Abstract

The concepts of inheritance and subtyping are adding new dimensions to programming language design and use, but they are not as well understood as longer established ideas such as data abstraction and modularity. Confusion about these concepts leads to language-design problems, such as designs that are deficient with respect to either or both concepts, that result in overlap or interference between the implementations of the two concepts, or that are poorly understood or described. Language-design problems in turn lead to language-usage problems, such as uncertainty about how to use inheritance and subtyping, uncertainty about which mechanism to use, and conflicting styles of use within a software system.

In this paper we attempt to clarify the issues surrounding inheritance and subtyping in programming languages. Our approach to this task is based on a belief that inheritance and subtyping must be treated as separate, although not independent, aspects of language design. We begin by offering definitions of inheritance and subtyping that help in characterizing the relationship between these concepts. In light of these definitions, we briefly review the variety of inheritance and subtyping mechanisms found in modern programming languages. We then advance a notion that we call *behavioral abstraction*. Based upon that notion, we propose and discuss a number of pragmatic principles of programming-language design regarding inheritance and subtyping. We examine the extent to which a number of existing programming languages adhere to these principles and, finally, speculate on how the principles, and the notion of behavioral abstraction in general, might affect future languages and the language-design process.

Contents

1 Introduction

Inheritance, as a programming language mechanism, has its origins in SIMULA [Dahl and Nygaard, 1966], where it was intended to aid in hierarchical system decomposition. A SIMULA system typically is composed of several *classes*, which are simply encapsulators of program-entity declarations such as those for variables and procedures. A class serves as a template from which particular instances, called *objects*, are created. Indeed, a class bears a resemblance to type constructors found in more recent programming languages.

In SIMULA, a class can be defined independently of other classes or it can be defined as a *subclass* of another, so-called *superclass*. In the latter case, the subclass "inherits" all the program-entity declarations of the superclass, implicitly encapsulating a set of declarations identical to those of its superclass. Inheritance is in fact transitive, so that a subclass inherits declarations not only from its immediate superclass, but also from the superclass of its superclass, and so on. A subclass may also include additional declarations of its own, which distinguish objects of the subclass from objects of the superclass.

Consider, for example, the development of a sequence abstraction in SIMULA. We can define a class for sequence elements that embodies the notion of "next element" by encapsulating in that class a declaration for a pointer to a sequence element together with the declarations of procedures representing an appropriate set of abstract operations on sequence elements. Every instance of the class, i.e., every sequence element, will thus have a pointer to the next element and can be operated on by the procedures. Notice, however, that we have made no mention of the kind of data that can be put into sequence. The reason is that any kind of data can be put into sequence as long as the class defining that kind is a subclass of the class for sequence elements. For instance, if we want a sequence of objects that store employee information, then we simply make the class for employee information a subclass of the class for sequence elements; through inheritance, employee-information objects would gain next-element pointers and procedures.

The hierarchical decomposition that results from the use of inheritance in a SIMULA system is actually a tree or, more accurately, a forest of subclasses. The design method engendered by such an approach to system decomposition encourages the identification of common, general-purpose entity declarations and isolation of those declarations into classes high in the hierarchy. This is what occurs with the notion of "next element" in the

sequence example.

The SIMULA inheritance mechanism has strongly influenced the designs of inheritance mechanisms found in many other programming languages. The most familiar of these languages is probably Smalltalk [Goldberg and Robson, 1983], whose inheritance mechanism is essentially the same as SIMULA's. As experience with inheritance has grown, however, various language designs have been developed that seek to find more powerful notions of inheritance, particularly ones based on so-called *multiple inheritance*, which allows a subclass to have more than one immediate superclass. Languages that incorporate these newer inheritance mechanisms include a number of object-oriented extensions to LISP (e.g., Flavors [Moon, 1986], CommonLoops [Bobrow *et al.*, 1985], CommonObjects [Snyder, 1985], and CLOS [Bobrow *et al.*, 1987]), C++ [Stroustrup, 1986; Stroustrup, 1987], Trellis/Owl[1] [Schaffert *et al.*, 1986], and Eiffel[2][Meyer, 1988], as well as succeeding versions of SIMULA and Smalltalk themselves (e.g., [Kristensen *et al.*, 1983; Kristensen *et al.*, 1987; Borning and Ingalls, 1982]).

Subtyping is a recent development in programming languages and is another means for organizing the components of a software system. In essence, *subtype* is a relationship between components that are types. The exact meaning of that relationship varies from language to language, but its purpose is generally universal: to aid in establishing, documenting, and/or enforcing the type structure and type consistency of a system.

The comparable subtyping mechanisms of Emerald [Black *et al.*, 1987] and Trellis/Owl (Owl for short) are good examples. The subtyping mechanisms of these languages are based on the principle that objects of a subtype should exhibit behavior similar to that of objects of a supertype. For example, we would expect that the type Car would be a subtype of the type Vehicle, since a car should be usable anywhere a (generalized) vehicle may be used. This is not to say, however, that the behaviors must be equivalent. In particular, given a type T and given a type S that is a subtype of T, an object of type S should be usable anywhere an object of type T is usable, but the reverse need not be true.

