Inheritance: One Mechanism, Many Conflicting Uses*

Stephen H. Edwards

Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, OH 43210
Tel: (614) 447–9803
Email: edwards@cis.ohio-state.edu

Abstract

Inheritance has many beneficial uses, but merging them all into a single hierarchy can bring out serious conflicts. The reasons there are such conflicts, and the reasons why they are commonly allowed in OO languages can be seen by examining the different applications of inheritance, and who makes use of them. To illustrate this point, this position paper includes a rudimentary taxonomy of the various uses for inheritance in OO programming. The primary partition in the taxonomy, between inheritance as used by the implementer of a class or as used by the client of a class, helps to point out the ways various uses of inheritance can conflict, and can hamper effective programming.

Keywords: Inheritance, Object-oriented programming, reuse, specification inheritance, code inheritance.

Workshop Goals: Cross-fertilization of ideas with other researchers; building on the results of WISR'92; keeping abreast of other ongoing reuse work; advancing the theoretical foundations of software reuse.

Working Groups: Design guidelines for reuse, reuse and formal methods, reuse and OO methods, reuse handbook.

*This work is supported in part by the National Science Foundation, CCR-9111892.

1 Background

For the past several years, the Reusable Software Research Group at the Ohio State University has been exploring the technical problems of software reuse, focusing on a disciplined approach to software engineering as a possible solution. In aiming for a discipline of software construction that will scale up to large systems, the RSRG has centered on the concept of supporting modular reasoning or *local certifiability* at the component level [1, 2]. In short, the client of a reusable component or subsystem should only have to consider a bounded amount of information (i.e., local, rather than global) when reasoning about whether that component is the correct one to choose, and whether it is being used correctly. [3] describes one way to ensure that all components have this property, by applying a specific software discipline.

For most OO languages in use today, programmers are likely to take modular reasoning about classes as a given. Unfortunately, the inheritance mechanism as it is most commonly realized actually *prevents* reasoning about classes in a modular fashion, often in very subtle ways [4]. It is possible to overcome this problem by only using inheritance in a disciplined way. This position paper explores some of the reasons for the conflicts between inheritance and modular reasoning, which are actually conflicts between different ways that inheritance is used in programming, and in reasoning about classes.

2 Position

Inheritance, which allows new items to be defined with respect to already existing items, is now widely used in software construction. In fact, inheritance is almost universally regarded as a cornerstone of methods termed "object-oriented" [5]. One reason for the success of inheritance is the fact that it meshes with human cognitive abilities—new entities are defined by saying how they are different from existing entities, which is very similar to the way people seem to learn by association, learn rules of generality, and learn exceptions to such rules.

In a programming context, inheritance has many beneficial uses, but merging them all into a single hierarchy can bring out serious conflicts. Many OO practitioners know that certain inheritance practices are not advisable, such as hiding or removing methods in descendant classes, or arbitrarily redefining method semantics, because subclasses that do not behave as expected may be produced. What is the real reason for such conflicts, and why are they allowed in current OO programming languages? An answer can be found by examining the different applications of inheritance, and *who* makes use of them.

Figure 1 depicts a taxonomy of the various uses for inheritance in object oriented programming (OOP). This taxonomy is not meant to be exhaustive, but it does illustrate several interesting divisions. The primary partition in this figure is between inheritance in the *implementer's dimension* and inheritance in the *user's dimension* [6]. From the implementer's perspective, inheritance has many uses in creating or defining new modules, classes, or objects. From the client's perspective, inheritance has many uses in understanding, applying, and reasoning about classes.

Note that for the remainder of this discussion, I will use the term *class* to denote a software artifact defined within an inheritance framework. Similarly, the terms *child* class or *subclass* will be used to refer to a software artifact that inherits from some *parent* class or *superclass*. I have chosen these terms since they are commonly used and understood, but I do not wish to imply that only class-based inheritance systems will be considered [6].

