On the Notion of Inheritance

ANTERO TAIVALSAARI

Nokia Research Center

One of the most intriguing—and at the same time most problematic—notions in object-oriented programming is *inheritance*. Inheritance is commonly regarded as the feature that distinguishes object-oriented programming from other modern programming paradigms, but researchers rarely agree on its meaning and usage. Yet inheritance is often hailed as a solution to many problems hampering software development, and many of the alleged benefits of object-oriented programming, such as improved conceptual modeling and reusability, are largely credited to it. This article aims at a comprehensive understanding of inheritance, examining its usage, surveying its varieties, and presenting a simple taxonomy of mechanisms that can be seen as underlying different inheritance models.

Categories and Subject Descriptors: D.1.5. [**Programming Techniques**]: Object-Oriented Programming; D.3.2. [**Programming Languages**]: Language Classifications *object-oriented languages*; D.3.3. [**Programming Languages**]: Language Constructs and Features

General Terms: Languages

Additional Key Words and Phrases: Delegation, incremental modification, inheritance, language constructs, object-oriented programming, programming languages

1. INTRODUCTION

A characteristic feature of object-oriented programming is *inheritance*. Inheritance is often regarded as the feature that distinguishes object-oriented programming from other modern programming paradigms, and many of the alleged benefits of object-oriented programming, such as improved conceptual modeling and reusability, are largely accredited to it. Despite its central role in current object-oriented systems, inheritance is still quite a controversial mechanism, and researchers tend to disagree on its meaning and usage. The only currently well-developed and widely accepted area seems to be the theory of inheritance in terms of denotational semantics [Cook 1989a; Cook and Palsberg 1989; Reddy 1988].

Despite the fact that much effort has been targeted on research into inheritance in the past years, it seems that inheritance is still often inadequately understood. Many studies of inheritance concentrate only on one specific viewpoint, such as type theory or conceptual modeling, and mostly ignore the other possible viewpoints. Depending on the viewpoint, inheritance is regarded either as a structuring or modeling mech-

Author's address: Nokia Research Center, P.O. Box 45, 00211 Helsinki; Finland; email: antero.taivalsaari@research.nokia.com.

anism for reasoning about programs, or a mechanism for code sharing and reuse. It seems that a more general, comprehensive view of inheritance is still missing. The main motivation for this article was the desire to reach a more thorough understanding of inheritance from different viewpoints.

This article provides a comprehensive introduction to inheritance, starting from its history and conceptual background, examining its intended and actual usage, and discussing its essence in light of the current knowledge. Furthermore, the varieties of inheritance are analyzed, and a simple taxonomy of inheritance mechanisms is presented. This article is a continuation to an earlier article on object-oriented programming [Taivalsaari 1993a], and part of a larger study of inheritance, published as the author's doctoral thesis [Taivalsaari 1993c].

2. INHERITANCE

If I have seen a little farther than others, it is because I have stood on the shoulders of giants.

—ISAAC NEWTON

2.1 Definitions

In general, to *inherit* is to receive properties or characteristics of another, normally as a result of some special relationship between the giver and the receiver [Danforth and Tomlinson 1988]. This broad definition of inheritance comes from the usage of the term in the real world, and at the first consideration it seems to have very little to do with computers. Nevertheless, in the context of programming, inheritance was introduced as early as in the end of the 1960s as a central feature of the programming language Simula [Dahl et al. 1968]. However, Simula's inheritance mechanism was originally known by a different name, *concatenation* [Dahl et al. 1972, pp. 202–204; Nygaard and Dahl 1978], and the intuitively more appealing term *inheritance* was invented some years later. Well-known currently used synonyms for inheritance in object-oriented systems are *subclassing, derivation* (in C++) and *prefixing* (in Simula and Beta); in some papers the term *subtyping* is also used in the same meaning [Halbert and O'Brien 1987], although more commonly that term is reserved for another purpose, as will be discussed in Section 2.2.2.

The basic idea of inheritance is simple. Inheritance allows new object definitions to be based upon existing ones; when a new kind of an object class is to be defined, only those properties that differ from the properties of the specified existing classes need to be declared explicitly, while the other properties are automatically extracted from the existing classes and included in the new class. Thus, inheritance is a facility for *differential*, or *incremental* program development. Formally, inheritance can be characterized as follows [Bracha and Cook 1990; Cook 1989a; Wegner and Zdonik 1988]:

$$R = P \oplus \Delta R.$$

In this maxim, R denotes a newly defined object or class, P denotes the properties inherited from an existing object or class, ΔR denotes the incrementally added new properties that differentiate R from P (the *delta* part), and $\oplus$ denotes an operation to somehow combine ΔR with the properties of P. As a result of this combination, R will contain all the properties of P, except that the incremental modification part ΔR may introduce properties that overlap with those of P so as to *redefine* or *defeat* (cancel) certain properties of P; thus, R may not always be fully compatible with P. Compatibility issues will be discussed in Section 2.2.2.

There are certain terms and notions pertaining to inheritance that will be used throughout the article. For instance, in the maxim above, P is known as R's *parent* or *immediate ancestor*; in class-based systems the corresponding term is *superclass*. Similarly, R is P's

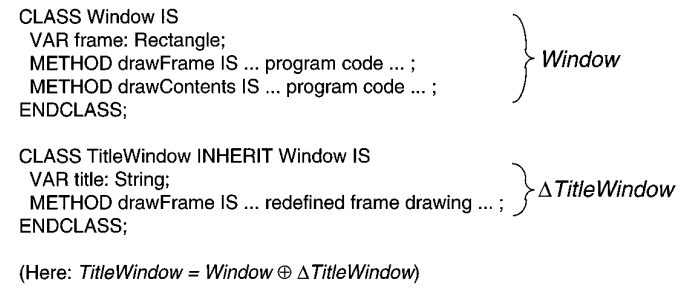

Figure 1. A simple example of inheritance.

child, immediate descendant, or in class-based systems its *subclass*. Inheritance relationships are transitive, so that a parent or superclass may be a child or subclass of another object or class. This implies that in addition to incrementally defined properties, an object will contain all the properties of its parents, parents' parents and so on. The terms *ancestor* and *descendant* are used in the obvious manner to denote the immediate and nonimmediate parents and children of a class.

Most modern object-oriented systems allow inheritance from several parents at the same time. Such inheritance is known as *multiple inheritance*, as opposed to *single inheritance* discussed above. As multiple inheritance offers considerably more possibilities for incremental modification than single inheritance, a generally accepted view is that a modern object-oriented language should support it, despite the fact that multiple inheritance also introduces many conceptual and technical intricacies. Issues pertaining to multiple inheritance will be discussed briefly in the following section. For different definitions of inheritance, refer to Cardelli [1984]; Cook [1989a]; Wegner [1987]; Wegner and Zdonik [1988].

As an example of inheritance, consider the following pseudocode definition (Figure 1). In the example, two classes, Window and TitleWindow, are defined. Class Window defines one variable, *frame*, and two operations (methods) *drawFrame* and *drawContents*. Class TitleWindow inherits class Window and adds a new variable called *title* plus a method that redefines frame drawing. The actual implementation of the methods is elided.

In its basic form, inheritance can be characterized formally as *record combination* [Bracha and Lindstrom 1992; Cardelli 1984; Cook 1989a]. An object or a class that inherits the properties of another is viewed as a record which is otherwise similar to its parent, but which has been extended with some additional properties. This record combination can take place in several different ways, however. For instance, in class-based object-oriented systems (see Section 3.1) the properties of an object are typically located physically in different places, and this tends to make the analysis more complicated. Furthermore, other issues such as early binding and encapsulation also encumber the analysis of inheritance. In general, inheritance is not an independent language feature, but it usually operates in tight interaction with other language mechanisms.

2.2 Conceptual View of Inheritance

To attain knowledge, add things every day; to obtain wisdom, remove things every day.

—LAO-TZU, *Tao Te Ching*

A programming language is a notation and, as such, serves to record and assist the development of human thought in a particular direction. For a notation to be effective, it must carry a mental load for the user and must have, among other

properties, economy and the ability to subordinate detail [Marcotty and Ledgard 1991]. By economy it is meant that a wide range of programs can be expressed using a relatively small vocabulary and simple grammatical rules. An important goal in achieving this is the *orthogonality* of language constructs. In other words, the language should provide no more than one way of expressing any action in the language, and each construct should be backed up by solid conceptual and practical reasons.

In order to motivate the use of inheritance in modern programming languages, let us now examine the relationship of object-oriented programming and conceptual modeling.

2.2.1 *Object-Oriented Programming and Conceptual Modeling*

> This structure of concepts is formally called a hierarchy and since ancient times has been a basic structure for all Western knowledge.
>
> —ROBERT M. PIRSIG, *Zen and the Art of Motorcycle Maintenance*

> Q: What's big and gray, has a trunk, and lives in the trees?
>
> A: An elephant. I lied about the trees.
>
> —R. J. BRACHMAN

Object-oriented programming originated from Simula, a programming language that was initially targeted for the simulation of real-world phenomena. Simulation is a task that requires a great deal of conceptual modeling skills, and therefore right from the beginning the developers of Simula emphasized the importance of a close correspondence between the program and problem domain. As noted in Madsen et al. [1990], the primary motivation leading to the introduction of the class concept in Simula, for example, was to model the concepts in the application domain. Similarly, the inheritance mechanism was initially introduced to represent certain kinds of modeling relationships, namely *conceptual specialization* [Madsen et al. 1990]. Let us now take a look at the basic conceptual modeling relationships in order to elucidate the role of inheritance as a modeling mechanism.

Generally speaking, conceptual modeling can be defined as the process of organizing our knowledge of an application domain into hierarchical rankings or orderings of abstractions, in order to obtain a better understanding of the phenomena in concern. The principles according to which this process takes place are usually referred to as the *abstraction principles, abstraction mechanisms* or *abstraction concepts* [Borgida et al. 1984; Mattos 1988]. One of the often mentioned benefits of object-oriented programming is that unlike other modern programming paradigms, it provides direct support for each of the most important abstraction principles: (1) classification/instantiation, (2) aggregation/decomposition, (3) generalization/specialization and (4) grouping/individualization.

Classification, which is usually considered the most important abstraction principle, is grouping like things together into *classes* or *categories* over which uniform conditions hold [Borgida et al. 1984]. Ideally, classes should share at least one such characteristic that the members of other classes do not have. Classification is an *intensional* abstraction principle that ought to be based ideally on such properties of substances that do not change in the course of time. The reverse operation of classification is *instantiation*, or *exemplification* [Knudsen and Madsen 1988]. Instantiation produces *instances*, entities which fulfill the intensional description of their class. Collectively the instances of a class form the *extension* of that class. Most object-oriented languages provide support for classification and instantiation by allowing the construction of classes and instances. In prototype-based object-oriented systems (Section 3.1) there are no classes, but the effect of instantiation can be simulated by copying concrete objects. For a more

extensive discussion on the notions of intension and extension, refer to Sowa [1984, pp. 10–11].

The second main abstraction principle is *aggregation*. By aggregation we refer to the principle of treating collections of concepts as single higher-level concepts: *aggregates* [Borgida 1984; Smith and Smith 1977a]. Aggregation describes things in terms of parts and wholes. A part is a part by virtue of being included in a larger whole. A part can also become a whole in itself, which can then be split into further parts. Therefore, aggregation hierarchies are sometimes also referred to as *part-whole hierarchies*. The reverse operation of aggregation is *decomposition*, which yields the individual components of an aggregate. Object-oriented languages typically support aggregation/decomposition by allowing the use of objects as *variables* in other objects. Variables can be used to hold other objects in order to construct more complex part-whole hierarchies. In some papers the term *composition* is used as a synonym for aggregation.

The third major abstraction principle, *generalization*, refers to the construction of concepts that cover a number of more special concepts sharing some similarities [Knudsen and Madsen 1988; Smith and Smith 1977b]. On the basis of one or more given classes, generalization produces the description of a more general class that captures the commonalities but suppresses some of the detailed differences in descriptions of the given classes [Borgida et al. 1984]. The converse operation of generalization is *specialization*. A concept C_s can be regarded as a specialization of another concept C if all phenomena belonging to the extension of the specialized concept C_s also belong to the extension of C [Pedersen 1989]. This implies that C and C_s are otherwise similar, but C_s may also possess some additional, more specific properties. Generalization and specialization complement classification in that they allow classes to be described naturally in terms of other classes [Mattos 1988]. In particular, specialization—allowing new concepts to be derived from less specific classes—seems intuitively as the natural high-level counterpart of inheritance, and therefore it has traditionally been assumed that inheritance and specialization are simply different views of the same thing. Although at the first glance this correspondence seems natural and aesthetically pleasant, recently it has been observed that the relationship between inheritance and conceptual specialization can be confusing. This issue will be discussed in more detail in the next section.

The fourth, perhaps the least obvious main abstraction principle is *grouping*, also known as *association, partitioning* or *cover aggregation* [Brodie 1983; Mattos 1988]. Frequently in conceptual modeling it becomes necessary to group objects together not because they have the same properties (classification), but because it is important to describe properties of a group of objects as a whole [Loomis et al. 1987; Mattos 1988]. Grouping addresses this need by allowing the representation of possibly nonhomogeneous collections of things related by their *extensional* rather than by their intensional properties. Being based on extensional properties, grouping bears some resemblance to the set theory of mathematics [Mattos 1988]. Object-oriented programming supports grouping by allowing the definition of arbitrary collection classes such as lists, sets, bags and dictionaries. The opposite of grouping is *individualization*, which yields individual members of a collection; the term individualization is, however, not very well-established.

The abstraction principles have been illustrated in Figure 2, which gives an example of each principle applied to the same application domain, namely *Car*. Furthermore, the list below summarizes the main characteristics of each principle (adapted from Smith and Smith [1980]).

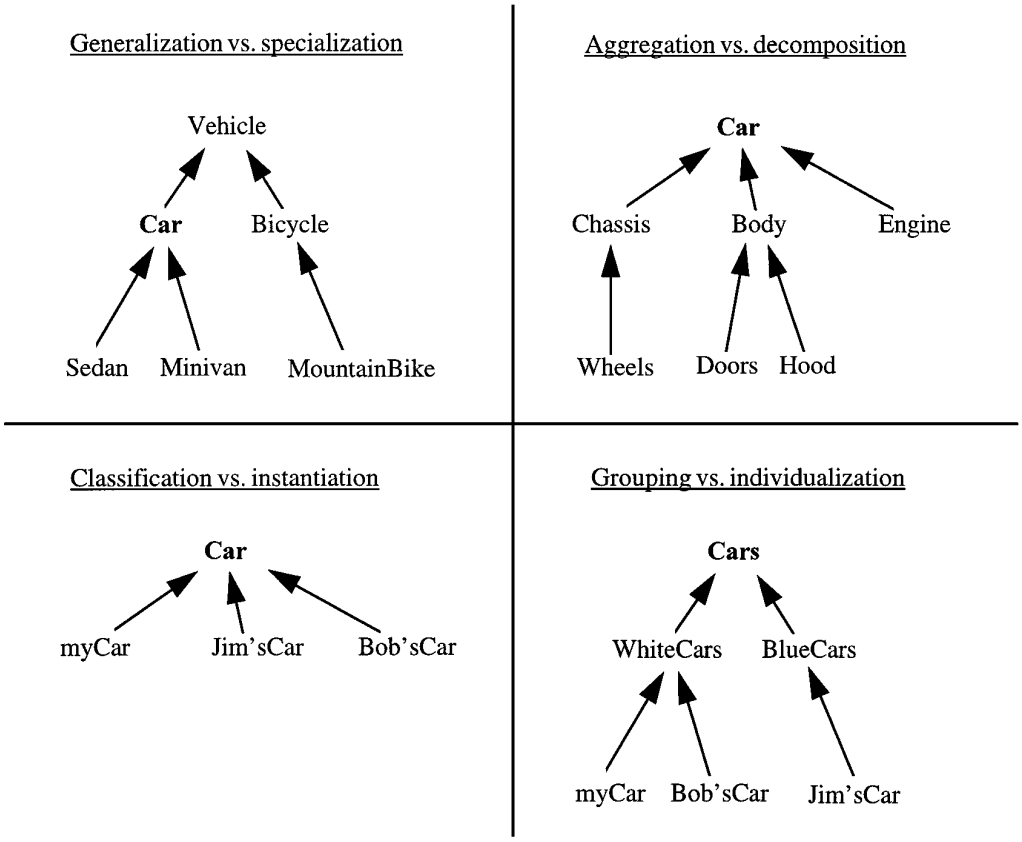

Figure 2. Abstraction principles.

—*Classification* suppresses details of instances and emphasizes properties of a class as a whole.
—*Generalization* suppresses the differences between categories and emphasizes common properties.
—*Aggregation* suppresses details of components and emphasizes details of the relationship as a whole.
—*Grouping* suppresses details of a group of objects and emphasizes the grouping of those objects together.

