering was not complete. We have encountered numerous examples of classic logical pitfalls in models developed by novices and by experts; with this article we begin what we hope will be a series of articles on the proper use of AI techniques in software engineering. We offer a few simple examples of the *limitations* of the object-oriented approach for modeling.

Let us begin by considering some knowledge about eagles: An eagle is a kind of bird, and we happen to know a particular eagle, Harry. Object-oriented systems provide two basic modeling primitives that we can use to

represent these objects: *classes* and *instances.* A class describes a set of instances, and an instance describes an object in the real world. In addition to these two primitives, object-oriented systems provide two basic relationships that we can use to help define how these objects relate to each other: **subclass** and **instance.** Using these facilities, we can construct a model of our simple domain, as shown in Figure 1.

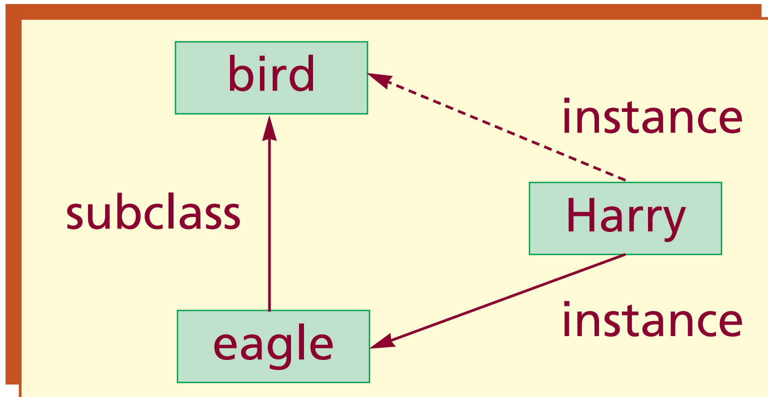

Figure 1. An instance of a subclass. In object-oriented languages, an instance of a subclass (i.e. derived class) is also an instance of the superclass.

Even this simple example raises what has always been an interesting idea in KR: When an object is an instance of a class, and the class is a subclass of some base class, the instance is also considered an instance of the base class. This property of object-oriented systems is sometimes mistakenly called *inheritance,* a misnomer that refers to something else.

What makes **Harry** a **bird** is the implied semantics of the primitives being used in the model. This is one of the gray areas that is not typically taught as part of any introduction to object-orientation. The original semantics of a *class* is as a set, the set of all instances of that type. A class definition includes a description that applies to all members of the set. The subclass relationship is actually the inverse of a subsumption relationship (i.e. subsumed-by), which from the logic of sets means simply subset. In other words, according to our model the set of all eagles is a subset of the set of all birds.

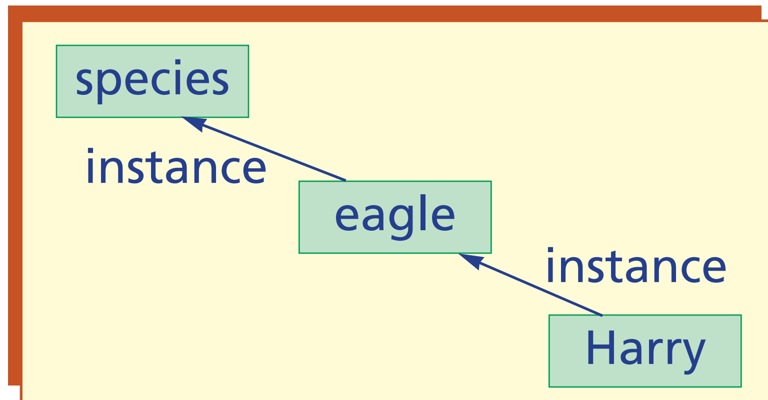

Figure 2. Object-oriented languages do not support instances of instances. The result is that domain modelers are forced to work around the semantics of the representation system.

We can quickly go from the trivial to the overly complex in our domain by adding the notion of species. The proper interpretation of species is shown in Figure 2. Technically this is not quite correct—an eagle is neither a species nor a proper scientific classification of birds at all, but our point is philosophical, not zoological. If we take as a given that eagle is a species, then **species** is the set of all species, and **eagle** is a member of that set. In an object-oriented language, however, we cannot represent something that is both an instance of one class as well as a class itself.