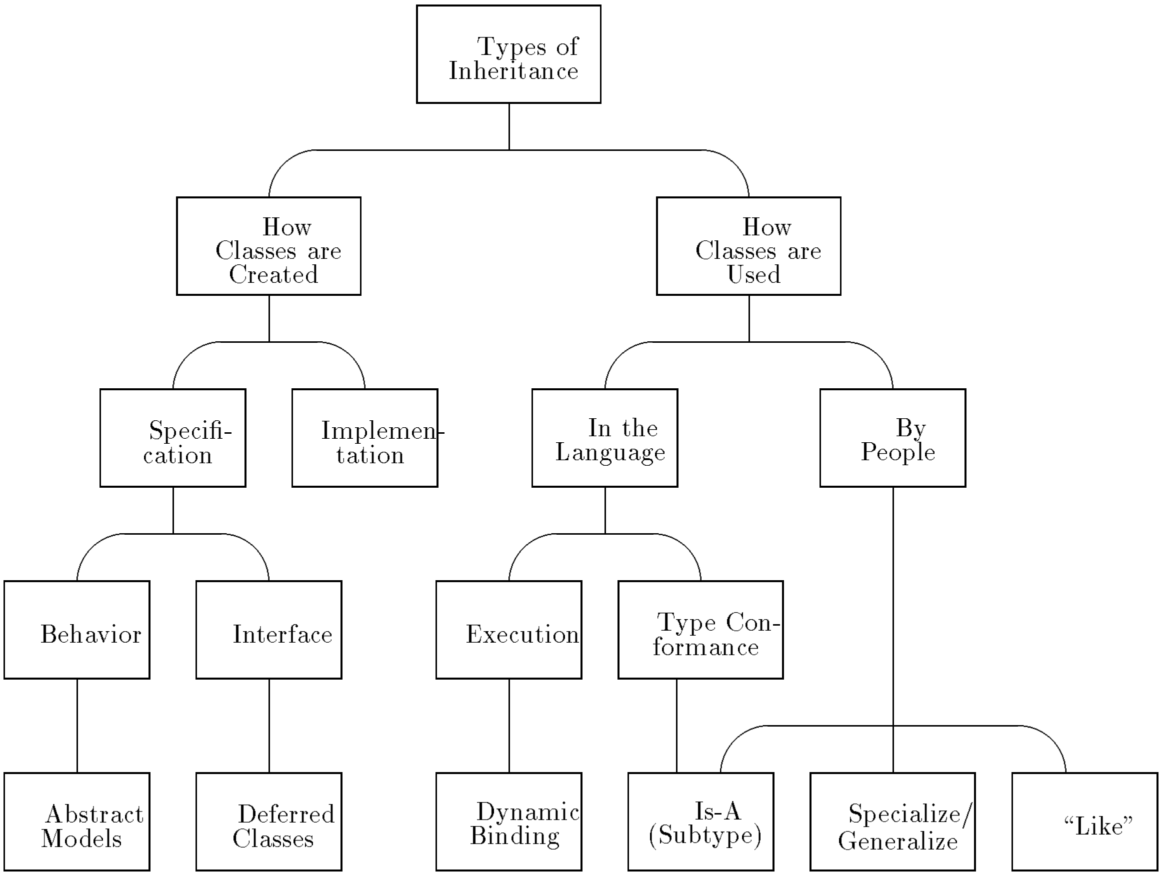

Figure 1: A Taxonomy of Uses for Inheritance

From the implementer's perspective, inheritance is often used as a method of *programming by difference*. This programming style allows implementers to define new classes differentially, simply by capturing how they differ from some pre-existing class. Traditionally, this capability has been put forth as one of the key benefits provided by OO languages, and as one of the main sources of reuse in OO environments.

The implementer can use inheritance to define the specification of a new class, the implementation of a new class, or both. Further, when defining the specification of a class by difference, the implementer can treat the parent class in one of two separate ways:

- The specification of the parent class defines a syntactic interface or protocol. The same interface, with possible variations in behavior, will be inherited by the subclass. This technique is often used in event-driven frameworks, where the inherited interface defines the call-back points the framework will use to invoke class-specific behaviors, without restricting what those class-specific behaviors might be. Virtual or deferred class definitions are one way of providing such a syntactic interface description in C++ or Eiffel.
- The specification of the parent class defines a behavioral interface, possibly through the use of an abstract model of object state. This same behavioral description will be inherited by the subclass. This is usually the stance taken when one wants to equate a *subtype* relationship with the *subclass* relationship. This type of inheritance can be further restricted by requiring that the subclass have the same abstract model as the superclass, rather than simply requiring that the behavior be the same.

From the client's perspective, inheritance is not used to define new classes, it is used to understand and apply existing classes. Inheritance can be used within the language definition to determine how classes can be used in language constructs. In this vein, inheritance is most often used for two purposes. First, it is used in the execution model for many OO languages to describe the notion of *dynamic binding*, and how that determines what code operations are executed. Second, inheritance is very commonly used to determine type conformance for parameters to operations. Most languages that use this approach choose to interpret inheritance relations as "is-a" or subtype relations. A class B is a *subtype* of another class A (B `is-a` A) if any instance of B can be used wherever instances of A are required [6]. The combination of subtyping and dynamic binding is often the key to providing polymorphism in OO languages.