The subtype relationship in Emerald and Owl, like inheritance in SIMULA, is transitive. Moreover, a subtype can have more than one immediate supertype, which means that the behavior of an object of a subtype might be similar to the behaviors of objects of quite

[1]Trellis is a trademark of Digital Equipment Corporation.

[2]Eiffel is a trademark of Interactive Software Engineering, Inc.

different supertypes. Thus, the subtype relationship defines a lattice, as compared to the forest of trees of SIMULA's inheritance mechanism.

Validation of subtype relationships is based on what is known as *conformance*, and is limited by the extent to which we currently understand how to specify and reason about behavior. In Emerald and Owl, for example, specification of behavior amounts to a list of operation signatures, where a signature consists of a name for the operation together with a list of the types of that operation's arguments and results. "Similar behavior" in this setting therefore reduces to *signature conformance*: for a type S to be a subtype of another type T, the signatures of S must conform to those of T. Conformance rules are designed so that treating an S object as a T object, from the standpoint of invoking operations, will never cause an S operation to be invoked with arguments of illegal types nor permit the S operation to return results incompatible with those of a corresponding T operation. In short, with the appropriate conformance rules and a requirement that subtypes conform to supertypes, one can perform type checks using the definition of T and guarantee type correctness for all subtypes of T. In Owl, which provides a special construct for indicating that one type is to be a subtype of another, signature conformance is (statically) checked by comparing the type definitions themselves. Signature conformance in Emerald, on the other hand, is (statically) checked at the site of uses of operations, since Emerald does not provide a construct for explicitly indicating intended subtype relationships.

Various rules of conformance have been proposed [Schaffert *et al.*, 1986; Black *et al.*, 1987; Horn, 1987]. For instance, a common rule is that for every operation of the supertype there must be a corresponding operation of the subtype, the arities of the operations must agree, and the argument and result types must themselves conform appropriately. We would expect that as our ability to formulate mechanically analyzable behavioral specifications becomes more sophisticated, so too would we be able to increase the sophistication of conformance rules.

Unfortunately, the concepts of inheritance and subtyping are easily confused. This leads to a number of language-design problems, such as designs that are deficient with respect to either or both concepts, that result in overlap or interference between the implementations of the two concepts, or that are poorly understood or described. Language-design problems in turn lead to language-usage problems, such as uncertainty about how to use inheritance and subtyping, uncertainty about which mechanism to use, and conflicting styles of use

within a software system.

There appear to be several reasons for the confusion surrounding inheritance and subtyping, including the following, which are discussed in greater detail in subsequent sections of this paper:

- both inheritance and subtyping can be viewed as relationships among "type-like" system components;
- both inheritance and subtyping contribute to software reuse; and
- every inheritance mechanism implies some sort of subtyping mechanism.

The problems have been exacerbated by a conservatism in language design that tends to preserve existing language frameworks. While such conservatism is justified in general on the grounds of lowered intellectual and economic costs, in this case it has led, among other things, to contortions in the designs and descriptions of the older concept of inheritance to fit the newer concept of subtyping. In particular, the two concepts are merged into one mechanism in many programming languages and, partially as a result, the two terms are used interchangeably, adding to the confusion. Another source of terminological confusion is historical: As language designers have switched from the term *class* (e.g., in older languages, such as SIMULA) to the term *type* (e.g., in newer languages, such as Owl), it is natural for some that they also switch from the term *subclass* to the term *subtype* to refer to inheritors. The term *subtype*, therefore, can have different meanings in different languages.[3] In sum, there has yet to arise any widely accepted set of definitions and terminology for inheritance and subtyping. Moreover, there is yet to arise any widely accepted understanding of how these concepts can be smoothly integrated into a programming language.

In this paper we attempt to clarify the issues surrounding inheritance and subtyping in programming languages. Our approach to this task is based on a belief that inheritance and subtyping must be treated as separate, although not independent, aspects of language design. We begin by offering definitions of inheritance and subtyping that help in characterizing the relationship between these concepts, and give examples to illustrate. This is followed by a brief review of the variety of inheritance and subtyping mechanisms

[3]In this paper we generally use the terms *subclass* and *superclass* in the context of inheritance, and the terms *subtype* and *supertype* in the context of subtyping.

found in modern programming languages. We then advance a notion that we call *behavioral abstraction*. Based upon that notion, we propose and discuss a number of pragmatic principles of programming-language design regarding inheritance and subtyping and examine the extent to which a number of existing programming languages adhere to those principles. We conclude by speculating on how the principles, and the notion of behavioral abstraction in general, might affect future languages and the language-design process. As a convenience to the reader and to avoid any possible misunderstanding, we include as an appendix a glossary of terms used in this paper.