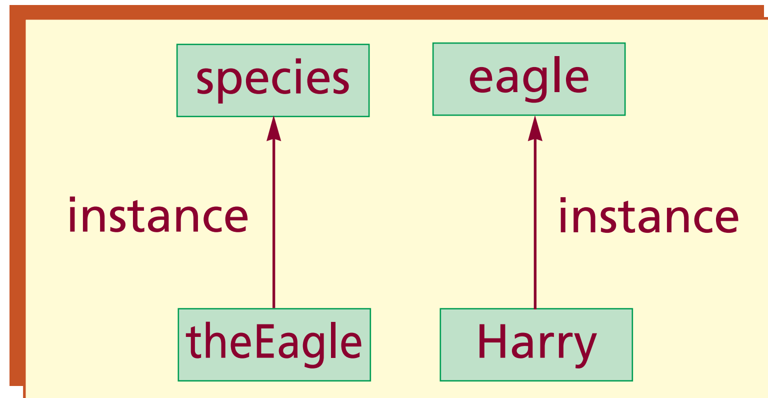

Figure 3. One possible workaround is to split the representation of eagle into two objects, one an instance of species and the other a class. What is the relationship between "theEagle" and "eagle"?

It is important to understand that Figure 2 shows the correct representation of the relationships among **species, eagle,** and **Harry**. Given a domain such as this, how can we represent it using object-oriented modeling?

Figure 3 shows one approach that breaks the **eagle** class into two parts: one is an instance of species, the other is a class of which **Harry** is an instance. The **eagle** instance has properties like the Boolean

"endangered"; the **eagle** class describes the properties of eagles in general, such as "sharp beak." This approach is within the power of object modeling to represent; however, the model in Figure 3 does not capture the true nature of the domain—it completely ignores the fact that **eagle** the class and **eagle** the instance refer to the same thing.

Another approach is shown in Figure 4. This is the most common solution we have come across when even experienced modelers are presented with this kind of problem. It is also the most incorrect solution. Although the approach shown in Figure 3 does not capture all the domain knowledge, it does not add anything that is incorrect. In Figure 4, by virtue of the semantics discussed earlier, we have made **Harry** an instance of **species,** which is certainly not correct.

This is a fairly simple example of a common problem that arises in domain modeling and cannot be properly addressed in an object-oriented model. We should note that this is not an academic thought-problem, but in fact prototypical of a class of problems that crop up quite frequently. For example, consider a domain model for a debugger. Figure 5 shows some of the objects in the domain that would need to be modeled. The debugger will need to know about data types and variables since they are part of the code, and it will also need to keep track of the relationship between variables and their data types. Again, the proper way to represent this is shown in Figure 5.

Some dynamic object languages, such as Smalltalk and Common Lisp Object System (CLOS), allow one exception to the rule that instances cannot be classes. In these systems, every class is an instance of the special class **class.** Classes can have two parts (see Figure 6)—a *meta-class* that contains properties of the class that are not passed down to instances, and the normal instance description.

However, using the meta-class facility is not a solution to the problem we outlined earlier, because classes are allowed to be instances of only one class: the class **class.** Therefore, we cannot represent the **species** object or have a **data-type** class whose instances are themselves classes that can have instances. Figure 7 illustrates this problem.

To gain a better understanding of this problem and how to recognize it, we have to