For further information on abstraction principles, the reader is referred to Borgida [1984]; Knudsen [1988]; Loomis et al. [1987]; Madsen and Møller-Pedersen [1988] and Mattos [1988].

2.2.2 *Inheritance, Subtyping and Specialization*

> The differentiae of genera which are different and not subordinate one to the other are themselves different in kind. For example, animal and knowledge: footed, winged, aquatic, two-footed, are differentiae of animal, but none of these is a differentia of knowledge; one sort of knowledge does not differ from another by being two-footed.
>
> —ARISTOTLE, *Categories §3*

The classical view of inheritance in object-oriented programming is that it is a hierarchical structuring mechanism intended for conceptual specialization. Indeed, language constructs for supporting generalization/specialization are

often mentioned as the main characteristic of a programming language supporting object-orientation [Madsen and Møller-Pedersen 1988]. Correspondingly, object-oriented programming has sometimes been characterized as "programming with taxonomically organized data" [Cardelli 1984].

Recently it has however been observed that the correspondence between inheritance and specialization is actually much more intricate than has previously been assumed [America 1987; Wegner and Zdonik 1988; Zdonik 1986]. Most of the problems in this respect arise from the fact that object-oriented systems do not typically provide any guarantees in that inheritance really is used for conceptual specialization. For instance, the redefined operations in a subclass do not usually have to bear any semantic relationship to the replaced operations in the superclass; the only semantic tie is that they share the same names [Zdonik 1986]. In general, if inheritance is implemented in the conventional fashion, allowing unlimited addition, redefinition and cancellation of properties in descendants, there is virtually nothing to ensure that the conceptual correspondence between parents and their children really prevails. Consequently, abstractions built using inheritance rarely are true conceptual specializations of their parents.

Some researchers have tried to guarantee the conceptual correspondence between parents and children by establishing certain *compatibility rules*. For instance, Wegner has identified four different levels of compatibility between classes and subclasses [Wegner 1990; Wegner and Zdonik 1988]. The weakest of these, *cancellation*, allows the operations of the class to be freely redefined and even cancelled (removed) in a subclass. The second level, *name compatibility* allows the operations to be redefined, but requires the subclass to preserve the same set of names (i.e., no properties may be removed). The third level, *signature compatibility* requires full syntactic (interface) compatibility between classes and their subclasses. The fourth level, *behavior compatibility* assumes full behavioral compatibility between classes and their subclasses. This implies that subclasses may not change the behavior of their superclasses in any radical way. The first three forms of compatibility, being based on mere syntactic aspects of class definitions, are relatively easy to guarantee, and inheritance mechanisms based on them are generally referred to as *nonstrict inheritance* [Wegner 1987; Wegner and Zdonik 1988]. Ensuring full behavior compatibility, however, turns out to be a much more difficult task. The term *strict inheritance* is often used to refer to behaviorally compatible forms of inheritance [Wegner 1987; Wegner and Zdonik 1988].

While strict inheritance at the first consideration seems to be the most desirable form of inheritance, there are several other reasonable ways to use inheritance that necessitate nonstrict inheritance. These alternative uses of inheritance will be discussed in detail in the next section. In fact, it has been argued that strict inheritance, i.e., the use of inheritance for conceptual specialization, is of limited utility in the evolutionary development of complex systems [Wegner 1987]. After all, strict inheritance restricts the use of inheritance to the refinement of existing abstractions only, and prohibits the incremental modification of those abstractions in more creative ways. When using strict inheritance only, the addition of anything truly *new* to the system has to be accomplished by constructing those abstractions from scratch. In general, it seems that nonstrict inheritance can increase the expressive power of object-oriented systems. However, at the same time it decreases structural clarity because so much less can be inferred about the properties of descendants [Wegner 1987].

Another problematic issue in equating inheritance with specialization is *multiple inheritance*. An often cited controversial example of multiple inheri-

tance can be found in a book by Meyer [1988]. In his book, Meyer defines a class Fixed_Stack by inheriting two previously defined classes Stack and Array [Meyer 1988, pp. 241–242]. This is highly questionable because inheriting these classes implies that Fixed_Stack, in addition to being a specialization of Stack, would also be a specialization of Array. From the conceptual viewpoint, this is incorrect since many operations on arrays, such as the indexed element access operations, are not generally applicable to stacks. The correct solution in this case would be to inherit class Stack only, and use class Array as a component. Similar abuses of multiple inheritance are surprisingly common in the literature. In general, since multiple inheritance always results in a combination of existing abstractions, it tends to be difficult to regard it as a form of conceptual specialization. Rather, multiple inheritance seems to have more in common with aggregation. As aptly expressed by Alan Snyder in a panel discussion at OOPSLA'87: "multiple inheritance is good but there is no good way to do it." For further information on using multiple inheritance refer to Ducournau and Habib [1987], Knudsen [1988] and Snyder [1991].

Based on the observations above it should be obvious that equating inheritance with conceptual specialization poses several problems. Generally speaking, there seems to be a discrepancy between inheritance as a language mechanism and inheritance as a facility for conceptual modeling. These two roles of inheritance can be characterized as follows [Korson and McGregor 1990]:

(1) As part of the high-level program design phase, inheritance serves as a means of modeling generalization/specialization relationships.
(2) In the low-level implementation phase, inheritance supports the reuse of existing classes as the basis for the definition of new classes.

One of the first researchers to emphasize this distinction was Brachman [1983] who investigated the role of *is-a* relationships in semantic networks. Brachman suggested that inheritance should be treated only as an implementational issue that bears no direct correspondence to conceptual modeling. Later papers to study the relationship of inheritance and specialization in the context of object-oriented programming are America [1987; 1991], Cook et al. [1990], Palsberg and Schwatzbach [1991], Porter [1992] and Raj and Levy [1989]. A common argument in these papers is that in object-oriented systems a clear distinction ought to be made between two important concepts: inheritance and subtyping. *Inheritance* is a more low-level mechanism by which objects or classes can share behavior and data. *Subtyping*, on the other hand, expresses conceptual specialization [America 1991]. Following this dichotomy, inheritance is a mechanism that is suited but not necessarily limited to specialization (see Figure 3). In this context inheritance is often also referred to more specifically as *implementation inheritance, representation inheritance* or in class-based systems *subclassing*. Commonly used synonyms for subtyping are *specification inheritance* or *interface inheritance* [America 1987, 1991; Madsen et al. 1990]. Some papers also use the terms *syntactic* and *semantic inheritance* to distinguish between inheritance of implementation and specification, respectively [Hartmann et al. 1992].

It is widely argued that using inheritance for specialization is both theoretically and practically valuable, while using it for mere implementation purposes is likely to cause difficulties and reflects poor understanding of the purpose of inheritance. Therefore, following the Aristotlean tradition of dividing things into *essential* and *accidental*, the use of inheritance for specialization is often characterized as *essential* use of inheritance, whereas the other uses are considered more or less *accidental* or *inci-*

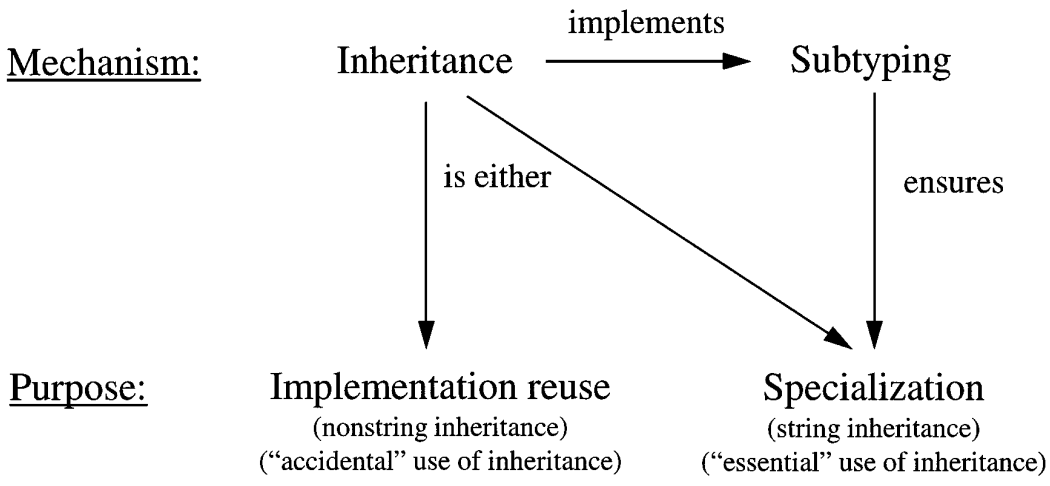

Figure 3. Inheritance versus subtyping (traditional view).

dental [Sakkinen 1989]. However, as already mentioned above and as will be discussed further in next section, there are other uses of inheritance beyond specialization that can be considered well-justified and thus essential, too.

Although the reasons for distinguishing between inheritance and subtyping are widely recognized and accepted, so far this distinction has been incorporated only in a few programming languages. Some languages making the distinction are POOL-I [America 1991] and Typed Smalltalk [Graver and Johnson 1990; Johnson 1986]. In these languages subtyping serves as a higher-level relation between types, whereas inheritance operates at the level of classes. In most other (strongly-typed) object-oriented languages—including C++, Eiffel, Trellis/Owl [Schaffert et al. 1986] and Simula—types are, however, equated with classes, and inheritance is basically restricted to satisfy the requirements of subtyping [Cook et al. 1990]. As observed in several papers [Cook 1989b; Palsberg and Schwartzbach 1991], this can lead to problems in type checking. These type checking issues are, however, beyond the scope of this article.

A drawback of the suggested separation between inheritance and subtyping is the extra complexity that it places on the implementation and use of languages supporting such separation. Furthermore, it has been argued that not even this distinction is enough if one wants to be specific about the orthogonality of language constructs. Similarly as there exists a distinction between inheritance and subtyping, there is a significant difference between subtyping and specialization. LaLonde [1989] and LaLonde and Pugh [1991] have proposed the following definitions for the three concepts:

—*Subclassing* is an implementation mechanism for sharing code and representation.
—*Subtyping* is a substitutability relationship: an instance of a subtype can stand in for an instance of its supertype.
—*Is-a* is a conceptual specialization relationship: it describes one kind of object as a special kind of another.

Subclassing, subtyping, and specialization are all important for different reasons. Subclassing supports reusability for the class library implementor:

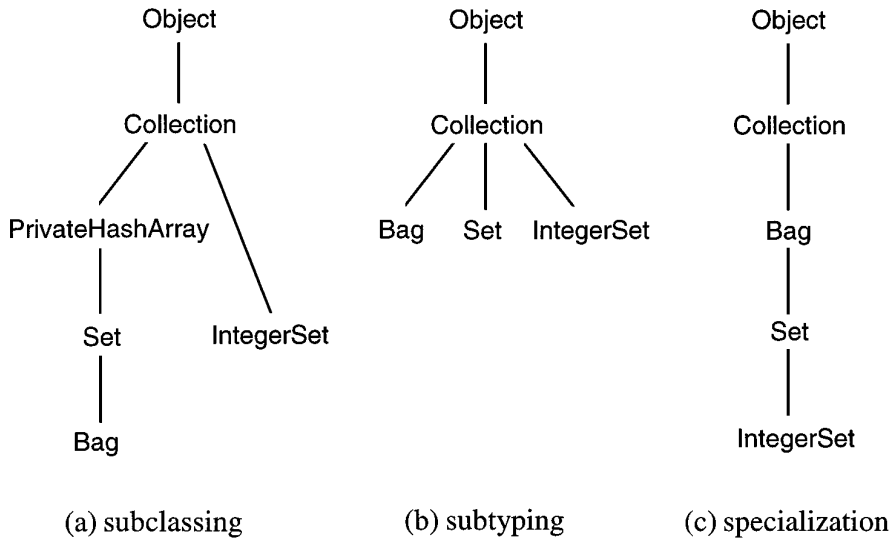

Figure 4. Subclassing, subtyping, and specialization.

new kinds of classes can be defined by leveraging off existing ones. Subtyping supports reusability for the class library user: to get maximum reusability, we need to know which classes can be substituted with which other classes. Specialization (is-a) relationships, in turn, are important for understanding the logical relationships between the concepts; in this sense, specialization is important for the class library designer. Figure 4 illustrates the differences between subclassing, subtyping, and specialization [LaLonde and Pugh 1991].

Each of the concepts in Figure 4 can be refined further. For instance, as observed by Brachman [1983; 1985], there are several kinds of is-a relationships. Varieties of subtyping have been examined by Wegner [1987], who distinguishes the following "subtypes" of subtyping:

—Subset subtyping. Int[1..10] is a subset subtype of Int.

—Isomorphic copy subtyping. Int is an isomorphic copy subtype of Real.

—Object-oriented subtyping. Student is an object-oriented subtype of Person.

The varieties of inheritance and subclassing will be examined in the following subsections.

2.2.3 *Use of Inheritance in Practice*

Like the ski resort full of girls hunting for husbands and husbands hunting for girls the situation is not as symmetrical as it might seem.

—ALAN MACKAY

Inheritance is a language mechanism that allows new object definitions to be based on existing ones. A new class inherits the properties of its parents, and may introduce new properties that extend, modify or defeat its inherited properties. In general, most object-oriented languages allow the inherited properties to be reimplemented, renamed, removed, duplicated, have their visibility changed, or undergo almost any other kind of transformation as they are mapped from parents to descendants. Therefore, it is apparent that inheritance can be used for purposes that go beyond specialization. In fact, the use of inheritance for conceptual specialization seems to be an ideal that is rarely realized. Most commercially available class libraries, such as the standard classes of Smalltalk-80 or commercial C++ class libraries, are permeated with examples of using inheritance in other ways. This section examines those uses of inheritance that are common in real object-oriented systems, but which do not

obey the semantic requirements of conceptual specialization.

Inheritance for implementation refers to situations in which inheritance is used not because the abstractions to be inherited are ideal from the conceptual point of view, but simply because they happen to contain appropriate properties for the new abstraction that is under construction. In implementation inheritance, theoretical and conceptual issues such as behavior compatibility are ignored, and pragmatic reasons such as potential savings in coding effort, storage space or execution speed are emphasized instead. As obvious, using inheritance for mere implementation reasons is often questionable, and it has been criticized in the literature [America 1987; Sakkinen 1989]. Some of the most apparent variations of implementation inheritance are (adapted from Rumbaugh et al. [1991, p. 64]):

—cancellation,
—optimization, and
—convenience.

In *cancellation*, the descendant restricts the inherited behavior by explicitly making certain inherited properties unavailable, or by tightening the parameter type requirements. Cancellations are common in the Smalltalk-80 collection class hierarchy that will be discussed below. In *optimization*, the descendant takes advantage of some specific implementation information to improve the code for its operations. For example, a superclass Set could have an operation to find the maximum element implemented as a sequential search; the subclass SortedSet could provide a more efficient implementation for the same operation since the elements are already sorted. In implementation inheritance for *convenience*, the new class is made a subclass of the existing class simply because the existing class seems to provide what is desired.

The archetypical example of the use of implementation inheritance is the *Collection* class hierarchy of Smalltalk-80 [Goldberg and Robson 1989, pp. 144–169; Graver and Johnson 1990; LaLonde 1989; LaLonde et al. 1986; Wegner 1990]. The Collection class hierarchy consists of more than 20 classes that represent various kinds of container objects such as Sets, Bags (sets that allow duplicate elements), Dictionaries (mappings from keys to values), and Strings. Many of these classes have been made subclasses of others simply because the superclass happens to conveniently provide the right functionality. For example, class *Dictionary* implements a keyed lookup table as a hash table of ⟨*key, value*⟩ pairs. The hash table implementation is inherited from class *Set* (which in turn is a subclass of Collection), but applications using Sets would behave quite differently if given Dictionaries instead. In other words, class Dictionary is behaviorally incompatible with its superclass. Besides convenience, the Collection class hierarchy provides examples of cancellations and optimizations as well. For instance, since Dictionary elements can only be removed by their key values, class Dictionary overrides the *remove:ifAbsent* method inherited from class Set with an error message method. Similarly, the iteration method *do:* is redefined in almost every collection class to optimize it for each particular kind of abstraction. Despite its idiosyncrasies, the Collection class hierarchy of Smalltalk does have its advantages. Generally it seems that implementing the same functionality in a conceptually more elegant fashion would necessitate a more complex and more memory-consuming class hierarchy. Cook has argued to the contrary, however, and has presented an alternative Collection class hierarchy based on the conceptual relationships of the classes [Cook 1992].