2 Defining and Characterizing Inheritance and Subtyping

A substantial portion of the confusion surrounding inheritance and subtyping can be attributed to the lack of concise, language-independent definitions of these concepts. We therefore offer the following definitions:

Inheritance: a means by which new system components can be constructed from old system components such that changes to the definitions of the old components can have an effect, subject to certain constraints, on the definitions of the new components.

Subtyping: a means by which the behavior of one object can be established or asserted as being similar to the behavior of another object such that the first object can be used, subject to certain constraints, in place of the second object.

It should be clear from its definition that inheritance is quite different from other techniques for constructing the components of a software system. In particular, the fact that a change to a component can have an effect on other components that inherit from the changed component implies that there must be language or environment support for maintaining this structural relationship. Compare this to a technique based, say, on a text editor. Using an editor, one would copy (portions of) a component's definition for use in the definition of another component. Copying suggests that no connection with the original is maintained.

Consider the sequence example discussed in the previous section. Following are skeletons of C++ definitions for the two classes, where the declarations inherited from Se-

quenceElement by EmployeeInformation are shown as comments:

```
class SequenceElement {
public:
  void                Insert ( . . . ) { . . . };
  SequenceElement* GetNext ( . . . ) {return NextElement};
  . . .
private:
  SequenceElement* NextElement;
};
```

```
class EmployeeInformation : public SequenceElement {
public:
  /* void                Insert ( . . . ) { . . . };
   * SequenceElement* GetNext ( . . . ) {return NextElement};
   * . . .                                                    */
  void SetStartDate ( . . . ) { . . . };
  . . .
private:
  /* SequenceElement* NextElement; */
  int StartDate;
  . . .
};
```

If we were to change the definition of SequenceElement so that it additionally supported the notion of a "previous element" by having a declaration for another pointer, having a declaration for a function to retrieve the previous element, and appropriately adjusting the other functions to account for the new pointer, then any subclass of SequenceElement, such as EmployeeInformation, would "automatically" obtain this additional functionality without itself requiring any changes.

Notice that there is a subtle difference between inheritance and the simple use, by reference, of an existing definition in the definition of a component. Inheritance results in the *concatenation* of declarations rather than reference to previously declared entities. For example, since EmployeeInformation inherits from SequenceElement, EmployeeInformation encapsulates declarations equivalent to the declarations encapsulated by SequenceElement plus additional ones specific to employee information. The distinction we are trying to draw

is illustrated by the following version of EmployeeInformation, which uses SequenceElement by reference instead of by inheritance:

```
class EmployeeInformation {
public:
  void                Insert ( . . . ) { . . . };
  SequenceElement* GetNext ( . . . ) {return Link.NextElement};
  . . .
  void SetStartDate ( . . . ) { . . . };
  . . .
private:
  SequenceElement Link;
  int StartDate;
  . . .
};
```

In this version, the sequence operations must be specially written for EmployeeInformation to operate on the variable (called a *member* in C++) Link. The point here is that an instance of a subclass does not consist of an instance of a superclass plus additions. Rather, an instance of a subclass is a new kind of entity that is associated with all the declarations of the superclass and possibly more.[4]This does not prevent one from treating an instance of a subclass as an instance of a superclass when it is convenient to do so (i.e., when the subclass is also a subtype; see below). It is simply that it is inappropriate to consider an instance of a superclass as a "part" or "piece" of an instance of a subclass. One effect of this is that if we wish, for example, to have employee-information objects simultaneously be elements of two sequences, then we cannot simply have its class inherit from the class for sequence elements "twice":

```
class EmployeeInformation : public SequenceElement,
                            public SequenceElement { /* ERROR */
  . . .
};
```

[4]There is debate as to whether the mechanism of *derived types* in the programming language Ada constitutes a genuine inheritance mechanism. We believe that it does, according to our definition. Nonetheless, the mechanism is one that has a significant limitation: a subclass, called a "derived type" in Ada, cannot add new (instance) variable declarations to the set of inherited declarations.

The result would give rise to ambiguities; which sequence are we referring to when we call Insert? Eiffel provides a special syntactic mechanism to deal with this problem, but it is one that essentially transforms inheritance into a reference mechanism.

Turning now to the definition of subtyping, it is interesting to notice that the definition is stated exclusively in terms of objects rather than types, especially since the discussion of subtyping in Section 1 centers primarily on types. As we discuss more fully below, we consider a type to be a *behavioral abstraction* of the objects that are its instances. For example, Owl definitions of a type for vehicles and a type for cars might look like the following:

```
typemodule Vehicle;
  operation GetLoad           (me)                  returns (Integer) is . . .;
  operation ChangeLoad        (me, Delta: Integer) returns (Integer) is . . .;
  operation GetNumberOfAxles (me)                  returns (Integer) is . . .;
  operation ForcePerAxle      (me)                  returns (Real) is . . .;
end typemodule Vehicle;


typemodule Car;
  subtypeof (Vehicle);
  operation GetLoad           (me)                  returns (Integer) is . . .;
  operation ChangeLoad        (me, Delta: Integer) returns (Integer) is . . .;
  operation GetNumberOfAxles (me)                  returns (Integer) is . . .;
  operation ForcePerAxle      (me)                  returns (Real) is . . .;

  operation GetMaxPassengers  (me)                  returns (Integer) is . . .;
  operation GetPassengers     (me)                  returns (Integer) is . . .;
  operation ChangePassengers  (me, Delta: Integer) returns (Integer) is . . .;
end typemodule Car;
```