In addition to defining the way classes are used within language constructs, inheritance can also be used to enhance the way people reason about and use classes. In [6], three distinct forms of inheritance that are useful for clients are described: is-a relationships, specialization (or generalization) relationships, and "like" or similarity relationships. The is-a relation that has already been described can clearly be used by clients to lower the cognitive load of reasoning about collections of classes. A similar relation, that a class B is a specialization of another class A, can also be used by clients as an aid to understanding. A class B is a specialization of A if the instances of B can be obtained from those of A through some form of restriction. [6] gives the following examples of specialization:

> ...Strings can be viewed as specializations of arrays in which the elements must be characters, arrays can be viewed as specialization[s] of dictionaries in which the keys (subscripts) must be positive integers. [6, p. 219]

Finally, the "like" relationship implies that two classes are the same except for some clearly delineated differences. A set is like a bag, but it does not permit duplicates to be inserted. This relation can be used to structure a collection of classes into a hierarchy that promotes understanding and careful "chunking" of differences (as opposed to a hierarchy set up for strict subtyping, or for defining specifications by difference).

Note that in typical practice (e.g., C++, Eiffel), an OO language only only supports a single inheritance hierarchy and *all* of these alternatives are blended together in this single language mechanism. Programmers working in OO environments most likely focus on inheritance as a balance between a subtype hierarchy, a code inheritance hierarchy, and a (syntactic) interface inheritance hierarchy.

From the point of view of supporting modular reasoning about programs, the uses of inheritance depicted in Figure 1 pose many challenges. When reasoning about the correctness of a given class C, one must wear the hats of both implementer and client:

- To reason about the correctness of a class C (which consists of a specification and implementation), one must consider the other classes (and inheritance relations) that went into defining the interface and code that realize C.
- To reason about the correctness of the code realizing the class C, one must also consider the classes that C is a client of.

Thus, all of the uses of inheritance in Figure 1 must be considered—not just those seen from the client's perspective.

Unfortunately, formal treatments of inheritance often center around an is-a interpretation of the inheritance hierarchy, which fails to address all of the inheritance uses discussed above. This interpretation is also at odds with the way inheritance is used in practice. If the interpretation were correct, and only true is-a relations were permitted in an inheritance hierarchy, modular reasoning would then be feasible. This is the approach taken in RESOLVE, where an implementer may only make use of inheritance through a very tight interpretation of abstract model specification inheritance (at the lower left of Figure 1). This restriction ensures that the single inheritance hierarchy present can only contain is-a relations.

Most often, however, this restriction is not made. A typical OO language like Smalltalk, C++, or Eiffel contains a single inheritance mechanism which is used for *many* (or even all) of the purposes described in Figure 1. This hopelessly muddles inheritance of implementation or representation details with the more abstract inheritance relations like subtyping. As a result, modular reasoning is not feasible, since the inheritance relationship itself does not have a stable meaning.

Further, class-based OO systems like Smalltalk, C++, and Eiffel pose specific problems for modular reasoning:

> In general, a class mechanism enforces the restriction that all objects of a specific class have the same representation. [6, p. 214]
>
> Neither [class-based system] separates the two notions of specification and implementation completely because there is always a one-to-one correspondence between the two. [6, p. 227]

The intertwining of specification and implementation concepts within a language can hamper modular reasoning because it allows implementation details to "creep in" to specifications.

In C++ and Eiffel, deferred or virtual classes are used to further separate specifications and implementations by placing the specification in a (virtual) superclass distinct from the concrete implementation (in a subclass). Disciplined use of this technique can remove the blurring of specifications and implementations inherent in class-based inheritance, but does not completely address the other problems limiting modular reasoning.

It appears that it might be possible to support modular reasoning within an inheritance framework, but only if conflicting uses of inheritance are decoupled by placing them in distinct hierarchies, and if the guarantees that must be made about information hiding along inheritance relations are completely spelled out. Without this approach, it appears that the scope of uses for inheritance must be significantly limited to make modular reasoning feasible.