Inheritance for combination refers to situations in which inheritance is used for combining existing abstractions with multiple inheritance [Halbert and O'Brien 1987]. As illustrated by the Fixed_Stack example discussed in Sec-

tion 2.2.2, such use of inheritance is error-prone, and in many cases it would be appropriate to use single inheritance and aggregation instead. Nevertheless, two subforms of combination inheritance can be identified. First, multiple inheritance is often used for combining abstractions of *equal importance*. The classical example of such use of multiple inheritance is to inherit two classes, Teacher and Student, to form a class TeachingAssistant [Halbert and O'Brien 1987; Sciore 1989]. In practice the combination of conceptually equal abstractions using multiple inheritance can be quite laborious, however, because such abstractions tend to contain a large number of overlapping properties that have to be dealt with in the subclass [Chambers et al. 1991]; sometimes this may necessitate considerable modifications. In many cases such situations could also be expressed more naturally using *roles* [Pernici 1990]. For instance, from the conceptual viewpoint, Teacher and Student are not really proper subclasses of Person, but they should rather be seen as different roles that persons are capable of playing at different times. The second form of inheritance for combination is *mixin-based inheritance*, or *mixin inheritance* [Bracha and Cook 1990; Hendler 1986], which has recently received considerable attention. Mixin inheritance originated from the programming languages Flavors [Moon 1986] and Oaklisp [Lang and Pearlmutter 1986], and has subsequently been utilized in the Common Lisp Object System (CLOS) [DeMichiel and Gabriel 1987; Keene 1989]. Mixin inheritance will be discussed in more detail later in Section 3.7.

Inheritance for inclusion. One alternative use of inheritance is also *inheritance for inclusion*. Many class-based object-oriented languages (e.g., Smalltalk) do not provide a separate module mechanism, and thus classes are sometimes needed for simulating modules or function libraries. For instance, to create a library of trigonometric functions in an object-oriented language without a module mechanism, the only viable way is to place those functions in a separate class. This library class can then be used in two different ways. The first possibility is to "instantiate" the library, which has the negative consequence that the functions will have to be accessed indirectly via an intervening variable. The second way is to use inheritance to simulate the *import* mechanism of many abstract data type (ADT) languages such as Modula-2 [Wirth 1985]. Using the library as a superclass, the functions in the library will be included in the scope of the subclass and can thus be referred to directly without extra effort. This is practical, but is yet another example of using inheritance for other reasons than conceptual specialization.

Other uses of inheritance. In addition to the above-mentioned uses of inheritance, some researchers have identified several rarer forms of inheritance. One such example is *inheritance for generalization* that can be seen as the opposite of inheritance for specialization [Halbert and O'Brien 1987; Pedersen 1989]. Like specialization, inheritance for generalization is based on conceptual reasons, but the actual direction of conceptual relationships has been reversed. In other words, rather than specializing the behavior of its parent, a descendant is designed as a generalization of its parent. Inheritance for generalization has been investigated in detail by Pedersen [1989], who considers such use of inheritance valuable when changes to the existing classification hierarchy are needed and when those changes should be accomplished incrementally rather than by modifying the existing inheritance hierarchy. Another advantage of generalization is that in some situations it may be easier to implement abstractions as generalizations of more special concepts than vice versa [Halbert and O'Brien 1987; Pedersen 1989]. For example, if a system already incorporates a *deque* ab-

straction, then the implementation of *stack* by generalization is easier than by constructing it from more general abstractions [Snyder 1986a].

In conclusion, there is no single answer as to what constitutes proper use of inheritance. Although conceptual specialization was originally regarded as the only legitimate reason for using inheritance, in practice inheritance is commonly used for other quite reasonable purposes. Consequently, a major theme in the object-oriented programming research in the past years has been the clarification of the role of inheritance. Despite these efforts, researchers still have differing opinions on inheritance, and a lot of work in this area remains to be done.

2.3 The Essence of Inheritance

On the basis of the preceding discussion it appears that the analogy between inheritance and conceptual specialization is a lot weaker than has often been claimed. In order to reach a better understanding of what the essence of inheritance in object-oriented programming really is, let us now set the conceptual modeling viewpoint aside for a while, and approach the issue from a more pragmatic viewpoint. The section starts by recognizing certain inherent problems in traditional programming methodologies and by showing how inheritance can address these problems. A more theoretical discussion of the essence of inheritance will then follow.

2.3.1 *Inheritance as an Incremental Modification Mechanism*

> Ours is a world of things—but of changing things not quiescent ones.
>
> —MARIO BUNGE, *The Furniture of the World*

> There are six kinds of change: generation, destruction, increase, diminution, alteration, change of place.
>
> —ARISTOTLE, *Categories §14*

A well-known fact in software development is that if a piece of software is useful, it will have to be changed. Therefore, one of the most important qualities of software is *malleability*, or modifiability; the easier the software system is to change, the more likely it is that it will fulfill the requirements it was built to satisfy, and the more easily it will be able to keep up with the evolving needs of the users. Traditional programming methodologies do not take this evolutionary nature of software into account very well. The principles of abstract data type (ADT) programming, such as locality of information, modularity and representation independence [Liskov 1987], provide some help in this respect, but are unable to solve some of the most central issues.

One of the main problems plaguing software development is that program modification, in the traditional fashion, is a *process of destructive* or *radical change* [Cook 1989a; Cook and Palsberg 1989]. In many situations, even small modifications to some parts of the system will have widespread effects on the other parts. The larger the software system is, the greater is the likelihood of undesired, inadvertent side-effects. Yet many of the changes in software systems do not really necessitate destructive modification. If the modifications, replacements, or removals of existing parts of the program were explicitly avoided, and an incremental programming style were adopted, these unforeseen effects could be avoided.

To properly understand the problems with destructive modification, let us consider an example in which we are building a simple graphical windowing system. In Figure 5, we define a class Window whose instances are assumed to be windows on the screen. To avoid extra verbosity and complexity, only a small subset of the variables and operations needed in implementing a real window system have been included. It is assumed that methods *drawFrame* and *drawContents* display the frame and the contents of a window on the screen at the location specified by the variable *rect*. Method *refresh* invokes both these

```
CLASS Window IS
    VAR rect: Rectangle;
    VAR contents: Printable;

    METHOD drawFrame IS          ... print the frame of the window ... ;
    METHOD drawContents IS       ... print the contents inside the frame ... ;
    METHOD refresh IS  self.drawFrame  self.drawContents ;
ENDCLASS;
```

Figure 5. Class Window.

methods in order to update both the frame and the contents. Since the example will be used to illustrate further aspects of inheritance in the end of this section, references to methods *drawFrame* and *drawContents* have been denoted using a special pseudovariable *self*. At this point these self-references can simply be ignored.

Suppose we would later like to modify our window system to support windows that are capable of displaying a title (name) on their frame. In conventional ADT systems this can be achieved in two basic ways. The first way is to edit the source code of the existing class Window to support both plain windows and windows with a title. This could be accomplished by adding a new variable to hold the title of the window, and another variable to store information of whether the current window is a plain one or one with a title. Additionally, a case statement should be added to method *drawFrame* to determine whether or not the title should be printed on the frame.

The second way to achieve the same effect is to copy the source code of the existing Window class and edit the copy to form a completely new TitleWindow class—this approach can be called the *"copy-and-modify"* scheme. Neither of these approaches is really satisfactory. In a large system direct modification of an existing component is likely to cause inconsistencies with the other components that happen to refer to the properties of the modified component. The copy-and-modify scheme, on the other hand, is uneconomical, because usually the changes to the existing components are relatively small in proportion to the overall size of the component. Furthermore, textual copying loses the relationship between the original and copied component, and this is likely to cause maintenance problems later on. In general, there are two arguments against tampering with existing source code [Krueger 1992].

—Editing source code forces the software developer to work at a low level of abstraction. The effort required to understand and modify the low-level details of a component offsets a significant amount of the effort saved in reusing the component.

—Editing source code may invalidate the correctness of the original component. This eliminates the ability to amortize validation and verification costs over the life of a reusable component.

Inheritance addresses the above mentioned problems by promoting incremental definition. By adding new properties, a new class can extend, modify and defeat the properties inherited from its parents, but the original class still remains the same. However, note that the avoidance of destructive changes itself is not enough. So as to avoid unnecessary redefinitions, some extra linguistic support is needed. Consider the definition in Figure 6.

The class definition in Figure 6 is an attempt to define a new class TitleWindow as an extension of the previously defined class Window. Since class Win-

```
CLASS TitleWindow INHERIT Window IS
    VAR title: String;
    METHOD drawFrame IS ... print the frame and the title on the frame ... ;
ENDCLASS;
```

Figure 6. Class TitleWindow.

dow implements the basic functionality of windows, it seems apparent that in order to create windows with a title on their frame, the only thing that needs to be redefined is the previously defined operation *drawFrame*. However, a problem with this approach is that after the redefinition of *drawFrame*, all those operations in superclass Window that happen to refer to the original operation *drawFrame*, such as *refresh*, are invalid. When applied to instances of TitleWindow, method *refresh* will still invoke the original method *drawFrame* of class Window, thus producing windows *without* a title on their frame.

In general, it is common for class definitions to contain a large number of operations that refer to the other operations in the same definition. These interdependencies between classes make incremental modification problematic; if one operation is redefined in a descendant, all the other operations referring to that operation will also have to be redefined. This is inconvenient and against the idea of incremental modification. For this reason, an additional language mechanism, *late binding*, is needed. Late binding allows references to the properties of objects to be postponed until the program is actually run, thus making operations *polymorphic*, i.e., able to take on different meanings at different times. By virtue of late binding, references to other properties from within an operation do not have to be statically fixed, but may invoke different properties depending on the context from which the operation is actually invoked. This dynamic context is determined by *self-reference*, a pseudovariable that denotes the object whose operation is currently being executed. In the window system example above, for instance, the use of late-bound self-reference causes the behavior of operation *refresh* to vary dynamically depending on which kind of a window it is currently acting upon, thereby ensuring that the correct *drawFrame* operation can be invoked.

What do we learn from all this? First, it is apparent that many modifications needed in software systems could be performed incrementally, yet conventional systems enforce these modifications to be carried out in a destructive manner. Second, late binding is a prerequisite for implementing a system that allows true incremental modification. Without late binding, there is no assurance that newly defined components will work correctly when inherited operations are applied to them. In general, inheritance can be defined as follows (adapted from Cook [1989a] and Cook and Palsberg [1989]):

Inheritance is an incremental modification mechanism in the presence of a late-bound self-reference.

A detailed description of the semantics of late binding and self-reference will be given in the next section.

2.3.2 *Late Binding and Self-Reference*

To thine own self be true.

—WILLIAM SHAKESPEARE, *Hamlet*

Inheritance is an incremental modification mechanism. By virtue of inheritance, a new object class can extend itself by internalizing properties of ancestors as though they were its own [Wegner 1987]. Late binding, as implied above, allows this internalization to be performed in a manner that minimizes

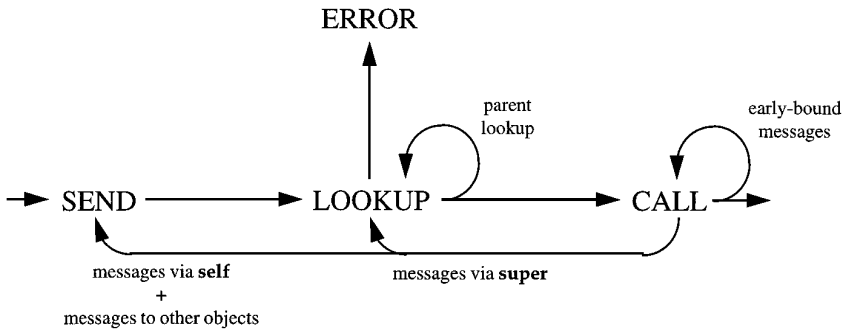

Figure 7. Semantics of message passing with late binding.

the need for physical code duplication, by letting the same operations invoke different properties depending on the context from which these operations are invoked, thereby promoting sharing of code. This section takes a more theoretical look at how incremental modification is actually achieved, and gives a general description of the semantics of late-bound message sending. Since we are not yet focusing on any particular model of inheritance, the description will be kept independent of implementation aspects.

Theoretically, inheritance can be characterized as the operation that derives new abstractions from previously defined ones, denoted formally $R = P \oplus \Delta R$, or in the case of multiple inheritance: $R = P_1 \oplus P_2 \oplus \ldots \oplus P_n \oplus \Delta R$. When applied transitively, the operation results in abstraction hierarchies that take the form of *tree* (in the case of single inheritance), or more generally: *directed acyclic graph* (DAG; see America [1987] and Knudsen [1988]). Each node in an inheritance DAG represents a collection of properties that has been incrementally added on top of the inherited properties at some particular level, corresponding to a delta part in some "$R = P \oplus \Delta R$" declaration.

Note that although in object-oriented systems inheritance is typically possible only between classes, there is usually an isomorphic correspondence between classes and their instances, so that instances can also be viewed as inheritance DAGs if so desired. The mapping between classes and their instances may not always be totally one-to-one, however, because *repeated inheritance* (e.g., a class having two superclasses that have a common superclass; also known as *fork-join inheritance* [Sakkinen 1989], or *diamond inheritance* [Bracha 1992, pp. 24–26]) may cause some instance variables to be duplicated. Bear in mind that an essential restriction on all inheritance hierarchies is that they may not contain *cycles*. This restriction is obvious as it would be nonsensical for abstractions to inherit their own properties.

If objects are viewed as directed acyclic graphs, the semantics of inheritance can be described quite easily and independently of any particular model of inheritance. Whenever a late-bound message is sent to an object, the following kind of an algorithm will take control. The algorithm is illustrated in Figure 7 (adapted from Cook [1989a] and Cook and Palsberg [1989]). Parts of the figure will be discussed also in the next subsection.

(1) First, self-reference is set to refer to the receiver of the message. Previous value of self-reference is saved so that it can be restored later. In Figure 7, this phase is called *SEND*.
(2) The given message selector is then matched against the properties of the receiver by traversing its inheritance DAG node by node. The ac-

tual traversal order depends on the form of inheritance being used (Section 3.3). In Figure 7, this phase is called *LOOKUP*.

(3) If a matching property is found in some node, the matching property is invoked, e.g., using a normal procedure call mechanism. In Figure 7, this phase is termed *CALL*. After execution, the previous value of self-reference is restored.

(4) In case no matching property can be found in the whole DAG, a binding error will be raised. In Figure 7, this phase is called *ERROR*.

As a result of successful late binding, the matching operation of the object will be invoked. During the execution of the operation, the self-reference will remain denoting the root of the current inheritance DAG, so that subsequent message sends via the self-reference from within that operation can access the other properties of the object. By virtue of late binding, operations inherited from parents can invoke such properties that were not necessarily even implemented at the time when the parent was defined.

Note that in actual implementations of object-oriented languages, the above described algorithm is usually replaced with much more efficient strategies. Simple inline caching techniques [Deutsh and Schiffman 1984], for instance, eliminate the need to perform the actual lookup in 95 percent of the situations. Various optimization techniques have been surveyed in Driesen et al. [1995].

To the programmer, the self-reference typically appears in the form of a special pseudovariable, *self*, that can be used explicitly to inform the system that late binding via self-reference is requested. At the language level, *self* is treated as a free variable whose textual occurrences in a program are bound to a particular object only at the time they are executed. By the term *pseudovariable* it is implied that *self* is system-maintained and cannot be modified by the programmer.

In different languages different names for the self-reference are used. Smalltalk and its derivatives use *self*, whereas in Simula, Beta [Kristensen et al. 1983; Madsen and Møller-Pedersen 1989] and C++ the corresponding language construct is known as *this*. Eiffel provides a pseudovariable *current* for the same purpose [Meyer 1988, p. 79 and 177]. In some object-oriented languages self-references are implicit. In Eiffel, for instance, *current* is usually elided, and thus late-bound operation invocations look syntactically identical to ordinary procedure calls in conventional programming languages. The C++ programmer may explicitly decide whether or not to use *this* in his member function calls.

2.3.3 *Accessing Overridden Properties*. Besides *self*, object-oriented languages typically provide other pseudovariables that contribute to the incremental definition of programs. The most common of these is *super* [Goldberg and Robson 1989, pp. 63–66] that is used for accessing those inherited properties that have been redefined in subclasses. When an operation sends a message via *super*, the lookup is started from the immediate parent of the node possessing that operation. This is different from normal message lookup in which the lookup is always started from the node denoted by the self-reference. When using *super*, the self-reference is left untouched, so that subsequent messages via *self* from the redefined operations will be able to access the more recently defined properties.

In order to illustrate the behavior of *super*, let us take the class TitleWindow defined in Section 2.3.1 into consideration again. In that example we redefined the operation *drawFrame* in order to construct windows with a title on their frame. In such a situation it would apparently be a great help if we could somehow utilize the previously defined *drawFrame* operation of superclass

```
CLASS TitleWindow INHERIT Window IS
    VAR title: String;
    METHOD drawFrame IS
        super.drawFrame          \ First, invoke the earlier drawFrame method
        ... new code ... ;       \ Then, print the title on the frame
ENDCLASS;
```

Figure 8. Class TitleWindow using super-reference.