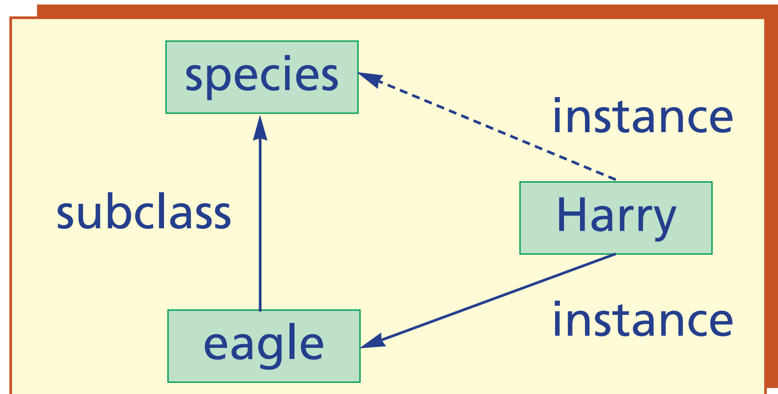

Figure 4. A more obviously incorrect workaround is to make eagle a subclass of species instead of an instance, which results in Harry being a species.

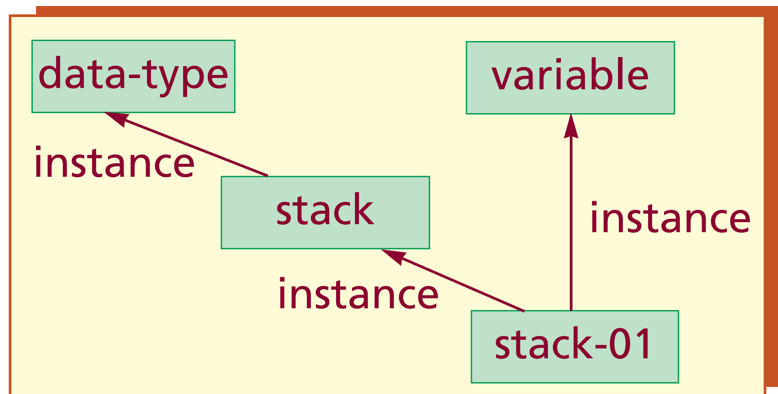

Figure 5. The problem with modeling multiple levels of instantiation does not only occur in hypothetical environmental domains. In this example, we are trying to model the objects used in a debugger, which needs to keep track of the data-types defined in a program, and the variables that instantiate those data-types.

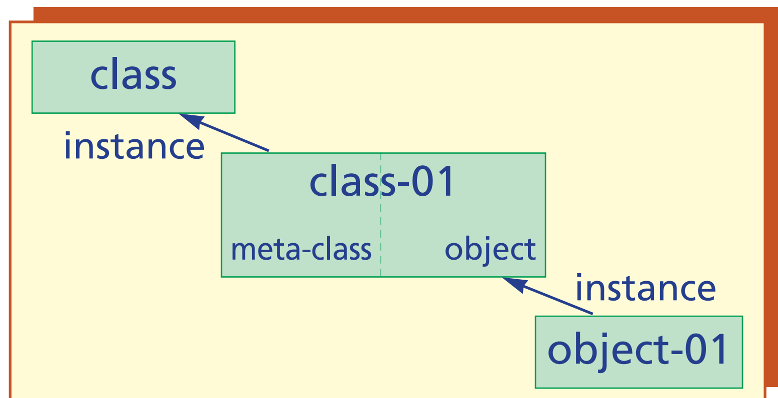

Figure 6. Smalltalk, CLOS, and other dynamic object languages provide a "meta-class" facility that provides for the capability to represent a class as both an instance and a class. Class descriptions are separated into two parts—one describes the properties of instances of the class, the other describes the properties of the class itself.

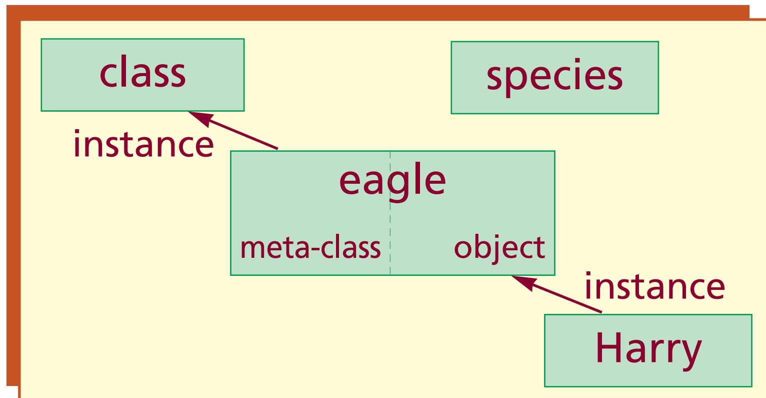

Figure 7. The meta-class facility is not a solution to our trivial modeling problem, because the interpretation of a class as an instance is only permitted as an instance of "class." The eagle class cannot be an instance of species.