As mentioned in the previous section, specification of behavior in Owl amounts to a list of operation signatures. The definition of type Car contains the assertion that Car is a subtype of Vehicle. The Owl compiler can verify that this is indeed true with respect to the conformance of the operation signatures; notice that the operations of Car are the operations of Vehicle plus the operations GetMaxPassengers, GetPassengers, and ChangePassengers.[5]Thus,

[5]Owl is a language that supports both inheritance and subtyping. For purposes of this example, however, we did not take advantage of its inheritance mechanism, which would have allowed us to avoid duplicating the definitions of the Vehicle operations within Car.

objects of type Car can be used anywhere objects of type Vehicle can be used, since the operations (i.e., behavior) of Vehicle are subsumed by the operations of Car. This is true even though the implementation of Car (e.g., the code of the conforming operations) might be quite different.

Because "subtypeness" is a legitimate aspect of the specification of behavior, languages that support subtyping and explicit type declaration, such as Owl, typically also support the expression of the subtype relationship as part of type definitions, as seen above. There are exceptions, however, such as Emerald, which has a type-declaration facility but no subtype-indication construct; the Emerald subtype relationship is implicit and is in some sense dependent upon how objects are actually used in a system. Moreover, although we know of no examples, it is conceivable for a language to support subtyping but not have an explicit type-declaration facility. (The preliminary work on enhancing the ML language with features for object-oriented programming, such as that reported in [Jategaonkar and Mitchell, 1988], is perhaps pointing in this direction.) Thus, to account for all these cases, our definition is cast in the more general terms of objects.

In the remainder this section, we use our definitions of inheritance and subtyping to help characterize the relationship between these concepts and demonstrate both their independent and interdependent application. In addition, we examine how inheritance and subtyping bear on the effectiveness of abstraction. In the next section, we explore the variety of actual inheritance and subtyping mechanisms — that is, the various ways program entities can be inherited and the various subtype relationships that can be established or asserted among objects. In the course of that exploration, we solidify what we mean by "similar behavior".

2.1 The Relationship Between Inheritance and Subtyping

The definitions given above reflect a very basic difference in the domains to which these concepts pertain. In particular, inheritance is concerned with the *construction* of a system, while subtyping is concerned with the *behavior* of a system. Clearly, the behavior of a system is related to how that system is constructed. This relationship, however, is analogous to that between implementation and specification. In other words, a given construction implies a particular behavior just as a given implementation possesses an implicit specification, but a given behavior can be constructed in a number of ways just

as a given specification can have a variety of implementations. Inheritance is only one method that can be employed to construct a software system, and subtype is only one relationship that can possibly result from the use of inheritance.

Because inheritance is primarily concerned with construction and subtyping with behavior, another useful analogy can be drawn: inheritance is to subtyping as syntax is to semantics. A syntactic description of a system captures how that system is constructed, while a semantic description captures behavior. And while two syntactically identical constructions (should) exhibit the same behavior, two syntactically different ones might also exhibit the same behavior. The point is that knowledge of how entities are constructed is not necessarily sufficient or relevant when reasoning about the behavioral relationships among those entities.

Thus we can see that inheritance and subtyping are related, but that subtyping does not necessarily imply the use of inheritance. Following are several examples that illustrate this point.

Example 1: Achieving similar behavior without using inheritance.

Consider two types, one for general sets and the other for ordered sets. The type for ordered sets is a subtype of the type for general sets, since ordered sets can be used, from the point of view of functionality, anywhere that general sets can be used. (Note that the converse is not true.) The two types need not achieve their similar behavior by having ordered sets be a subclass of (i.e., inherit from) general sets. In fact, one would expect that ordered sets would have an implementation very different from general sets because of incompatible efficiency concerns.

Example 2: Achieving similar behavior using inheritance.

Consider a class Queue with operations Append and Remove. A subclass of Queue, call it DEQueue, can be constructed by having it inherit the declarations of Queue and define the additional operations InsertAtFront and RemoveFromRear. This subclass realizes a double-ended queue, which is a queue that can be manipulated at both ends. It turns out that the subclass DEQueue is also a subtype of Queue, since it exhibits similar behavior: a double-ended queue can be used anywhere a standard queue can be used because it provides at least the functionality of a standard queue. In fact, the use of inheritance guarantees this relationship.

Example 3: Achieving "reverse" similar behavior using inheritance.