Window. However, since the purpose of class TitleWindow is primarily to describe the title extension, it should know as little as possible about the actual frame drawing process. In order to avoid code duplication, the new *drawFrame* operation for TitleWindow should apparently first invoke the earlier *drawFrame* operation, thus drawing the plain border, then calculate the correct position for the title, and print the title *on* the frame. This is exactly what can be accomplished using the super-reference (see Figure 8). Without *super*, the whole frame printing algorithm would have to be reimplemented in the subclass. Thus, the super-reference is an invaluable tool for reducing the need for code reimplementation and contributing to incremental modification.

Several variations of super-reference exist. Whereas *super* is characteristic of languages modeled after Smalltalk, in CLOS the corresponding language construct is known as *call-next-method* [Keene 1989, p. 233]. Unlike *super*, CLOS's *call-next-method* does not require any message selector as parameter, but uses the previous selector name by default. In this sense CLOS's *call-next-method* is analogous to Beta's *inner* construct [Kristensen et al. 1983], although otherwise Beta's inheritance scheme is radically different (see Section 3.3). C++ gives the programmer the ability to access redefined properties by using superclass names as qualifiers [Ellis and Stroustrup 1990, p. 390]. Eiffel does not provide any explicit super-reference, but allows the same effect to be simulated by *renaming* the inherited properties in the subclass [Meyer 1988, pp. 246–250]. Some languages, such as Simula and Deltatalk [Borning and O'Shea 1987] do not provide any facilities for accessing redefined operations, and this may limit incremental modification in many situations. In Self [Ungar and Smith 1987], the super-reference was originally called *super*, but later renamed as *resend* [Chambers et al. 1991].

In general, late binding, self-reference and super-reference are salient elements of all inheritance mechanisms. Late binding and self-reference allow the programmer to perform "surgery" which can change object behavior without physically reaching inside any object. Super-reference supplements these facilities by allowing access to redefined properties, thereby reducing the need for code duplication. When used properly, together these facilities allow the properties of objects to be reused without any textual copying or editing, which is a major advantage over other programming styles and techniques. In this sense, inheritance is a truly novel and fundamental mechanism for constructing programs [Cook 1989a].

2.3.4 *Inheritance as a Specificational Structuring tool*

> Inside every large program there is a small program trying to get out.
>
> —C.A.R. HOARE

Incremental modification is not a panacea. Not all modifications to programs can be performed incrementally. Nevertheless, many of the problems in conventional software development meth-

odologies can be alleviated considerably using incremental modification. One additional benefit of inheritance in this respect is that it enables the construction of *partially*, or *incompletely implemented* abstractions. By partially implemented abstractions we refer to abstractions whose definitions have purposely been left incomplete, and whose properties may thus refer to such components or properties that have not even been implemented yet; late binding is obviously needed for this. Abstractions of this kind are sometimes extremely useful, because they can serve as *specifications* or contracts upon which the concrete implementations can be based, thus facilitating the convergence of analysis, design, and implementation. Such abstractions often have a *communicative* role, allowing the designers to agree upon the interfaces of the abstractions before the actual implementation efforts are started. In general, these ideas are intimately related to Wirth's notion of *successive*, or *stepwise refinement* [Wirth 1971].

In class-based object-oriented systems, partial implementation is usually supported in the form of *abstract classes*. An abstract class is a class that specifies a message interface, but does not fully implement it. Such a class is written with the expectation that its subclasses will add to its structure and behavior, usually by completing the implementation of its incomplete operations. Since abstract classes are implemented only partially, by convention no instances are created from them. In fact, in some papers abstract classes are defined as classes which even *cannot* be instantiated; many systems, such as Smalltalk-80, do not guarantee this, however. Since abstract classes serve only to provide inheritable functionality, they are often also referred to as abstract *super*classes [Bracha 1990]. Other frequently used synonyms are *basic classes* in CLOS terminology [Keene 1989, p. 224], *deferred classes* in Eiffel terminology [Meyer 1988, pp. 234–240], *abstract superpatterns* in Beta terminology [Madsen and Møller-Pedersen 1989], and *partial types* [Halbert and O'Brien 1987].

Representative examples of abstract classes in object-oriented systems are classes *Object* and *Collection* in Smalltalk. Smalltalk's class Object is an abstract superclass that defines the general properties of all objects in the Smalltalk system [Goldberg and Robson 1989, pp. 94–103]. Instantiation of class Object would, however, make little sense since Object exhibits no other behavior than these general operations. Similarly, class Collection defines the general message protocol for various collection classes such as dictionaries, sets, arrays and strings. Many operations in class Collection, such as the iteration method *do:*, have deliberately been left unimplemented, however, and are to be added by the subclasses.

In systems utilizing mixin inheritance (see Section 3.7), partial implementations are used for another purpose. Mixins are small, noninstantiatable portions of behavior that are used solely for adding properties to other classes. Mixin classes do not define any general framework or serve as specifications of some larger abstraction hierarchy. Rather, they just describe reusable pieces of functionality that can be attached to other classes using multiple inheritance. Therefore, as opposed to ordinary abstract superclasses, mixin classes are sometimes referred to as *abstract subclasses* [Bracha and Cook 1990].

In general, we can distinguish two additional forms of inheritance: *inheritance from complete implementations* and *inheritance from partial implementations* [Halbert and O'Brien 1987]. Both forms rely on performing modifications in an incremental fashion and are thus subcategories of the more general notion of incremental modification. Inheritance from complete implementations allows the reuse and refinement of existing complete abstractions, while inheritance from partial implementations serves as a specificational structuring

tool, enabling the successive refinement of programs from abstract specifications towards successively more concrete implementations. Some researchers have argued that inheritance from complete implementations is harmful and should be avoided [Lieberherr et al. 1988; Lieberherr and Holland 1989].

Note that in a way, abstract classes have a *dual role*. On the one hand, they have a *conceptual* role, and serve as specification or design tools in the above described manner, their presence being motivated by the conceptual analysis of the problem domain. On the other hand, abstract classes also have a more *pragmatic* role: to serve as hooks for improving reusability. These two roles are not necessarily coincident; in order to make a class hierarchy as reusable as possible, it is often essential to split the problem domain into smaller classes (and have more abstract classes) than would otherwise be necessary from the conceptual viewpoint. This is because the smallest unit of reuse in most object-oriented systems is the class. Mixin inheritance discussed later in Section 3.7 is a good example of the potential discrepancy between reuse and modeling.

3. VARIATIONS OF INHERITANCE

Understanding depends on expectations based on familiarity with previous implementations.

—MARY SHAW

What we've got is freedom of choice. What we want is freedom from choice.

—From a song by a new-wave band, *Devo*

Object-oriented language design space is not a dichotomy, and these languages cannot be categorized into completely clear-cut, orthogonal subclasses. Rather, they seem to be more like a result of "mixin inheritance": the same basic themes and patterns are repeated over and over again in slightly different forms and variations. This similarity between models is intriguing, and it sparks the desire to explore the possible common constituents further. This section undertakes a detailed analysis of the variations of inheritance mechanisms in object-oriented programming systems, aiming at new insights as to what the possible common constituents of different inheritance models are. At the same time, some further concepts and notions will be introduced. Finally, a simple taxonomy of the basic mechanisms and issues underlying the seemingly divergent forms of inheritance will be presented.

3.1 Class Inheritance Versus Prototype Inheritance

Class is a state of grace that few people have.

—DOROTHY CULLMAN

Prototype: the first or primary of anything; the original (thing or person) of which another is a copy, imitation, representation, or derivation, or is required to conform; a pattern, model, standard, exemplar, archetype.

—*Oxford English Dictionary VIII,* p. 1512

Object-oriented systems are usually built around *classes*. Classes are descriptions of objects capable of serving as templates or "cookie-cutters" from which *instances*, the actual objects described by classes, can be created. This creation process is typically known as *instantiation*. In broad terms, a class represents a generic concept, or a "recipe," while an instance represents an individual. A class holds the similarities among a group of objects, dictating the structure and behavior of its instances, whereas instances hold the local data representing the state of the object.

Class-based systems are quite class-centric. To add a new kind of an object to the system, a class describing the properties of that object type must be defined first. Similarly, inheritance can only take place between classes, whereas instances are completely "sterile", i.e., incapable of serving as parents. Therefore, the model is commonly referred to as *class inheritance* [Stein et al. 1988; Wegner 1987, 1990].

An interesting alternative for traditional class inheritance is *prototype inheritance*, or *object inheritance* [Blaschek 1994; Borning 1986; Dony et al. 1992; Liebermann 1986; LaLonde et al. 1986; Stein et al. 1988; Stein 1987; Ungar and Smith 1987]. As opposed to class-based systems, in prototype-based systems there are no classes. Rather, new object types are formed directly by constructing concrete, full-fledged objects, which are referred to as *prototypes* or *exemplars*. A prototype can be thought of as a standard example instance, which represents the default behavior for some concept. Prototype-based systems provide no classes, and therefore in these systems there is no notion of instantiation either. The effect of instantiation, the ability to create multiple objects of similar kind, is achieved by *cloning* (copying) existing objects instead. In addition, objects in prototype-based systems are usually *individually modifiable*. By virtue of the individuality of objects, prototype-based systems support incremental modification at the level of individual objects, as opposed to class-based systems in which typically only groupwise (class-level) modification is possible.

From the philosophical viewpoint the distinction between class-based and prototype-based systems reflects the long-lasting philosophical dispute concerning the representation of abstractions. Plato viewed *forms*—stable, immutable descriptions of things—as having an existence more real than instances of those abstractions in the real world [Plato 1981, Books 6 and 7]. Object-oriented languages like Smalltalk and Simula are Platonic in their explicit use of classes to represent similarity among collections of objects. Prototype-based systems represent another way of viewing the world, in which one chooses not to categorize things, but rather to exploit their alikeness. A typical argument in favor of the prototype-based approach is that people seem to be a lot better at dealing with specific examples first, then generalizing from them, than they are at absorbing general abstract principles first and later applying them in particular cases [Lieberman 1986].

There are many variations of prototype-based systems [Dony et al. 1992; Taivalsaari 1993c]. Roughly speaking, prototype-based object-oriented languages can be divided into two broad categories according to how incremental modification is supported in them: *delegation-based* and *copying-based*. In delegation-based languages, there is a special *delegation* mechanism that provides the incremental modification capability [Lieberman 1986]. In delegation-based languages, when an object receives a message it does not understand, the object will delegate the message to those objects that have been designated as its "parents." Parent objects will then try to carry out the message on the delegating object's behalf. During the delegation process, the self-reference will remain pointing to the original receiver, allowing late-bound properties in parents to operate in the context of the original receiver. Delegation will be discussed in more detail in the next section. The best example of a delegation-based object-oriented language is *Self* [Ungar and Smith 1987], a Smalltalk-like language that has become the yardstick against which other prototype-based object-oriented languages are measured.

Borning [1986] has shown that a prototype-based language does not necessarily have to support delegation in order to be a full-fledged object-oriented language. Borning demonstrated that by utilizing cloning (copying) and the possibility to dynamically add new properties to objects in the presence of the regular late-binding message sending mechanism, it is possible to capture the essence of inheritance—incremental modification—in a prototype-based language even without delegation. Unless supplanted with some additional mechanisms and implementation techniques, this model has some problems, however. In particular, as a result of extensive use of copying, memory consumption can be excessively high. Also, groupwise

modification of objects is more tedious than in delegation-based systems due to absence of common parent objects or any explicit inheritance hierarchy. Examples of prototype-based object-oriented languages built along the lines of Borning's model are *Omega* [Blaschek 1991] and *Kevo* [Taivalsaari 1993c]. Prototype-based object-oriented languages have been discussed in detail in a book by Blaschek [1994].

3.2 Delegation Versus Concatenation

Inheritance is acquisition of properties from past generations. More practically, inheritance is a relationship between objects that guarantees that a descendant has at least the properties that its parents have. By incrementally adding, redefining or defeating properties, the descendant can become differentiated from its parents. At the level of object interfaces this implies that the record representing the descendant must contain at least those identifiers that its parents have. In addition, the descendant can introduce new identifiers that are either *distinct* or *overlapping*. An incrementally added identifier is said to be *distinct* if no other identifiers with the same name exist in the same inheritance DAG. Otherwise, the identifier is *overlapping*. In general, from the viewpoint of interface management, inheritance is simply a *combination operation* $R = P \oplus \Delta R$ on record structures $P = [\ldots]$ and $\Delta R = [\ldots]$ with possibly overlapping identifiers (see Wegner and Zdonik [1988]). This combination operation must be able to relate the interface (name space) of descendant R to the interface of parent P.

Within computer memory there are only two ways to express direct relationships between things: *references* or *contiguity*. Based on this observation, two elementary strategies for implementing inheritance can be distinguished: *delegation* and *concatenation*. Delegation is a form of inheritance that implements interface combination by *sharing* the interface of the parent—in other words, using references. Concatenation, on the other hand, achieves the same effect by *copying* the interface of the parent into the interface of the descendant—as a result of this, the descendant's interface will be contiguous. As an example of delegation and concatenation, consider the simple example below. Let us assume that we have the following kind of object *aPen* defined in our system with two variables *x* and *y* and a method called *draw*, defined using simple record notation.

```
aPen:-[VAR x, VAR y, METHOD draw];
```

Suppose that we now want to define another object *aTurtle* that inherits the properties of *aPen* and adds a new variable *heading* and a method *forward*. If we accomplish this using delegation, the interface of the descendant will be as follows:

```
aTurtle:-[PARENT p :-aPen, VAR heading,
METHOD forward];
```

On the other hand, using concatenation the interface of the resulting object will be contiguous, as illustrated below:

```
aTurtle:-[VAR x, VAR y, METHOD draw,
VAR heading, METHOD forward];
```

In both cases the basic result is the same: the descendant will be able to respond to messages corresponding to the inherited identifiers in addition to the incrementally added ones. However, the way in which message sending is handled in these approaches is different. In delegation, those messages that are not accepted by the most specialized part of the object must be *delegated* (forwarded) to parents. In concatenation, the interface of every object is self-contained, and thus no forwarding is needed. Since the term *delegation* implies *shared responsibility* for the completion of a task, it would be misleading to use that term to denote also the latter form of inheritance. Delegation and concatenation have been illustrated in Figure 9.

Note that the difference between delegation and concatenation is indepen-

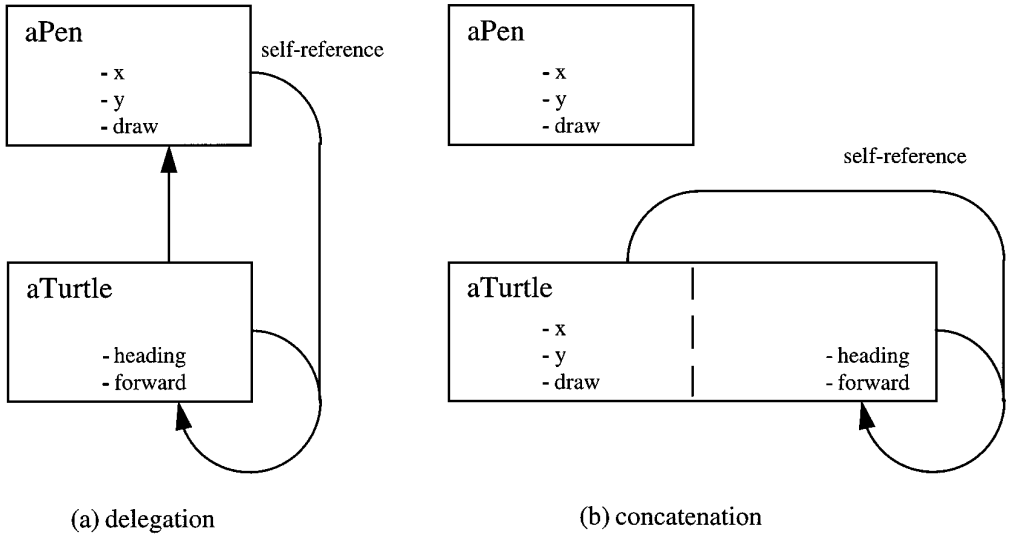

Figure 9. Delegation versus concatenation.

dent of the question of whether the system is based on classes or prototypes; delegation and concatenation can equally underlie both models. The main difference is that in class-based systems delegation or concatenation takes place between classes, whereas in prototype-based systems the same occurs between individual objects. In fact, although the term *concatenation* is not widely used in the current literature of object-oriented programming, there are good historical reasons for its use. The concept of *prefixing* in Simula [Dahl et al. 1968], which later evolved into the modern concept of inheritance, was originally defined in terms of *textual concatenation of program blocks* [Cook and Palsberg 1989] (see Dahl et al. [1972, pp. 202–204, Nygaard and Dahl 1987]). Even in the current implementations of Simula and its descendant Beta [Kristensen et al. 1983] the interfaces (to be more specific: the virtual operation search tables) of classes and their subclasses are self-contained and independent of each other [Madsen and Møller-Pedersen 1988; Magnusson 1991]. On the other hand, as shown by Stein [1987], the way that inheritance between classes is achieved in Smalltalk is analogous to the delegation approach adopted in prototype-based languages such as Self. In both approaches unresolved messages will always be forwarded to separate constructs; in Smalltalk this separate construct is a superclass, and in Self it is a parent object.