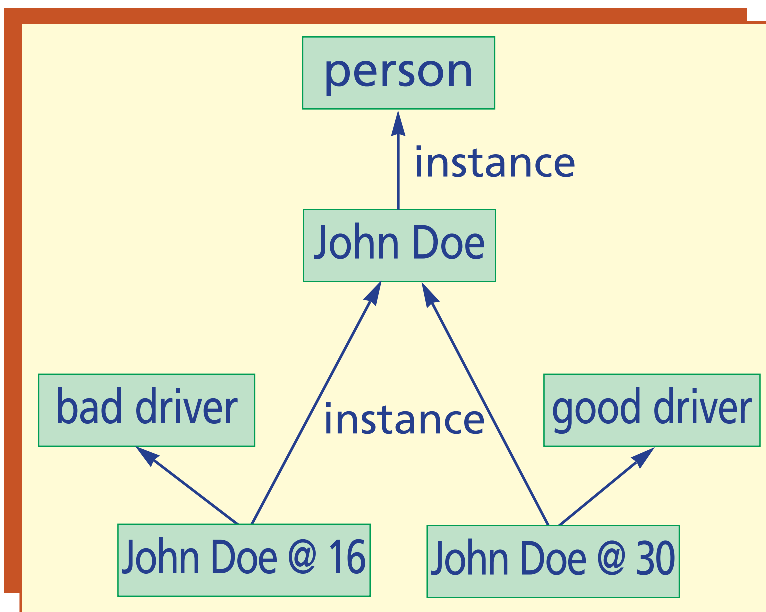

Figure 8. Even objects that most experienced modelers agree are clearly instances, such as a person, can still cause modeling problems. In this example, an insurance company may want to track a person's driving record over time. Clearly John Doe at 30 is a new man.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
© ACM 1523-8822 99/0600 $5.00

revisit logic. In fact, problems such as these were discussed almost a century ago by the first logicians as they experimented with their new tool (formal logic).

Once again, a class is normally interpreted as a set, and in predicate logic sets are represented as unary predicates, for example, *Eagle(Harry).* This implies that to represent eagles as a kind of species we would have to say *Species(Eagle).* The problem with this expression is that the symbol Eagle has already been used as a predicate, therefore the statement *Species(Eagle)* is a predicate of a predicate. Predicating a predicate symbol is the definition of *second order* in logic. Second-order logic is highly complex and computationally intractable.

The point we wish to express is that object-oriented languages are first order. Modelers using this technology should be aware of this and should understand the semantics of the primitives (classes, instances, and the subclass and instance relationships). They should also be familiar with the standard and well-known pitfalls of first-order systems so they can avoid making confusing or costly mistakes.

Finally, there is a related problem regarding the semantics of instances. We suggested here that instances are to be interpreted as members of sets, and we showed several examples of how the same object can be thought of as both a set and an instance. A frequent response to this point is that eagles and data types aside, there are certainly examples of things that are instances and always will be. Normally these obvious instances are objects that denote "real things," such as people or parts of an inventory.

A well-known argument in philosophy is that no matter how you slice it, there is always a way to think of anything as a class. We apply this philosophizing to a practical problem, as shown in Figure 8. An insurance company develops a software system to track its customers, and for each customer it maintains a record of whether the customer is a good or a bad driver. The problem with this approach is that when a customer changes from good to bad (or vice versa), the old classification information is lost.

The insurance company decides to change its model and keep track of information about people over time. One way to approach this is to think of each person, such as John Doe in Figure 8, as a class whose instances represent the "way a person was" at a certain age. Once again we have a model that seeks to represent instances of instances, and one that gives new meaning to the expression, "I'm a new man."