A variation on the previous example is to consider constructing Queue from DEQueue. This would be done by having Queue inherit from DEQueue and disregarding some of the inherited operations, specifically InsertAtFront and RemoveFromRear. (Disregarding inherited declarations is formalized in some languages and is discussed in the next section.) What we now have is subtyping that runs in the opposite direction to inheritance; Queue inherits from DEQueue, but DEQueue is a subtype of Queue. Notice that we can also construct a stack using declarations inherited from DEQueue. While the result is syntactically awkward because operation names are not the standard Push and Pop, it nonetheless is semantically (i.e., behaviorally) sensible. The syntactic problem can be corrected by a simple renaming facility.[6]

These examples demonstrate the danger in equating inheritance and subtyping. In particular, it cannot be assumed that the inheritance hierarchy of a system is the same as the subtype hierarchy;[7] the two hierarchies are independent in the first example and are counter in the third example. If a language provides an inheritance mechanism but no separate subtyping mechanism (e.g., Smalltalk and Flavors), then a subtype cannot be defined that is implemented differently from its supertypes. Some languages (e.g., CommonObjects) do indeed provide ways of specifying the inheritance hierarchy separately from the subtype hierarchy. Still other languages (e.g., Owl) provide a hybrid approach in which inherited entities can be selectively reimplemented. The hybrid approach, however, cannot adequately address situations such as that of the third example.

The third example demonstrates another point that has not yet been discussed. If a language allows a subset of the inherited declarations to be disregarded, then having a class inherit declarations from a superclass does not necessarily imply that the subclass will exhibit similar behavior. This is the case for Queue, which inherits from DEQueue but is not a subtype of DEQueue. Actually, the issue is more complicated than would appear because it is based on the meaning of "similar behavior". We therefore defer further discussion of this issue to Section 3. We do point out here, however, that researchers have recognized that there is something intuitively unsatisfying about a component that inherits (i.e., is constructed) from widely different components, and yet is said to be similar to every one of those components. This intuition reflects the fact that current views of "similar behavior"

[6]Proposals are beginning to appear that attempt to address some of these awkward syntactic problems, such as the notion of "upward inheritance" described in [Schrefl and Neuhold, 1988].

[7]Snyder [Snyder, 1986] attributes this observation to Peter Canning.

are too narrow.

To summarize, we can relate back to the three sources of confusion between inheritance and subtyping mentioned in Section 1. First there is the issue of inheritance and subtyping both being viewed as relationships among "type-like" system components. This source of confusion can be avoided by recognizing that the relationships implied by inheritance and subtyping operate at very different levels. The inheritance relationship can be characterized as "*a is constructed from b*", while the subtype relationship can be characterized as "*a behaves like b*". Moreover, as the examples given above demonstrate, these relationships can run independently within the same system.

Second is the issue of inheritance and subtyping both contributing to software reuse. This source of confusion is similar to the previous one in that it also involves a single concept operating at two very different levels. The reuse associated with inheritance is a low-level, implementation-oriented effect that details what, how, and where actual declarations are used. The reuse associated with subtyping, on the other hand, is a high-level, specification-oriented effect. When a type S is established as a subtype of a type T, then this means that any portion of a system designed to work on objects of type T should also work on objects of type S, independent of how S and T are actually implemented.

Finally, the third source of confusion mentioned in Section 1 is that every inheritance mechanism implies some sort of subtyping mechanism. While this is generally true, the implied subtyping mechanism may not be a very useful one or, at least, may not be useful in all situations. Given this, language designers can exercise one of three options:

1. rely on the inheritance mechanism to define the subtyping mechanism;
2. provide an additional, explicit subtyping mechanism; or
3. disallow the implied subtyping mechanism and in its stead provided an explicit one.

The first option is equivalent to letting semantics be driven by syntax or specification be driven by implementation. It is typical of (older) languages where an attempt is made to retrofit subtyping concepts and, partially as a result, has served to cloud the distinction between inheritance and subtyping. The second option leads to the widely acknowledged language defect of having more than one way to express the same thing, in this case in situations where the implied subtyping is appropriate. Such a defect is confusing to system developers, since they do not know which mechanism to use, and confusing to

system maintainers, since they do not know what to infer from seeing the use of one mechanism or the other. The third option is probably superior to the first two, but can prove inconvenient or redundant when implied subtyping would suffice. In Sections 4 and 5 we argue for a new approach:

4. first decide on an appropriate subtyping mechanism and then design an inheritance mechanism to support that subtyping mechanism.

In a sense we are just advocating the language-design maxim that syntax should be designed to reflect semantics and the system-design maxim that implementation should follow specification.

2.2 Impact on Abstraction

Abstraction is a means for managing complexity through selective suppression of detail [Shaw, 1984]. Probably the two foremost abstraction techniques used to manage complexity in software systems today are *information hiding* and *specification*. Information hiding is a technique for distinguishing those details of a system component that should be of interest to another component from those details that should not be of interest. Specification is a technique for describing the essential characteristics of a component independent of how that component actually realizes those characteristics.

An important question to ask is how the effectiveness of abstraction, particularly information hiding and specification, are affected by the presence in a language of inheritance and subtyping mechanisms. As it turns out, inheritance can have a negative impact on information hiding, while subtyping can enhance specification.