In general, delegation and concatenation are distinct strategies for implementing inheritance, both of which can equally underlie class-based and prototype-based systems. Simula and its descendant Beta are examples of languages which implement *class inheritance by concatenation*. Smalltalk, on the other hand, implements *class inheritance by delegation*. Self is the best example of a language that implements *prototype inheritance by delegation*. The copying-based approach discussed in Section 3.1 represents *prototype inheritance by concatenation*. Thus, although it has sometimes been claimed that delegation is a mechanism that can be thought of as underlying all inheritance mechanisms [Stein 1987], this is not really the case. One of the few papers in which a similar distinction between different kinds of object-oriented systems has been made is, Snyder [1991],

Table 1. Delegation versus concatenation.

Inheritance strategy	Delegation	Concatenation
Record combination strategy	sharing/references	copying/contiguity
Interface dependence	dependent (life-time sharing)	independent (creation-time sharing)
Inheritance DAG	preserved	flattened

wherein Snyder distinguishes *monolithic objects* and *multiobjects*. Snyder's notion of monolithic objects is roughly analogous to our notion of concatenation, whereas his multiobjects represent the use of delegation (see also Sakkinen [1989]).

The main differences between delegation and concatenation have been gathered into Table I. Most of the differences between the approaches pertain to the treatment of inheritance graphs (DAGs). Delegation preserves inheritance DAGs without modifications, but concatenation requires DAGs to be flattened out into contiguous name spaces. This flattening implicitly prevents hazardous cycles in inheritance DAGs, but at the same time it causes some important differences in the usage of these models. The notions of *life-time sharing* and *creation-time sharing* mentioned in the table will be defined later in Section 3.8.

Delegation and concatenation both have certain advantages over each other. For instance, owing to dependent interfaces, delegation enables flexible propagation of change: any change to parents will be automatically reflected to descendants. This can be extremely useful at times, allowing a large group of objects to be modified simply by changing a common parent. On the other hand, thanks to self-contained interfaces, concatenation supports independent modification of object interfaces, making possible *renaming* and *selective inheritance* (Section 3.6). As for efficiency, delegation is generally more space-efficient due to extensive use of sharing, but concatenation is generally more time-efficient since the absence of sharing makes it easier to optimize the method lookup more aggressively.

3.3 Direction and Completion of Message Lookup

> Perfection is achieved only on the point of collapse.
>
> —C.N. PARKINSON

In Section 2.3.2 we described the general message sending algorithm that can be seen as underlying all inheritance mechanisms. The algorithm consisted of four elements—SEND, LOOKUP, CALL and ERROR—that were intentionally defined somewhat vaguely. In particular, the LOOKUP part of the algorithm simply stated that as part of message matching the inheritance DAG will be traversed node by node. We did not, however, specify what the actual traversal order is. Neither did we completely specify when this lookup will eventually end. For example, if the DAG contains several overlapping identifiers, it is unclear which one of these properties should be invoked, or should they all become invoked. In the latter case it would be important to specify also the exact order in which all these overlapping properties are to be executed. Moreover, in case of multiple inheritance the overlapping of identifiers is often inadvertent, and the undesired identifier *name collisions* should somehow be taken into account. These issues will be addressed in this and the next section. Note that all these questions apply equally to class-based and prototype-based systems, regardless of whether they are based on delegation or concatenation.

In most contemporary object-oriented systems the message lookup proceeds

from the most recently defined (descendant) parts of objects towards the least recently defined parts (parents). For example, the behavior of message lookup in Smalltalk has been described informally as follows [Goldberg and Robson 1989, p. 61]: "When a message is sent, the methods in the receiver's class are searched for one with a matching selector. If none is found, the methods in that class's superclass are searched next. The search continues up the superclass chain until a matching method is found". Thus, in Smalltalk message lookup always has a specific direction, and will terminate as soon as the first matching property is found. This convention is used in various other object-oriented languages including C++ and Eiffel, except that in many systems multiple inheritance makes things somewhat more complicated. Since the lookup is dominated by the most recently defined parts of objects, i.e., those properties that have been defined at the descendant level, we call this scheme *descendant-driven* inheritance.

As usual, there are some interesting variations. Beta [Kristensen et al. 1983; Madsen and Møller-Pedersen 1989], a descendant of Simula, uses a totally opposite *parent-driven* lookup scheme. In Beta, property lookup is started from the topmost superpattern (the Beta equivalent of superclass) in the current inheritance DAG. Beta supports single inheritance only, so this topmost superpattern is always unique, and the inheritance DAG can be seen simply as a linear path from parents to descendants. When a message is sent to an object (in Beta terminology: a virtual procedure is invoked), lookup proceeds from the virtual procedures defined in the topmost superpattern towards more recently defined superpatterns of the object until a matching procedure is found. This matching procedure will then be invoked in the normal manner. Since superpatterns have precedence over their subpatterns, the Beta-style reverse inheritance strategy ensures that descendants can never totally override the behavior inherited from parents, thereby providing additional support for maintaining the behavioral compatibilities between parents and descendants.

The distinction between descendant-driven and parent-driven lookup schemes is illustrated in Figure 10. Note that the parent-driven lookup scheme is not really suited to systems with multiple inheritance, because in such systems inheritance DAGs do not necessarily have any single root node from which the lookup could be started.

Besides the differences in the *direction* of message lookup, the inheritance scheme of Beta deviates from mainstream object-oriented languages also in other ways. One important difference pertains to the *completion* or exhaustion of message lookup. In most object-oriented systems message lookup is terminated as soon as a matching property has been found and executed. If the programmer wants to execute the overridden properties inherited from the parents, it is the programmer's responsibility to explicitly do so by defining the methods to explicitly call the methods of their parents. Many object-oriented languages provide a special *super-reference* exactly for this purpose, as discussed earlier in this paper.

Beta addresses the lookup exhaustion issue differently. In Beta, the overlapping ("extended") properties in subpatterns will always be invoked *automatically* without programmer intervention. After finding and executing a matching property, the lookup will be continued implicitly in the parent-driven lookup order until *all* the corresponding properties in the object have been executed. This is essentially different from mainstream object-oriented systems in which even one matching property will suffice to terminate the lookup. As noted by Madsen and Møller-Pedersen, the Smalltalk-style of terminating the lookup immediately after encountering the first matching method is more flexible, but is also a potential source of errors since the programmer may forget

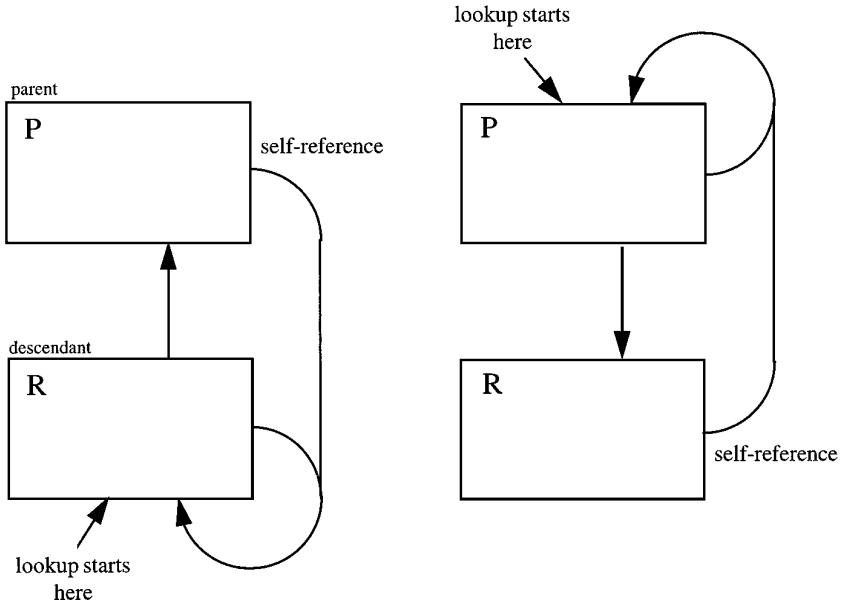

Figure 10. Descendant-driven and parent-driven forms of inheritance.

to execute the corresponding property in the superclass [Madsen and Møller-Pedersen 1989].

Theoretically, the direction and exhaustion issues are closely related to record combination. In his thesis, Cook [1989, p. 10] suggested that a distinction be made between *left preferential* and *right preferential* record combination in order to specify the precedence of message lookup. However, since the terms *left* and *right* are quite imprecise and open to interpretations, the terms *parent* and *descendant* are used here in the same meaning. In general, theoretically we can make a distinction between five basic inheritance (record combination) schemes, as illustrated in Table II. *Strict* record combination refers to situations in which overlapping property names are strictly forbidden. *Asymmetric* combination (see Harper and Pierce [1991]) refers to such schemes in which the lookup is limited to the first matching property. *Composing* combination requires all the matching properties to be executed. In Table II, x is the property to be executed, and P and R denote the sets of properties defined at the parent and child level, respectively. A special operation o is used to denote execution ordering; for instance, $a\ o\ b$ implies that both a and b are executed, but a is invoked before b.

Note that although the inheritance scheme of Beta is radically different from than in most other object-oriented systems, these systems are typically capable of simulating each other quite easily. In Smalltalk, for instance, Beta's *composing* message lookup scheme can be simulated simply by systematically placing an explicit super-reference within every method. Moreover, if these super-references are always positioned in the beginning of each method, the lookup order becomes logically parent-driven, even though the physical search order still remains descendant-driven. The same technique can be used in Beta which provides a special *inner* construct for controlling the order of subpattern execution [Kris-

Table 2. Variations of record combination for message lookup

Object = $P \oplus R$	$x \in R \wedge x \notin P$	$x \in R \wedge x \in P$	$x \notin R \wedge x \in P$
strict	*R.x*	*ERROR*	*P.x*
asymmetric descendant-driven	*R.x*	*R.x*	*P.x*
asymmetric parent-driven	*R.x*	*P.x*	*P.x*
composing descendant-driven	*R.x*	*R.x* o *P.x*	*P.x*
composing parent-driven	*R.x*	*P.x* o *R.x*	*P.x*

tensen et al. 1983; Madsen and Møller-Pedersen 1989]. The behavior of *inner* is analogous to that of super-reference, but its direction is reverse. Rather than invoking properties of *super*classes, Beta's *inner* construct invokes properties in *sub*patterns. Thus, by placing an inner-reference in the beginning of each virtual function in a Beta program, the lookup order becomes logically descendant-driven. However, there is no easy way to simulate *asymmetric* message lookups in Beta.

There are also other techniques that can be used to control the direction and exhaustion of message lookup. In particular, *mixin inheritance* discussed in Section 3.7 allows all the inheritance schemes listed in Table II to be simulated flexibly; different schemes are achieved simply by changing the order in which base classes and mixins are combined. For a thorough discussion on this idea, refer to Bracha and Cook [1990]. Among the currently available object-oriented languages, the CLOS system with its special *before, after* and *around* methods provides perhaps the most comprehensive facilities for dealing with the lookup direction and exhaustion. For further information on these aspects of CLOS, see DeMichiel and Gabriel [1987] and Keene [1989, pp. 11–13, and 50].

3.4 Ordered Versus Unordered Inheritance

A *name* is a spoken sound significant by convention, without time, none of whose parts is significant in separation.

—ARISTOTLE, *De Interpretatione §2*

It's useful to the people that name them, I suppose. If not, why do things have names at all.

—LEWIS CARROLL, *Through the Looking Glass*

A problematic issue pertaining to multiple inheritance are *name collisions*. A name collision occurs when the inheritance DAG contains overlapping properties and the property lookup algorithm is unable to decide which one of these to execute. Not all overlapping properties result in name collisions, however. For example, in single inheritance the overlapping of properties is a natural result of incremental modification: when a descendant redefines some of its inherited properties to differentiate itself from its parent, the newly redefined properties naturally overlap with the inherited ones. This form of overlapping of properties poses no problems for message lookup algorithms and is sometimes called *vertical overlapping* (see Knudsen [1988]). In multiple inheritance vertical overlapping occurs analogously when two properties with the same name are located along the same path in an inheritance DAG. For example, in Figure 11a identifier *x* in node D vertically overlaps with identifier *x* in its parent node A.

Problems arise when overlapping of properties is *horizontal*. Horizontally overlapping identifiers are those that are located along distinct paths in an inheritance DAG (see property *y* in Figure 11b), and thus horizontal overlapping can take place only in multiple inheritance. Generally speaking, there are two basic approaches to dealing with name collisions: *ordered* and *unordered* inheritance [Chambers et al.

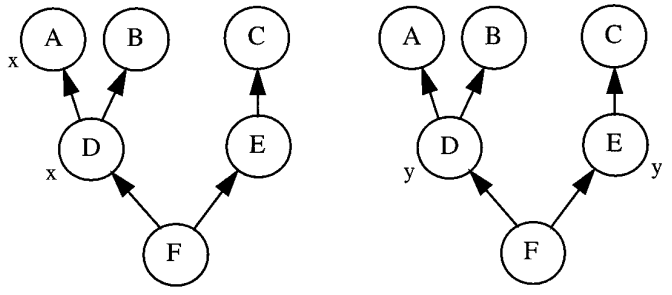

Figure 11. Vertically and horizontally overlapping identifiers.

1991]. Most Lisp-based systems such as Flavors [Moon 1986], CommonLoops [Bobrow et al. 1986] and CLOS [DeMichiel and Gabriel 1987; Keene 1989] obey ordered multiple inheritance schemes. In these systems the inheritance DAG is first somehow *linearized*, and the first matching property on this linear path will then be executed regardless of the other properties with the same name. In contrast, languages like Trellis/Owl [Schaffert 1986], Eiffel [Meyer 1988], C++ [Ellis and Stroustrup 1990] and CommonObjects [Snyder 1986b] treat inheritance DAGs without any relative ordering. In these languages, name collisions are considered programming errors, and any ambiguities must be resolved explicitly by the programmer.

Note that not even horizontal overlapping of properties should always be treated as a name collision. If two conceptually unrelated abstractions are combined using multiple inheritance, it may well happen that these abstractions accidentally contain overlapping identifiers [Chambers et al. 1991]. Since the abstractions are conceptually unrelated, it is clear, however, that the operations of these abstractions are not intended to access each others' properties. For instance, in the example above, the late-bound invocation of property *y* via self-reference from node A or D should *not* result in a name collision, although a similar invocation from node F certainly should. In order to be able to cope with these kinds of situations, some systems have been augmented with quite complicated mechanisms and inheritance rules. For example, an earlier version of Self included a special *sender path tiebreaker rule* for addressing this particular problem [Chambers et al. 1991]. For further information on the treatment of name collisions, see particularly Knudsen [1988].

3.5 Dynamic Inheritance

Wood may remain ten years in the water, but it will never become a crocodile.

—*Congolese proverb*

In Section 3.2 a distinction between two fundamental strategies for implementing inheritance—delegation and concatenation—was made. It was concluded that both strategies have their own benefits that are difficult to capture using the other strategy. In this section this discussion is taken a bit further by discussing *dynamic inheritance*: the ability to change parents of objects dynamically at runtime [Chambers et al. 1991; Stein et al. 1988]. Despite being intuitively a rather dangerous language feature, dynamic inheritance provides some notable benefits, and therefore, although dynamic inheritance has typically been available only in some proto-

type-based systems, it has recently captured the interest of many researchers of class-based systems as well.

Recall from the earlier discussion that delegation-based systems accomplish the interface combination required by incremental modification by *sharing* the interface of the parent, i.e., using *references* rather than copying. In class-based systems, the references between children and parents are usually maintained by the system and are completely inaccessible to the programmer. In contrast, in prototype-based systems these references are typically implemented as *parent slots*. By giving these slots names, and by making them assignable, we allow the parents of objects to be changed on the fly. Consider the following object definitions:

```
aPEN:-[VAR x, VAR y, METHOD draw];
aTurtle:-[PARENT parent:-aPen, VAR
heading, METHOD forward];
```

Assuming that the parent slot *parent* behaves like an ordinary variable, we can dynamically change the parent slot to refer to some other object. Below we define a new object *a3dPen*, supposedly representing a three-dimensional counterpart of *aPen*, and make this object the parent of *aTurtle*.

```
a3dPen:- [VAR x, VAR y, VAR z, METHOD
draw];
aTurtle.parent:- a3dPen;
```

After these operations, the behavior of *aTurtle* is no longer the same as it was before. The object contains an extra variable *z*, and the actual method *draw* is different than before. As apparent, dynamic inheritance can be quite dangerous. If the properties of the new parent do not comply with those needed by the descendant, runtime binding errors will result. Dynamic inheritance can also have a deteriorating effect on performance, since the possibility to change parents at runtime prevents certain kinds of message lookup optimizations from being done [Chamber et al. 1991].