3 Comparison

The central theme in Section 2 is separating the *mechanism* of inheritance from the *ends* it can be used to achieve. As in most cases where there is more than one possible goal, conflicts between goals arise. A language designer who wishes to ensure the correctness of his typing system may choose to restrict inheritance in certain ways [7, 4] to achieve this, which may in turn prevent some programmers from achieving other goals (certain kinds of code inheritance). Alternatively, a programmer may prefer to arrange his classes in a way that best exploits code sharing, which may interfere with a user's ability to cleanly understand and reason about the class hierarchy.

In [6], many of these same conflicts are raised. In the end, LaLonde concludes that (arbitrarily many) distinct inheritance hierarchies are needed to separate the different inter-class relationships that are being captured. However, in [6], it is the actual conflicts of different inheritance practices that drives the discussion—the goals that drive the practices emerge from the discussion of the conflicts, rather than the other way around.

As mentioned in Section 2, attempting to apply formal methods in an OO environment forces this problem into the open. Most often, the approach of researchers is to give a formal definition of inheritance that has "good" properties, and then restrict inheritance to such uses [8, 4, 9]. In some cases, authors have even recommended separating the inheritance hierarchy used for type checking (i.e., the "specification" inheritance hierarchy) from that used for code sharing [10]. Unfortunately, few, if any, of these approaches is based on the goals that inheritance is used to achieve—instead, they are often based on a single conception of a "good" inter-class relationship. Fortunately, all of these approaches tend to choose "good" inheritance relations that ensure modular reasoning is possible.

By focusing on "what" programmers and clients want to do with inheritance, instead of simply the "how" of the inheritance mechanism itself, one can step back from the problem and gain a slightly different perspective. It is clear that some of the "whats" require conflicting properties of the inheritance relationship itself. Perhaps as LaLonde and others have recommended, using distinct hierarchies (or lattices) for separate inter-class relationships is the way to go. Of course, by examining the support needed for each application of inheritance, one will find that there will be unique restrictions and requirements for these distinct lattices. Exploring what those requirements are, and how the lattices should be interrelated, is an area ripe for research.

References

[1] B. W. Weide, W. F. Ogden, and S. H. Zweben, "Reusable software components," in *Advances in Computers* (M. C. Yovits, ed.), Academic Press, 1991.

[2] B. W. Weide and J. E. Hollingsworth, "Scalability of reuse technology to large systems requires local certifiability," in *Proceedings of the Fifth Annual Workshop on Software Reuse*, October 1992.

[3] J. Hollingsworth, *Software Component Design-for-Reuse: A Language Independent Discipline Applied to Ada.* PhD thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1992.

[4] F. Weber, "Getting class correctness and system correctness equivalent," Tech. Rep. ProSt/91-4, Forschungszentrum Informatik an der Universität Karlsruhe, 1991.

[5] R. G. Fichman and C. F. Kemerer, "Object-oriented and conventional analysis and design methodologies," *Computer*, pp. 22–39, October 1992.

[6] W. R. LaLonde, "Designing families of data types using exemplars," *ACM Transactions on Programming Languages and Systems*, vol. 11, pp. 212–248, April 1989.

[7] B. Meyer, *Object-Oriented Software Construction.* New York, NY: Prentice Hall, 1988.

[8] J. C. Royer, "A new set interpretation of the inheritance relation and its checking," *OOPS Messenger*, vol. 3, pp. 22–40, July 1992.

[9] F. Weber, "Towards a discipline of class composition," Tech. Rep. ProSt/92-3, Forschungszentrum Informatik an der Universität Karlsruhe, 1992.

[10] P. S. Canning, W. R. Cook, W. L. Hill, and W. G. Olthoff, "Interfaces for strongly-typed object-oriented programming," in *OOPSLA'89 Proceedings*, ACM, October 1989.

[11] O. L. Madsen, B. Magnusson, and B. Møller-Pedersen, "Strong typing of object-oriented languages revisited," in *OOPSLA ECOOP '90 Proceedings*, ACM, October 1990.

Biography

Stephen H. Edwards is a doctoral student in the Department of Computer and Information Science at the Ohio State University. His research interests are in software engineering, the use of formal methods in programming languages, and information retrieval technology. The current focus of his work is on developing and refining a formal model of software components which explicitly addresses reuse concerns. Prior to entering the Ohio State University, Mr. Edwards was a research staff member at the Institute for Defense Analyses, where he worked on software reuse activities, simulation frameworks, active databases, and Ada programming issues. Mr. Edwards received his MS in computer science from Ohio State in 1992, and his BS in electrical engineering from Caltech in 1988.