It has been observed (notably by Snyder [Snyder, 1986]) that the typical inheritance mechanism found in programming languages violates the basic information hiding mechanism of those languages. Indeed, in most cases, this was the intended effect. In Smalltalk, for example, the variables declared in a class are hidden from ordinary clients of the class. Subclasses of a class, on the other hand, are afforded access to the declarations of those variables so that the programmer of a subclass can develop additional functionality for manipulating the variables. The problem that this can cause is not so much technical as it is methodological. The purpose of hiding the variables is to allow a programmer the freedom to change the (low-level) details of a class without having to be concerned that the change

might affect the clients of the class. A special kind of client, however, can be affected by a change, namely subclasses of a class. Thus, the programmer of a class is in fact not free to make changes; the value of the information hiding mechanism is significantly reduced.

A concrete example should clarify this point. Recall the sequence example, where we have a class providing the notions of next and previous element and a subclass for a sequence of employee information objects. Assume that we are programming in Smalltalk and that a method in the subclass for some reason directly retrieves the value of the variable for holding the previous-element pointer, which is inherited from the superclass, instead of using the retrieval method, which is also inherited from the superclass. If we wish to change the implementation of the notion of previous element, say, to save space by eliminating the pointer and instead doing traversals of the sequence, then we have a problem. In particular, we must also change the subclass. More generally, we must examine the transitive closure of the subclasses of the class to be changed in order to determine the possible effects of that change; this can be a major undertaking in a large system. Clearly, having the otherwise hidden details available undermines the intended abstraction.

In Section 4 we argue for inheritance mechanisms that avoid this problem. But it is still interesting to review how some languages have addressed the problem. Smalltalk and CLOS reveal all to inheritors. C++ and Owl allow explicit control of what is visible to all (*public*), what is hidden from all (*private*), and what is revealed to inheritors (*protected* in C++, *subtype visible* in Owl). CommonObjects provides similar controls. C++ additionally allows definitions to be made visible to specific other components, which must be named explicitly. The PIC family of languages [Wolf, 1985; Wolf *et al.*, 1988] support the ultimate in precise control over visibility, such that visibility of individual declarations can be determined from the perspective of a provider, an inheritor, or an ordinary client of a declaration.

The effect that subtyping has on abstraction is also significant, but in this case the effect is a positive one. The main point is that subtyping enhances specification by allowing one to manipulate and relate specifications in more powerful ways. Further, one reasons about these relationships in a fundamentally semantic and abstract manner. Therefore, the positive effect is that subtyping encourages one to think in terms of abstractions rather than implementations, since subtyping permits, and actually requires, programming in terms of abstractions. We further conjecture that building subtype structures encourages

the design of "better" (more reusable, at least) abstractions.

3 Varieties of Inheritance and Subtyping

Taking the view that inheritance is language supported reuse of an essentially syntactic nature, we conclude that there might be many different kinds of inheritance in the sense that there are different language structures one might wish to reuse. There are also several different proposed notions of subtype that fall within the scope of our definition, as well as different approaches to rules for determining type conformance. In this section we explore the varieties of inheritance and subtyping, including conformance rules. Our aim is not so much to provide an exhaustive survey as to illustrate the range of possibilities.

3.1 Varieties of Inheritance

Working from the thesis that inheritance is syntactic, let us consider the various syntactic components of a class definition in a programming language.[8]The primary components are the *interface*, the *code*, and the *representation* of the class. The interface is the specification of the class. It is typically some kind of list of operation signatures, where a signature gives the name of the operation and possibly types for the arguments and results. The code is the (statement) bodies of the operations. Operations are known in some languages as procedures, functions, or methods. The representation is simply the data structure part of the class definition. Just as the interface and code of a class break down into a set of operations, the representation usually breaks down into a set of *slots*, which have also been called variables[9]and fields. The code together with the representation form what is usually referred to as the implementation of the class.

3.1.1 Code and Representation Inheritance

The kind of inheritance most frequently discussed and implemented is inheritance of code for operations of a class. Since code tends to rely and depend upon the representation,

[8]We are not considering *delegation* (see [Lieberman, 1986], for example), which is inheritance at the object level, as opposed to the class level.

[9]More precisely, they are called *instance variables* in Smalltalk (and some other languages), primarily to distinguish them from other kinds of variables, e.g., global variables.

representation inheritance and code inheritance are frequently done together. This form of inheritance is provided by SIMULA and Smalltalk, for example.

As discussed in Section 2 and in [Snyder, 1986], it is both possible, and preferable for reasons of abstraction, to provide an *abstract* interface to the representation, thus defining what might be called an "abstract representation". It might also be preferable to base inheritance on this interface rather than on the concrete representation and code of the class. This would allow code for all operations except those that define the abstract representation to be inherited and used with any other concrete representation that provides the same abstract interface. To clarify, consider a class Point, which has a concrete representation consisting of x and y slots. Suppose we define operations to fetch and store abstract slots x, y, r, and theta. If we define all *other* Point operations in terms of these, then those other operations will work correctly on both the usual Cartesian coordinate (x-y) representation as well as a polar coordinate (r-θ) representation.