The benefits of dynamic inheritance are best described by giving an example. In the design and implementation of programs we are often faced with things and structures that can be in different states or conditions. In this context we are not referring to the concrete state of the variables of these objects, such as the integer 12345 in variable *foo*, but rather to a more conceptual or logical kind of state. For instance, container objects such as stacks, lists and queues can be empty, nonempty or full. In graphical user interfaces, windows can usually be open, closed or represented as icons. Similarly, application-specific objects such as bank accounts, flight reservations, or patient registers also exhibit many kinds of logical states. The behavior of the operations of objects typically varies considerably depending on these logical states. For instance, the behavior of the *pop* operation of a stack depends essentially on whether the stack contains items or is empty; in the latter case *pop* must not return any value but rather raise an error. Similarly, when a bank account has been frozen by the bank, the teller machine should not dispense any money when a withdrawal is requested but rather swallow the card. In conventional object-object systems, these kinds of logical states or *modes*, as they are sometimes called, are hard to express. Usually, additional state variables and control structures are needed. For instance, each of the basic window manipulation operations such as *open, close, iconify, resize* and *zoom* typically has to contain an explicit if-statement or case-statement to determine whether the operation is appropriate for the window in the current situation. Obviously, the diffusion of mode information and accumulation of control structures both tend to have a negative impact on the readability and maintainability of code.

Dynamic inheritance provides a solution to the problems with logical states. Since the behavior of objects can be changed at runtime, not all the information about an object has to be described in one class or prototype. Rather, by implementing a separate set of methods

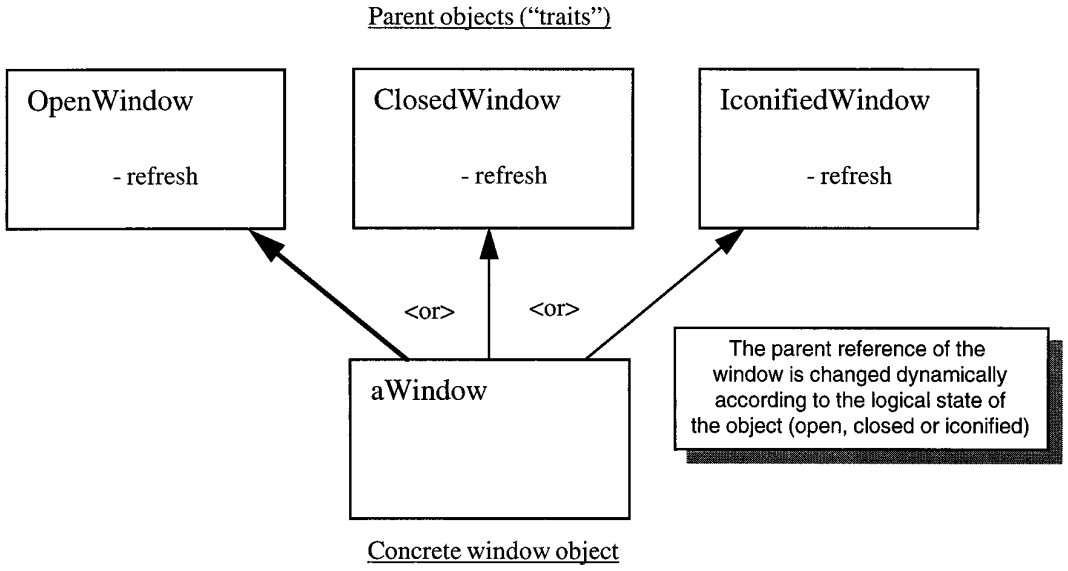

Figure 12. Dynamic inheritance.

for each logical state, the behavior of each mode can be kept clearly apart from the others. Dynamic inheritance is illustrated in Figure 12 in which we see a window object with three parent objects ("traits" in Self terminology [Chambers et al. 1991]) representing the different logical states of windows (open, closed, iconified). Each of the parent objects implements the same basic set of methods, but the behavior of the methods is specialized to each particular logical state. Initially the parent slot of the prototype is set to point to traits OpenWindow, implying that only those operations that are applicable to open windows are available, but when the logical state of the object changes at runtime, e.g., the window becomes closed, the parent slot is changed correspondingly. Using this approach, method definitions can be kept concise and clear, because the methods in one mode do not have to know how the other modes behave. The number of control structures in programs is reduced substantially.

Note that dynamic inheritance is not typically possible in class-based systems. This is mainly due to the fact that in class-based systems classes describe not only behavioral but *structural* aspects as well; if the superclass of a class were suddenly changed at runtime, the structure of the instance would likely be incompatible the structure specified by the new superclass. The fact that most class-based systems do not support late binding of variables makes the situation even more problematic. Due to the apparent benefits of modes, however, there have been some proposals on how to add corresponding mechanisms to class-based systems [Chambers 1993; Stein 1989; Taivalsaari 1993b]. Among these, Chambers' *predicate classes* seem particularly promising.

3.6 Selective Inheritance

Just as delegation enables dynamic inheritance, concatenation gives rise to some interesting variations of inheritance. For instance, the self-contained nature of object interfaces in concatenation has the advantage of making the object interfaces independent of each other. This possibility is utilized in

Eiffel, which allows the inherited properties to be *renamed* in subclasses [Meyer 1988, pp. 246–250]. Renaming apparently requires that the inheritance mechanism is implemented using concatenation; otherwise the modification of inherited identifiers would not be possible without affecting superclasses.

Besides allowing renaming, independent interfaces have other benefits. In particular, they make possible such forms of inheritance in which *not all* properties of inherited abstractions have to be passed to descendants. Rather, the descendant can be given the explicit possibility to decide which particular properties of parents to inherit and which not. Such forms of inheritance are said to represent *selective inheritance* [Wilkes 1988]. Selective inheritance has been tried out in Sina/St [Aksit and Tripathi 1988], a Smalltalk-based system that dispenses with classes and uses a two-level type concept with separate interfaces and implementations instead. At the level of type interfaces, the programmer can explicitly select which properties of objects to inherit and which not. The Kevo system [Taivalsaari 1993c] also supports selective inheritance through its partial reuse mechanisms.

One of the interesting theoretical aspects of selective inheritance is that it shows that the difference between *inheriting* an abstraction (such as a class or a prototype) and using the abstraction as a *variable* is actually very subtle. From the interface combination viewpoint the only difference between these is that in the former case the property names of the inherited component are implicitly included into the interface of the descendant, and thus the inherited properties can be referred to directly. In contrast, in the latter case the properties are not included and must be accessed indirectly via an intervening variable. As an example of this, consider the following example. Assume that we have the following objects $t1$ and $t2$ defined in our system. Since we are dealing with interface aspects only, the actual method implementations and variable values are ignored.

```
t1 :- [METHOD m1, VAR v1];
t2 :- [METHOD m2, VAR v2];
```

Suppose that we now define two other objects $t3$ and $t4$ which make use of the existing objects $t1$ and $t2$. Object $t3$ uses $t1$ and $t2$ as its parents, whereas $t4$ uses $t1$ and $t2$ as its variables. Note that in this particular example $t3$ has been implemented using delegation and $t4$ using reference variables, but the situation would be analogous using concatenation and containment.

```
t3 :-[PARENT p1 :- t1, PARENT p2 :- t2,
     METHOD m3, VAR v3];
t4 :-[VAR p1 :- t1, VAR p2 :- t2, METHOD
     m3, VAR v3];
```

Basically, the only difference between objects $t3$ and $t4$ pertains to how their properties can be accessed. In case of $t3$ ($t1$ and $t2$ as *parents*), the properties of $t1$ and $t2$ can be accessed directly without having to mention the intervening identifiers $p1$ and $p2$. In case of $t4$ ($t1$ and $t2$ as *variables*), the intervening identifiers along the access path to properties $m1$, $v1$, $m2$ and $v2$ must be mentioned explicitly. For example, the following expressions

```
t3.m1;
t3.v2;
```

are legal, whereas the corresponding messages to $t4$ would lead to binding errors. Thus, for $t4$ the intervening identifiers $p1$ and $p2$ must be used as shown below.

```
t4.p1.m1;
t4.p2.v2;
```

Note that in the presence of selective inheritance the distinction between parents and variables becomes blurred. Selective inheritance allows only a subset of the identifiers of parents to be brought into the interface of the descendant, and hence the degree of "parentness" and "variableness" can be varied. In particular, if a language provides a mechanism by using which any uniquely named property within an object can be accessed di-

```
CLASS P IS
    ... properties ...
ENDCLASS;

CLASS R
INHERIT P IS
    ... modifications (ΔR) ...
ENDCLASS;
```

(a) ordinary (single) inheritance

```
CLASS P IS
    ... properties ...
ENDCLASS;

CLASS R_Mixin IS
    ... modifications (ΔR) ...
ENDCLASS;

CLASS R
INHERIT P, R_Mixin IS
ENDCLASS;
```

(b) mixin inheritance

Figure 13. Ordinary inheritance versus mixin inheritance.

rectly without having to specify the full path to it, then from the viewpoint of interface management the distinction between variables and parents will vanish altogether. This idea is utilized in Sina/St where multiple inheritance is achieved simply by declaring multiple variables at the interface of a type and making the properties of these variables directly accessible by means of an appropriate predicate [Aksit and Tripathi 1988]. Of course, in most object-oriented languages *encapsulation* (information hiding) changes the picture to some extent, since not all variables or methods may be accessible to outside clients due to being declared private.

The use of selective inheritance can lead to some conceptual and technical problems. In particular, incautious omission of inherited properties can produce objects whose interfaces are incomplete, and that do not possess all the properties to which their operations need access. Therefore, additional mechanisms may have to be introduced to guarantee the completeness of objects. For further information on selective inheritance, refer to Aksit and Tripathi [1988], Cook [1989a] and Wilkes [1988].

3.7 Mixin Inheritance

Mixin inheritance is a certain way of using inheritance that has received quite a lot of attention in the literature of object-oriented programming [Bracha 1992; Bracha and Cook 1990; Hendler 1986]. This is not surprising, considering that mixin inheritance has certain important theoretical and pragmatic benefits. On the theoretical side, mixin inheritance has been proven to be capable of capturing the functionality of several other forms of inheritance [Bracha and Cook 1990]. Therefore, mixin inheritance is sometimes regarded as a more "elementary" form of inheritance that can provide help in understanding the other forms. On the pragmatic side, mixin inheritance is beneficial because it can considerably improve the reusability of program components.

The basic idea of mixin inheritance is simple. Unlike in ordinary inheritance, in which the modification (delta) parts in class definitions are embedded directly in the new definitions (see Figure 13a), in mixin inheritance separate *mixin classes* are created to hold the modifications [Bracha and Cook 1990; Hendler 1986]. A mixin class is syntactically identical to a normal class, but

its intent is different. Such a class is created solely to be used for adding properties to other classes—one never creates an instance of a mixin. New concrete classes, such as class *R* in Figure 13b, are constructed by combining primary parent classes (class *P* in Figure 13b) with secondary mixin classes (*R_Mixin*) using multiple inheritance.

Since mixin classes do not have superclasses and are not therefore structurally bound to any specific place in the inheritance hierarchy, the same modifications (delta parts) can be repeatedly used (inherited) without having to rewrite or copy their contents manually when similar modifications are needed in other places [Bracha and Cook 1990]. This enhances reusability by allowing the same functionality to be flexibly added to various components. Using single inheritance in a similar situation, considerable code duplication would result because the same pieces of code would have to be repeatedly written in several classes. Ordinary multiple inheritance would usually suffice to solve the problem, too. However, ordinary multiple inheritance is prone to cause difficulties if the same classes become inherited two or more times along different inheritance paths (e.g., class has two superclasses that have a common superclass). As already mentioned earlier in the article, this problem is commonly referred to as *repeated inheritance* [Meyer 1988, pp. 274–279], *fork-join inheritance* [Sakkinen 1989], or *diamond inheritance* [Bracha 1992, pp. 24–26].

Note that in mixin inheritance multiple inheritance is used strictly for combinatory purposes. Provided that the mixin classes are implemented correctly, no additional manual "gluing" is needed when base classes and mixins are inherited (combined). In practice this requires that the methods of mixin classes are implemented in such a way that they are open to extensions, and remember to invoke the corresponding methods in their surrounding environment (see, e.g., Bracha and Cook [1990]).

Mixin inheritance is not without problems. From the conceptual point of view, one of the less beneficial characteristics of mixin inheritance is that it tends to cause confusion in the relationship of object-oriented programming and conceptual modeling by diversifying the role of classes. As apparent in Figure 13b, in mixin inheritance classes are used for three distinct purposes. First, there are *base classes* that define the general characteristics of a hierarchy of concepts. These classes are either concrete instantiable classes, or abstract noninstantiable classes that exist solely to be refined by their subclasses. Second, there are *mixin classes* that serve as elementary units of reusable behavior. Mixin classes cannot be used in isolation but must first be combined with a concrete class to become functional; therefore, mixin classes are sometimes referred to as abstract *sub*classes [Bracha and Cook 1990]. Third, mixin-based systems also have *combination classes* that are needed for instantiating concrete instances with the desired functionality.

In summary, mixin inheritance can considerably increase the reusability of class definitions, but at the same time it may decrease the conceptual clarity of the system by requiring the same class mechanism to be used for three different purposes in a rather subtle and unintuitive way. In this sense, mixin inheritance serves as a practical example of the fact that *reusability and conceptual modeling are opposite goals*: in order to obtain maximum reusability, one usually has to sacrifice modeling, and vice versa. For further information on mixin inheritance, refer to Bracha and Cook [1990].

3.8 Life-Time Sharing Versus Creation-Time Sharing

Inheritance is a familiar concept from the real world. All living organisms inherit genetic information from their an-

cestors, and thus inheritance serves as one of the fundamental mechanisms for maintaining life on earth. Besides its biological meaning, the term inheritance has various uses in human languages. For instance, it is common to speak of *cultural inheritance* ("Aboriginals inherited their traditions and customs from their ancestors"), *economic inheritance* ("Frank inherited a lot of money from his aunt"), *inheritance of status* ("princess Victoria inherited her rank from her father, the king of Sweden") and so on. The term inheritance can also have more colloquial uses such as "this paper inherited a lot of ideas from my earlier work." All these uses of the term inheritance have certain things in common. In particular, inheritance is synonymous with *receiving*; in most sentences the verb *inherit* could be replaced with *receive* and the meaning of the sentence would remain the same. Note that *receiving* implies that there must necessarily be at least two separate entities—the donor and the receiver—that participate in the inheritance process.

A characteristic aspect of inheritance is *sharing*. As a result of inheritance, the donor and receiver are expected to *share* something with each other. Sharing need not be physical, however. For instance, although it is common to say that the child "shares the blue eyes of his father", it does not mean that the father and child will have to use the same pair of eyes. Similarly, although identical twins "share" the same genetic information, they are nevertheless two separate persons. In general, in its real world meaning, inheritance usually refers to very transitory relationships between things. In biological inheritance, for instance, the combination of genetic information takes place within a short period of time, and the resulting new "abstraction" later becomes completely independent of its parent (well, at least in principle, although in case of human beings the economic ties usually remain. . .). This independence is reasonable, because it would be impractical for parents and children to physically share organs with each other.

In object-oriented programming the meaning of inheritance is somewhat different. Although inheritance still basically implies receiving, in object-oriented programs inheritance usually creates much stronger relationships between things than in the real world. For instance, when a subclass inherits properties from its superclass, it is common that the methods of the superclass are made available to the subclass by physically sharing the same code. Although such "Siamese" sharing is alien to biological inheritance, it is reasonable in programming, because requiring everything to be duplicated in the subclass would waste a lot of memory. Snyder has expressed this by claiming that "a key goal for any implementation of an object-oriented language is to allow the same compiled code to be used for each class that inherits a given method" [Snyder 1986b]. Not all inheritance in object-oriented programming results in physical sharing, however. For instance, the instance variables in instances of classes are always distinct from those in the instances of superclasses.

From the theoretical viewpoint, it is useful to make a distinction between two fundamental forms of sharing that serve as a basis for property inheritance: *life-time sharing* and *creation-time sharing*. Life-time sharing refers to physical sharing between parents and children. Once the sharing relationship between parent and child has been established, the child remains sharing the properties of its parent until the relationship is removed. Furthermore, every change to the parent is implicitly reflected to the child. Creation-time sharing, in turn, implies that sharing occurs only while the receiving process is in progress. Creation-time sharing is characterized by the independent evolution of the parent and the child, and is typical of the real world. Delegation discussed in Section 3.2 results in life-time sharing, whereas concatenation results in creation-time sharing. The notions of

life-time sharing and creation-time sharing were introduced by Dony et al. [1992], who realized that these concepts are useful in analyzing the differences between various prototype-based object-oriented systems.

The forms of sharing have a profound effect on the semantics of object-oriented languages. Generally speaking, inheritance in real-world object-oriented languages can occur at three different levels: behavior, state and structure. *Inheritance of behavior* refers to the passing of operations from parents to their children. Since operations are typically immutable, in many cases this can be accomplished simply by sharing code. *Inheritance of state*, in turn, refers to the passing of contents (values) of variables from parents to the children. This can imply either sharing or duplication, depending on what is appropriate in each particular situation. Sharing of state leads to objects whose states are intimately connected to each other and which thus are *dependent* on each other. Duplication of state, on the other hand, leads to *independent* objects [Stein 1987]. Note that inheritance of state is not typically possible in class-based systems, because in such systems inheritance occurs between *definitions* of objects (i.e., classes) rather than between objects themselves. Instead, class inheritance supports *inheritance of structure*, by which it is meant that normally only the information that is needed for reconstructing (allocating) the inherited instance variables is passed to descendants. This implies that there must be an additional mechanism to pass information about instance variables to subclasses. Some prototype-based languages provide special *copy-down slots* for this purpose.

3.9 Single Dispatching Versus Multiple Dispatching

Most object-oriented systems are based on *single dispatching*. Messages are sent to distinguished receiver objects, and the runtime type of the receiver determines the method that is invoked by the message. The possible parameters in the message are passed on to the invoked method, but do not participate in the actual lookup. Although this approach is intuitive and works well in most situations, it has some drawbacks. For instance, one implication is that a binary expression such as "5+4" gets the unnatural interpretation.

```
5.plus(4).
```

In other words, 5 is treated as the receiver of the message *plus*, while 4 is regarded as a parameter that plays no role in actual method lookup. Although such "currying" works fine in most situations, it is conceptually undesirable since in arithmetic expressions right hand sides are usually no less important than left hand sides. It can also have a negative effect on program readability. For instance, in Smalltalk all binary expressions are parsed left to right, and normal arithmetic precedence rules do not apply. This makes mathematical expressions in Smalltalk different from those in most other languages in which multiplication and division take precedence over addition and subtraction [Goldberg and Robson 1989, p. 28].

Besides conceptual problems, single dispatching can also lead to some more technical problems and require duplication of code. In order to make single dispatching handle properly situations in which the same type of object may receive different kinds of parameters, the use of additional control structures is usually needed. For instance, sending a message "5 + 4.4" requires the integer method + to be able to check at runtime whether the parameter is an integer or a floating point number, and handle both integer and floating point addition. Conversely, since it is equivalently legal for the programmer to send a message "4.4 + 5," the floating point class must include the corresponding code to handle both real numbers and integers. As a result, the same pieces of code may

have to be included in two places, or, if the message is *n*ary rather than binary, in *n* places.

In general, traditional message passing schemes are sometimes too "selfish." There is a single receiver whose type determines which method is executed, while the parameters are ancillary and have no role in method selection. An object-oriented language based on this kind of an approach is termed a *single dispatching language*. In many situations, however, it would be beneficial to generalize the message binding mechanism to more than just single receiver. To surmount the limitations of single dispatching languages, some object-oriented languages include a more general form of message passing in which multiple arguments to a message can participate in the lookup process. These languages are called *multiple dispatching* languages. Since methods in multiple dispatching languages may belong to several classes simultaneously, they are called *multimethods*. Multiple dispatching is common in Lisp-based object-oriented systems, where multimethods are known as *generic methods* [DeMichiel and Gabriel 1987]. Multimethods were pioneered in CommonLoops [Bobrow et al. 1986], but the best-known multiple dispatching object-oriented language of today is CLOS [DeMichiel and Gabriel 1987; Keene 1989].

Some researchers claim that multiple dispatching provides increased expressive power over single dispatching [Chambers 1992]. With multimethods, the message lookup may involve all arguments, not just the receiver, and thus the above mentioned problems with the "asymmetry" of single dispatching can be avoided. Multiple dispatching is far from being widely accepted, however. Multiple dispatching languages are generally more complex than single dispatching languages, and their implementations have not traditionally been very efficient. In practice, the biggest single concern to the programmers is however that languages supporting multiple dispatching do not "feel object-oriented." This feeling reflects a basic difference in the programming styles encouraged by single and multiple dispatching systems [Chambers 1992]. When using a single dispatching language, the programmer mentally focuses on defining abstract data types (ADTs), associating operations with the data types for which they are implemented. These operations are "contained" in the data type, and together the operations and the data form an encapsulated whole. A whole design and implementation methodology has been developed around this ADT-oriented programming style.

Multiple dispatching languages do not provide much linguistic support for the ADT-oriented programming style. Since multimethods are dispatched on multiple arguments, they are not contained in any single class, and the ADT-oriented view with strong encapsulation breaks down. In practice, the programming style encouraged by multiple dispatching systems seems to be closer to functional, or traditional procedural programming style than ADT style. To many advocates of object-oriented programming, multimethods—having originated from Lisp-based systems—seem like an ad hoc solution that is a consequence of trying to fit round objects to the square world of functions. This feeling is reinforced by the fact that multimethods do not "look object-oriented" either. In multiple dispatching languages the use of the object-oriented *object.message*(*parameters*) notation is discouraged, and the traditional *function*(*parameters*) or Lisp-like (*function param1 . . . paramN*) notation is used instead. Although some researchers have proposed (*receiver1, receiver2, . . . , receiverN*).*message* notation as a compromise, these notations are not as well suited to ADT programming style as the normal dot notation or its equivalents.

Recently Chambers, one of the original implementors of the Self language [Ungar and Smith 1987], has spoken strongly in favor of multimethods [Chamber 1992]. One of Chambers'

main arguments is that problems with multiple dispatching arise mainly from the limitations of text-based programming, and by providing a good graphical programming environment, most of the problems with multimethods disappear. Instead of placing multimethods outside of any single class, or by duplicating the same definition in every participating class, a good visual browser would automatically display the same multimethod in association with all its classes. Intuitively, this sounds like a good idea, but it still remains to be seen whether multiple dispatching really catches on in the object-oriented community.

3.10 Classifying Inheritance Mechanisms

Earlier in the article we saw that while inheritance is a simple linguistic mechanism with relatively straightforward semantics from the theoretical point of view, many variations of inheritance exist that tend to make the analysis of inheritance difficult in practice. In addition to basic questions—such as whether the system is based on classes or prototypes, or whether it supports single or multiple inheritance—there is a large number of more elementary issues that have a considerable impact on the semantics and pragmatics of the system. When implemented in concrete object-oriented programming languages, typically these mechanisms tend to interfere with each other, making the analysis and comparison of the languages complicated.

In trying to understand the concrete realizations of inheritance in object-oriented programming languages, it is useful to use a *progressive* approach, i.e., to start from conceptually simpler issues and then gradually progress towards more complicated aspects. The following kind of three-level taxonomy has proved helpful in analyzing inheritance in object-oriented programming languages.

(1) *Inheritance as incremental modification.* As discussed in Section 2.3.1, the "core" of inheritance is incremental modification. If this criterion is not fulfilled, then we are not talking about inheritance in the object-oriented sense.
(2) *Interface inheritance.* At this level, inheritance is viewed as an interface combination or name space combination mechanism. Questions pertaining to the management of the actual properties (methods and variables) of objects are ignored.
(3) *Property inheritance.* The actual properties (methods and variables) of objects are included in the analysis.

Figure 14 gives an overview of the issues and sources of variability that arise at each level. For a more detailed treatment on the subject, refer to Taivalsaari [1993c].

4. CONCLUSION

Broadly speaking, the history of software development is the history of ever-later binding time.

—*Encyclopedia of Computer Science*

In this article we have surveyed the notion of inheritance, examining its intended and actual usage, its essence and its varieties. We realized that although inheritance was originally introduced as a mechanism to support conceptual specialization, in reality many alternative roles for inheritance can be recognized, ranging from conceptual modeling to implementation-level usage (e.g., using inheritance simply as a code sharing mechanism to save storage space or programming effort). Also, we pointed out that there is no single, commonly accepted model of inheritance, but a lot of variations of the same basic theme exist.

The fundamental observation underlying object-oriented inheritance mechanisms is that they all are essentially *incremental modification mechanisms*, i.e., mechanisms that allow existing programs to be extended and refined without editing existing code. In general, inheritance can be defined gener-

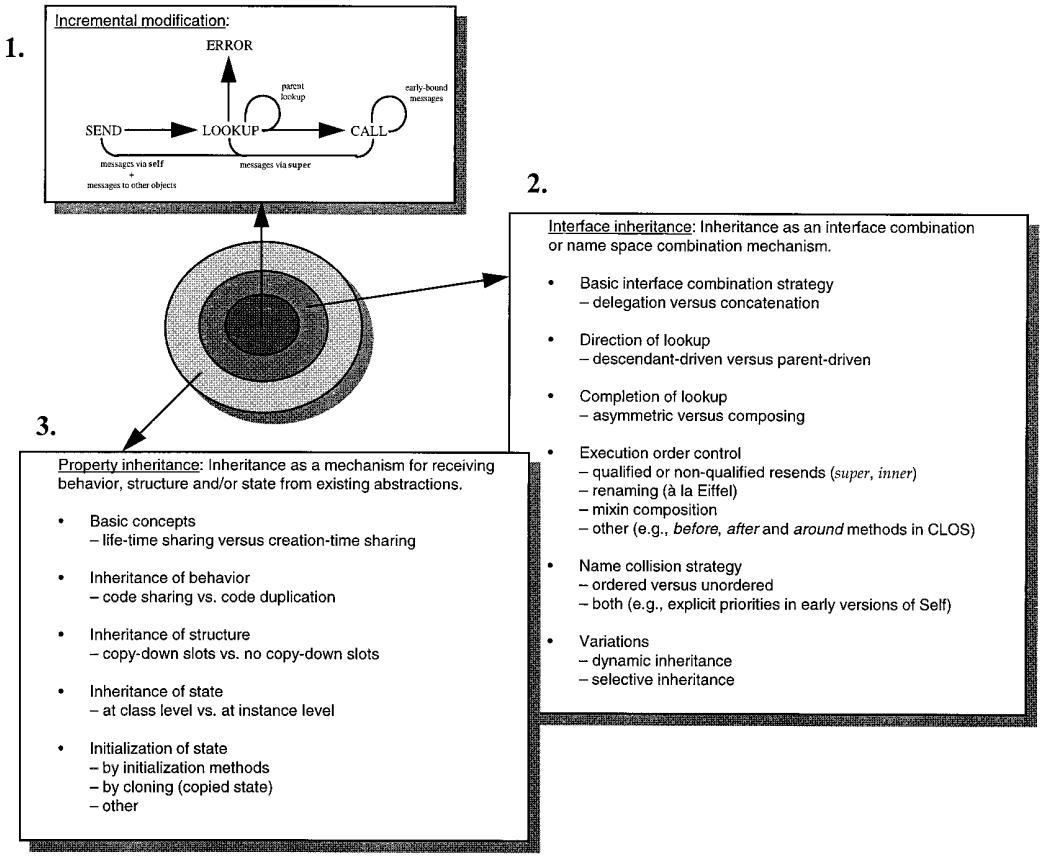

Figure 14. A simple taxonomy for analyzing inheritance mechanisms.

ally as *an incremental modification mechanism in the presence of late-bound self-reference*. Although object-oriented programming has many other benefits compared to other programming paradigms, none of them seems as profound as the incremental capability. To conclude, let us quote Stein et al. [1988] who have noted that "the true value of object-oriented techniques as opposed to conventional programming techniques is not that they can do things the conventional techniques can't, but that they can often extend behavior by adding new code in cases where conventional techniques would require editing existing code instead." This captures the heart of the matter.

REFERENCES

AKSIT, M. AND TRIPATHI, A. 1988. Data abstraction mechanisms in Sina/st. In *OOPSLA'88 Conference Proceedings* (San Diego, California, Sept. 25–30). *ACM SIGPLAN Not. 23,* 11 (Nov.), 267–275.

AMERICA, P. 1987. Inheritance and subtyping in a parallel object-oriented language. In *ECOOP '87: European Conference on Object-Oriented Programming* (Paris, France, June 15–17). Lecture Notes in Computer Science 276, Springer-Verlag, 234–242.

AMERICA, P. 1990. Designing an object-oriented programming language with behavioural subtyping. In *Proceedings of the Foundations of Object-Oriented Languages, REX School/Workshop*. J. W. De Bakker, W. P. De Roever, G. Rozenberg, Eds. (Noordwijkerhout, The Netherlands, May 28–June 1). Lecture Notes in Computer Science 489, Springer-Verlag, 1991, 60–90.

BLASCHEK, G. 1991. Type-safe OOP with prototypes: the concepts of Omega. *Structured Program. 12,* 12 (Dec.) 1–9.

BLASCHEK, G. 1994. *Object-Oriented Programming with Prototypes.* Springer-Verlag.

BOBROW, D. G., KAHN, K., KICZALES, G., MASINTER, L., STEFIK, M. AND ZDYBEL, F. 1986. CommonLoops: merging Lisp and object-oriented programming. In *OOPSLA'86 Conference Proceedings* (Portland, Oregon, Sept. 26–Oct. 2). *ACM SIGPLAN Not. 21,* 11 (Nov.), 17–29.

BORGIDA, A., MYLOPOULOS, J. AND WONG, H. K. T. 1984. Generalization/specialization as a basis for software specification. In *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages,* M. L. Brodie, J. Mylopoulos, J. W. Schmidt, Eds. Springer-Verlag, 87–117.

BORNING, A. H. AND O'SHEA, T. 1987. Deltatalk: an empirically and aesthetically motivated simplification of the Smalltalk-80 language. In *ECOOP'87: European Conference on Object-Oriented Programming* (Paris, France, June 15–17). Lecture Notes in Computer Science 276, Springer-Verlag, 1–10.

BORNING, A. H. 1986. Classes versus prototypes in object-oriented languages. In *Proceedings of ACM/IEEE Fall Joint Computer Conference,* (Nov.) 36–40.

BRACHA, G. 1992. The programming language Jigsaw: Mixins, modularity and multiple inheritance. Ph.D. thesis, Univ. of Utah, March.

BRACHA, G. AND COOK, W. 1990. Mixin-based inheritance. In *OOPSLA/ECOOP'90 Conference Proceedings* (Ottawa, Canada, Oct. 21–25). *ACM SIGPLAN Not. 25,* 10 (Oct.), 303–311.

BRACHA, G. AND LINDSTROM, G. 1992. Modularity meets inheritance. In *Proceedings of the 1992 International Conference on Computer Languages* (Oakland, California, April 20–23), IEEE Computer Society Press, 282–290.

BRACHMAN, R. 1983. What Is-a is and isn't? *IEEE Comput. 16,* 10 (Oct.) 30–36.

BRACHMAN, R. 1985. I lied about the trees—or, defaults and definitions in knowledge representation. *AI Magazine 6,* 3 (Fall) 80–93.

BRODIE, M. L. 1983. Association: a database abstraction for semantic modelling. In *Entity-Relationship Approach to Information Modeling and Analysis.* P. P. Chen, Ed. Elsevier Science Publishers (North-Holland), 577–602.

CARDELLI, L. 1984. A semantics of multiple inheritance. In *Semantics of Data Types*, G. Kahn, D. B. MacQueen, G. Plotkin, Eds. Lecture Notes in Computer Science 173, Springer-Verlag, 51–67.

CHAMBERS, C. 1992. Object-oriented multi-methods in Cecil. In *ECOOP'92: European Conference on Object-Oriented Programming* (Utrecht, The Netherlands, June 29–July 3). Lecture Notes in Computer Science 615, Springer-Verlag, 33–56.

CHAMBERS, C. 1993. Predicate classes. In *ECOOP'93: European Conference on Object-Oriented Programming* (Kaiserslautern, Germany, July 26–30). Lecture Notes in Computer Science 707, Springer-Verlag, 268–296.

CHAMBERS, C., UNGAR, D., CHANG, B.-W. AND HOLZLE, U. 1991. Parents are shared parts of objects: inheritance and encapsulation in Self. *Lisp Symbolic Comput. 4,* 3 (Jun.).

COOK, W. R., HILL, W. L. AND CANNING, P. S. 1990. Inheritance is not subtyping. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages* (San Francisco, California, Jan. 17–19), ACM Press, 125–135.

COOK, W. R. 1989. A denotational semantics of inheritance. Ph.D. thesis, Brown University, Tech. Rep. CS-89-33, May.