Taking the abstract representation approach in Smalltalk would imply that a subclass would not directly access the slots defined by its superclass. Rather, operations would be provided to access real or virtual slots, and the subclass would invoke these operations as necessary. Similar techniques could be, and have been, developed for other languages.

The point is that judicious application of abstraction allows representation inheritance to be decoupled from code inheritance, and hence that code and representation inheritance are distinct. The distinction is brought out reasonably well in Owl, where a slot is always treated as a pair of operations that implement fetching and storing the value of the slot. The code for these operations may be inherited, in which case the slot will appear in the subclass. Alternatively, the operations can be reimplemented in terms of other slots, and the representation is not inherited. In either case the same abstract interface — the fetching and storing operations — is maintained. This can be illustrated by extending an example of the previous section:

```
typemodule Car;
  . . .;

  component me.Passengers: Integer is field;

  operation ChangePassengers (me, Delta: Integer) is
  begin
    me.Passengers := me.Passengers + Delta;
  end ChangePassengers;

end typemodule Car;
```

In this example, the declaration component me.Passengers: Integer implicitly creates operation signatures that are indistinguishable from these:

```
operation GetPassengers (me) returns (Integer) is ...;
operation PutPassengers (me, Value: Integer) returns (Integer) is ...;[10]
```

The is field part of the component declaration tells the Owl compiler to implement Passengers as a slot and to generate code for GetPassengers and PutPassengers in terms of that slot. It is possible, however, to write one's own implementations for the operations. In that fashion a virtual slot, implemented by a pair of "get" and "put" procedures, is indistinguishable in Owl from a real slot created by an is field declaration. As a (slightly contrived) example, suppose that we wish to implement cars so that passengers means the number of people in addition to the driver. We can provide a TotalPeople component as follows:

```
component me.TotalPeople: Integer
  get is (me.Passengers + 1)
  put is (me.Passengers := value - 1);
```

The implied signatures for GetTotalPeople and PutTotalPeople are similar to those for GetPassengers and PutPassengers, including the name Value for the argument to PutTotalPeople.

[10] PutPassengers returns an integer because Owl has a convention that all assignment-like forms return the value assigned, which can then be used in larger expressions.

3.1.2 Interface Inheritance

In addition to code and representation, the interface of a class can be inherited. The code and signature of an operation are usually bound together syntactically, so that when code is inherited, signature is too. This is the normal case in C++, Owl, and Eiffel: a subclass inherits the operation signatures of its superclasses. In Smalltalk, subclasses also inherit signatures, to the extent that Smalltalk can be said to provide signatures.[11] In fact, Smalltalk inherits interfaces quite strictly: a subclass always has all the operations of its superclass, though the subclass may add more operations and is free to change the implementation of any operation.

Still, are there any cases in which code is inherited *without* inheriting interface? We have not been able to determine any. What about the converse question: is it possible to inherit the interface without inheriting code? This question is difficult to answer because of the way code and signature are bound together syntactically — it all hinges on exactly what is admitted as inheritance. The problem is that if a subclass supplies new code for an operation, it always supplies a signature as well.

It appears necessary to supply at least part of the signature so that one can tell if an operation declaration is overriding one that would be inherited or if it is defining a new operation. Let us call those aspects of the signature that serve to distinguish it from other signatures as the *identifying part* of the signature. The identifying part always includes the operation name, but may also include the number of arguments, their types, and possibly further information, all depending on the whether the language permits overloading and how it resolves any overloading that occurs.

Suppose that operation O is defined in both classes S and T, with S inheriting from T. By saying O is defined in both places, we are indicating that the identifying parts of the two definitions are the same. We will say the signature of O is inherited if S is constrained in some way by the definition in T. Such a constraint demonstrates the connection or dependency that our definition of inheritance requires. In languages where the identifying part is the whole signature, this says that there is no interface inheritance separate from code inheritance; Smalltalk and C++ have no separate interface inheritance under this

[11] Smalltalk signatures consist only of the names and arity of operations, but convey no information as to the expected types of arguments or results.

definition. It is tempting to say that the virtual functions of C++ define pure signatures that are inherited, but virtual functions really just permit dynamic binding of operation calls to operation code. In Owl, though, there is separate interface inheritance because the identifying part is just the operation name and the type of the (implicit) first argument. Thus, inheritors are constrained regarding the types of the other arguments. In Eiffel, inheritors are constrained in a more interesting way. In particular, Eiffel supports the specification of preconditions and postconditions whose correctness must be maintained by inheritors.

In sum, interface inheritance can be distinguished from code (and representation) inheritance. On the other hand, because of the way most programming languages bind together the signature and code of an operation, interface and code are usually inherited together. It is only when the signature goes beyond what is used to identify operations that separate interface inheritance can be seen at work. It operates by constraining the legal signatures, or preconditions and postconditions in the case of Eiffel, for the same operation in a subclass.