COOK, W. R. 1989. A proposal for making Eiffel type-safe. *Comput. J. 32,* 4 (Aug.), 305–311. Also in *ECOOP'89: Proceedings of the Third European Conference on Object-Oriented Programming* (Nottingham, England, July 10–14). The British Computer Society Workshop Series, Cambridge, Univ. Press, 57–70.

COOK, W. R. 1992. Interfaces and specifications for the Smalltalk-80 collection classes. In *OOPSLA'92 Conference Proceedings* (Vancouver, Canada, Oct. 18–22). *ACM SIGPLAN Not. 27,* 10 (Oct.), 1–15.

COOK, W. R. AND PALSBERG, J. 1989. A denotational semantics of inheritance and its correctness. In *OOPSLA'89 Conference Proceedings* (New Orleans, Louisiana, Oct. 1–6). *ACM SIGPLAN Not. 24,* 10 (Oct.), 433–443.

DAHL, O-J., DIJKSTRA, E. W. AND HOARE, C. A. R. 1972. *Structured Programming.* Academic Press.

DAHL, O-J., MYHRHAUG, B. AND NYGAARD, K. 1968. SIMULA 67 common base language. Tech. Rep., Norwegian Computing Center, Oslo, May.

DANFORTH, S. AND TOMLINSON, C. 1988. Type theories and object-oriented programming. *ACM Comput. Surv. 20,* 1 (Mar.) 29–72.

DEMICHIEL, L. G. AND GABRIEL, R. P. 1987. The Common Lisp object system: An overview. In *ECOOP'87: European Conference on Object-Oriented Programming* (Paris, France, June 15–17). Lecture Notes in Computer Science 276, Springer-Verlag, 157–170.

DEUTSCH, L. P. AND SCHIFFMAN, A. M. 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages* (Salt Lake City, Utah, Jan. 15–18), ACM Press, 297–302.

DONY, C., MALENFANT, J. AND COINTE, P. 1992. Prototype-based languages: from a new taxon-

omy to constructive proposals and their validation. In *OOPSLA'92 Conference Proceedings* (Vancouver, Canada, Oct. 18–22). *ACM SIGPLAN Not. 27,* 10 (Oct.), 201–217.

DRIESEN, K., HÖLZLE, U. AND VITEK, J. 1995. Message dispatch on pipelined processors. In *ECOOP'95: European Conference on Object-Oriented Programming* (Aarhus, Denmark, Aug. 7–11). Lecture Notes in Computer Science. 952, Springer-Verlag, 253–282.

DUCOURNAU, R. AND HABIB, M. 1987. On some algorithms for multiple inheritance in object-oriented programming. In *ECOOP'87: European Conference on Object-Oriented Programming* (Paris, France, June 15–17). Lecture Notes in Computer Science 276, Springer-Verlag, 243–252.

ELLIS, M. A. AND STROUSTRUP, B. 1990. *The Annotated C++ Reference Manual.* Addison-Wesley.

GOLDBERG, A. AND ROBSON, D. 1989. *Smalltalk-80: The Language.* Addison-Wesley.

GRAVER, J. O. AND JOHNSON, R. E. 1990. A type system for Smalltalk. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages* (San Francisco, California, Jan. 17–19), ACM Press, 136–150.

HALBERT, D. C. AND O'BRIEN, P. D. 1987. Using types and inheritance in object-oriented programming. *IEEE Softw. 4,* 5 (Sept.), 71–79. Revised version in *ECOOP'87: European Conference on Object-Oriented Programming* (Paris, France, June 15–17). Lecture Notes in Computer Science 276, Springer-Verlag, 20–31.

HARPER, R. AND PIERCE, B. 1991. A record calculus based on symmetric concatenation. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages* (Orlando, Florida, Jan. 21–23), ACM Press, 131–142.

HENDLER, J. 1986. Enhancement for multiple inheritance. In *OOPSLA'86 Conference Proceedings* (Portland, Oregon, Sept. 26–Oct. 2). *ACM SIGPLAN Not. 21,* 11 (Nov.), 98–106.

HARTMANN, T., JUNGCLAUS, R. AND SAAKE, G. 1992. Aggregation in a behavior oriented object model. In *ECOOP'92: European Conference on Object-Oriented Programming* (Utrecht, The Netherlands, June 29–July 3). Lecture Notes in Computer Science 615, Springer-Verlag, 57–77.

JOHNSON, R. E. 1986. Type-checking Smalltalk. *OOPSLA'86 Conference Proceedings* (Portland, Oregon, Sept. 26–Oct. 2). *ACM SIGPLAN Not. 21,* 11 (Nov.), 315–321.

KEENE, S. E. 1989. *Object-Oriented Programming is Common Lisp: A Programmer's Guide to CLOS.* Addison-Wesley.

KRISTENSEN, B. B., MADSEN, O. L., MØLLER-PEDERSEN, B. AND NYGAARD, K. 1983. Abstraction mechanisms in the Beta programming language. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages* (Austin, Texas, Jan. 24–26), ACM Press, 285–298.

KNUDSEN, J. L. AND MADSEN, O. L. 1988. Teaching object-oriented programming is more than teaching object-oriented programming languages. In *ECOOP'88: European Conference on Object-Oriented Programming* (Oslo, Norway, Aug. 15–17). Lecture Notes in Computer Sci. 276, Springer-Verlag, 21–40.

KNUDSEN, J. L. 1988. Name collision in multiple classification hierarchies. In *ECOOP'88: European Conference on Object-Oriented Programming* (Oslo, Norway, Aug. 15–17). Lecture Notes in Computer Sci. 276, Springer-Verlag, 93–109.

KORSON, T. AND MCGREGOR, J. D. 1990. Understanding object-oriented: A unifying paradigm. *Commun. ACM 33,* 9 (Sept.) 40–60.

KRUEGER, C. W. 1992. Software reuse. *ACM Comput. Surv. 24,* 2 (June) 131–183.

LALONDE, W. R. 1989. Designing families of data types using exemplars. *ACM Trans. Program. Lang. Syst. 11,* 2 (Apr.) 212–248.

LALONDE, W. R. AND PUGH, J. 1991. Subclassing ≠ subtyping ≠ is-a. *J. Object-Oriented Program. 3,* 5 (Jan.), 57–62.

LALONDE, W. R., THOMAS, D. A. AND PUGH, J. R. 1986. An exemplar based Smalltalk. In *OOPSLA'86 Conference Proceedings* (Portland, Oregon, Sept. 26–Oct. 2). *ACM SIGPLAN Not. 21,* 11 (Nov.), 322–330.

LANG, K. J. AND PEARLMUTTER, B. A. 1986. Oaklisp: an object-oriented Scheme with first class types. In *OOPSLA'86 Conference Proceedings* (Portland, Oregon, Sept. 26–Oct. 2). *ACM SIGPLAN Not. 21,* 11 (Nov.), 30–37.

LIEBERHERR, K. J., HOLLAND, I. M. AND RIEL, A. J. 1988. Object-oriented programming: an objective sense of style. In *OOPSLA'88 Conference Proceedings* (San Diego, California, Sept. 25–30). *ACM SIGPLAN Not. 23,* 11 (Nov.) 323–334.

LIEBERHERR, K. J. AND HOLLAND, I. M. 1989. Assuring good style for object-oriented programming. *IEEE Softw. 6,* 5 (Sept.) pp. 38–48.

LIEBERMAN, H. 1986. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPSLA'86 Conference Proceedings* (Portland, Oregon, Sept. 26–Oct. 2). *ACM SIGPLAN Not. 21,* 11 (Nov.), 214–223.

LISKOV, B. H. 1987. Data abstraction and hierarchy. In *Addendum to the Proceedings of OOPSLA'87. ACM SIGPLAN Not. 23,* 5 (May), 1988, 17–34.

LOOMIS, M. E. S., SHAH, A. V. AND RUMBAUGH, J. E. 1987. An object modeling technique for conceptual design. In *ECOOP'87: European Conference on Object-Oriented Programming* (Paris, France, June 15–17). Lecture Notes in

Computer Science 276, Springer-Verlag, 192–202.

MADSEN, O. L. AND MØLLER-PEDERSEN, B. 1988. What object-oriented programming may be and what it does not have to be? In *ECOOP'88: European Conference on Object-Oriented Programming* (Oslo, Norway, Aug. 15–17). Lecture Notes in Computer Science. 276, Springer-Verlag, 1–20.

MADSEN, O. L. AND MØLLER-PEDERSEN, B. 1989. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA'89 Conference Proceedings* (New Orleans, Louisiana, Oct. 1–6). *ACM SIGPLAN Not. 24,* 10 (Oct.), 397–406.

MADSEN, O. L., MAGNUSSON, B. AND MØLLER-PEDERSEN, B. 1990. Strong typing of object-oriented programming revisited. In *OOPSLA/ECOOP'90 Conference Proceedings* (Ottawa, Canada, Oct. 21–25). *ACM SIGPLAN Not. 25,* 10 (Oct.), 140–149.

MAGNUSSON, B. 1991. Implementation of inheritance in Simula. Personal communication.

MARCOTTY, M. AND LEDGARD, H. 1987. *The World of Programming Languages.* Springer-Verlag.

MATTOS, N. M. 1988. Abstraction concepts: the basis for knowledge modeling. In *Proceedings of the 7th International Conference on Entity-Relationship Approach* C. Batini, Eds. Rome, (Nov. 16–18), 331–350.

MEYER, B. 1988. *Object-Oriented Software Construction.* Prentice-Hall.

MOON, D. A. 1986. Object-oriented programming with Flavors. In *OOPSLA'86 Conference Proceedings* (Portland, Oregon, Sept. 26–Oct. 2). *ACM SIGPLAN Not. 21,* 11 (Nov.), 1–8.

NYGAARD, K. AND DAHL, O-J. 1987. The development of the Simula languages. In *ACM SIGPLAN History of Programming Languages Conference*, Ed. (Los Angeles, California, June 1–3), R. L. Wexelblat, *ACM SIGPLAN Not. 13,* 8 (Aug.) 245–272.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1990. Type substitution for object-oriented programming. In *OOPSLA/ECOOP'90 Conference Proceedings* (Ottawa, Canada, Oct. 21–25). *ACM SIGPLAN Not. 25,* 10 (Oct.), 151–159.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1991. Static typing for object-oriented programming. Aarhus University Tech. Rep. DAIMI PB-355, Denmark, June.

PEDERSEN, C. H. 1989. Extending ordinary inheritance schemes to include generalization. In *OOPSLA'89 Conference Proceedings* (New Orleans, Louisiana, Oct. 1–6). *ACM SIGPLAN Not. 24,* 10 (Oct.), 407–417.

PERNICI, B. 1990. Object with roles. In *Proceedings of the ACM/IEEE Conference of Office Information Systems* (Cambridge, Massachusetts, Apr. 25–27), F. H. Lochovsky and R. B. Allen, Eds. *ACM SIGOIS Bull. 11,* 2/3 (Apr.) 205–215.

PLATO. *The Republic.* Kustannusosakeyhtiö Otava, Keuruu, Finland. (Finnish translation, 1981).

PORTER, H. H. III. 1992. Separating the subtype hierarchy from the inheritance of implementation. *J. Object-Oriented Program. 4,* 9 (Feb.) 20–29.

RAJ, R. K. AND LEVY, H. M. 1989. A compositional model for software reuse. *The Comput. J. 32,* 4 (Aug.) 312–322.

REDDY, U. S. 1988. Objects as closures: abstract semantics of object-oriented languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (Snowbird, Utah, July 25–27), 289–297.

RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F. AND LORENSEN, W. 1991. *Object-Oriented Modeling and Design*. Prentice-Hall.

SAKKINEN, M. 1989. Disciplined inheritance. In *ECOOP'89: Proceedings of the Third European Conference on Object-Oriented Programming* (Nottingham, England, July 10–14). The British Computer Society Workshop Series, Cambridge University Press, 39–56.

SCHAFFERT, C., COOPER, T., BULLIS, B., KILLIAN, M. AND WILPOLT, C. 1986. An introduction to Trellis/Owl. In *OOPSLA'86 Conference Proceedings* (Portland, Oregon, Sept. 26–Oct. 2). *ACM SIGPLAN Not. 21,* 11 (Nov.), 9–16.

SCIORE, E. 1989. Object specialization. *ACM Trans. Inf. Syst. 7,* 2 (April) 103–122.

SMITH, J. M. AND SMITH, D. C. P. 1977. Database abstractions: aggregation. *Commun. ACM 20,* 6 (Jun.) 405–413.

SMITH, J. M. AND SMITH, D. C. P. 1977. Database abstractions: aggregation and generalization. *ACM Trans. Database Syst. 2,* 2 (Jun.) 105–133.

SMITH, D. C. P. AND SMITH, J. M. 1980. Conceptual database design. Also in *Tutorial on Software Design Techniques*, 4th ed., Freeman, P., Wasserman, A. I. Eds. IEEE Computer Society Press, 1983, 437–460.

SNYDER, A. 1986. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA'86 Conference Proceedings,* (Portland, Oregon, Sept. 26–Oct. 2). *ACM SIGPLAN Not. 21,* 11 (Nov.), 38–45.

SNYDER, A. 1986. CommonObjects: an overview. In *ACM SIGPLAN Not. 21,* 10 (Oct.), 19–28.

SNYDER, A. 1991. Modeling the C++ object model: an application of an abstract object model. In *ECOOP'91: European Conference on Object-Oriented Programming* (Geneva, Switzerland, July 15–19). Lecture Notes in Computer Science. 512, Springer-Verlag, 1–20.

SOWA, J. F. 1984. *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley.

STEIN, L. A. 1987. Delegation is inheritance. In *OOPSLA'87 Conference Proceedings* (Orlando,

Florida, Oct. 4–8). *ACM SIGPLAN Not. 22,* 12 (Dec.), 138–146.

STEIN, L. A., LIEBERMAN, H. AND UNGAR, D. 1988. A shared view of sharing: the treaty of Orlando. In *Object-Oriented Concepts, Applications and Databases*, W. Kim and F. Lochowsky, Eds. Addison-Wesley, 31–48.

STEIN, L. A. 1989. Towards a unified method of sharing in object-oriented programming. In *Workshop on Inheritance Hierarchies in Knowledge Representation and Programming* (Viareggio, Italy, Feb. 6–8).

TAIVALSAARI, A. 1993. On the notion of object. *J. Syst. Softw. 21,* 1 (Apr.) 3–16.

TAIVALSAARI, A. 1993. Object-oriented programming with modes. *J. Object-Oriented Program. 6,* 3 (Jun.) 25–32.

TAIVALSAARI, A. 1993. A critical view of inheritance and reusability in object-oriented programming. Ph.D. thesis, Jyväskylä Studies in Computer Science, Economics and Statistics 23, Univ. of Jyväskylä, Finland, Dec. 276 pages.

UNGAR, D. AND SMITH, R. B. 1987. Self: the power of simplicity. In *OOPSLA'87 Conference Proceedings* (Orlando, Florida, Oct. 4–8). *ACM SIGPLAN Not. 22,* 12 (Dec.), 227–241.

WEGNER, P. 1987. The object-oriented classification paradigm. In *Research Directions in Object-Oriented Programming.* B. Shriver and P. Wegner, Eds. The MIT Press, 479–560.

WEGNER, P. 1990. Concepts and paradigms of object-oriented programming. *ACM OOPS Messenger 1,* 1 (Aug.) 7–87.

WEGNER, P. AND SHRIVER, B. Eds. 1986. Object-oriented programming workshop (Yorktown Heights, New York, June 9–13). *ACM SIGPLAN Not. 21,* 10 (Oct.).

WEGNER, P. AND ZDONIK, S. B. 1988. Inheritance as an incremental modification mechanism or what Like is and isn't like. In *ECOOP'88: European Conference on Object-Oriented Programming* (Oslo, Norway, Aug. 15–17). Lecture Notes in Computer Sci. 276, Springer-Verlag, 55–77.

WILKES, W. 1988. Instance inheritance mechanisms for object-oriented databases. In *Advances in Object-Oriented Databased Systems, 2nd International Workshop on Object-Oriented Database Systems,* (Bad Munster am Stein-Ebernburg, Germany, Sept. 27–30), K. R. Dittrich, Ed. Lecture Notes in Computer Science 334, Springer-Verlag, 274–279.

WIRTH, N. 1971. Program development by stepwise refinement. *Commun. ACM 14,* 4 (Apr.) 221–227.

WIRTH, N. 1985. *Programming in Modula-2.* 3rd Ed. Springer-Verlag.

ZDONIK, S. B. 1986. Why properties are objects, or some refinements of "is-a." In *Proceedings of ACM/IEEE Fall Joint Computer Conference,* (Nov.) 41–47.

Received April 1994; revised September 1995; accepted January 1996