3.1.3 Abstract Classes and Exemplars

At this point it is useful to consider how *abstract classes* relate to inheritance of interface, code, and representation. An abstract class is one that has no instances — its only role is to be a source for inheritance. An example abstract class is SequenceElement, described in Section 2. The class is abstract in the sense that it is not useful to have a sequence by itself; it must be a sequence *of* something, such as the data defined by the subclass EmployeeInformation. Note that the concept of an abstract class differs from that of a *parameterized class*. A parameterized class is effectively a shorthand definition for a whole family of classes, while an abstract class is a repository of pieces of interface, code, or representation to be inherited and incorporated into other classes.

Sometimes an abstract class defines representation that can be inherited. For the representation to be most useful, however, the abstract class generally needs to supply some operations as well. This is the case for SequenceElement. In Smalltalk, a class for doubly-linked lists is used as an abstract class for the data structure representing the saved state of Smalltalk pseudo-processes, which are typically linked into lists such as the list

of processes ready to execute or the list of processes waiting on a particular semaphore. The class defines forward and backward links, as well as operations to insert and remove elements from a doubly-linked list. It would appear that the purpose of embedding the links in the process-state vectors is for time and space efficiency, since processes do not act much like linked lists. The situation also indicates a possible confusion between part-of and inheritance: we may grant that for efficiency it is useful to have list information as part of a process-state vector, but that does not imply that the process-state-vector class should inherit from the doubly-linked-list class.

It is perhaps more interesting to note that abstract classes can define useful behaviors, in terms of interfaces, in a fairly abstract way. For example, Smalltalk systems have an abstract class called SequenceableCollection, which supplies a large number of operations for examining, modifying, and searching a collection of objects that can be named by the integers $1, \ldots, n$ where n is the size of the collection. Subclasses of SequenceableCollection include OrderedCollection, which is a doubly-ended queue with the sequential order determined by queue position, LinkedList, which is a singly-linked list with the order being that defined by the links, and Interval, which provides a finite arithmetic progression in terms of a starting number, limit, and increment. These subclasses have quite different representations, but all support the SequenceableCollection operations.

Some of the SequenceableCollection operations, such as accessing the nth element of a sequence, are implemented only in the subclasses, since they need access to the representation. Other operations can be defined in terms of these, and can in fact be implemented only in the abstract class. This is effectively an application of representation abstraction without a specific representation. It is clear that what is being defined by the abstract class is *behavior*, and that the abstract class is also relying on the subclasses to provide additional behavior.

In Owl and Eiffel one can actually supply an operation signature without a code body. This is useful in building abstract classes since it allows one to describe the operations that the subclass must implement — that is, what the abstract class is assuming of its concrete subclasses. The same kind of thing is done in Smalltalk by giving an implementation in the superclass that always "blows up" when called, thus forcing the subclass implementor to supply a working definition. The name for this is suggestive: **subclassResponsibility**. The virtual functions of C++ can be used to build such abstract classes. As in Smalltalk,

if the operation is to be supplied by the subclasses only, the superclass must supply an implementation that "blows up" when used.

Exemplars [LaLonde *et al.*, 1986; LaLonde, 1987] carry the abstract class idea even further. Not only does one define abstract classes, but each implementation may itself consist of multiple representations, each of which implements the abstract class operations. An example is provided by singly-linked lists, in which the EmptyList and ListCell exemplars cooperate to implement SinglyLinkedList, which is itself an implementation of the abstract class List. In essence, exemplars use the dynamic binding mechanism of the language to implement an automatic case statement on an implicit variant record. In a sense, the true and false objects of Smalltalk do the same kind of thing, except that they are instances of the classes True and False, which are subclasses of Boolean. Exemplars avoid the extraneous subclass structure.

With abstract classes we see code inherited without representation (through application of the abstract-representation approach), and interfaces inherited without code or representation. Thus, abstract classes further illustrate that inheritance of interface, code, and representation are indeed distinct.

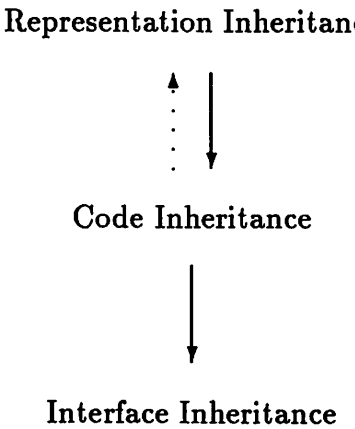
3.1.4 Relationships Among Inherited Items

We have seen that inheritance of interface, of code, and of representation are at times coincident and at other times independent. Let us summarize the exact relationships. Allow x and y to stand for any of *interface*, *code*, or *representation*. We will say that inheriting x *implies* inheriting y if the usefulness of inheriting x is substantially impaired when y is not inherited also. The strength of the implication can vary, depending on the importance of situations in which the usefulness of inheriting x depends on whether y is inherited.

- Representation inheritance strongly implies code inheritance, because a representation is virtually meaningless without operations on it.
- Code inheritance weakly implies representation inheritance, because code relies upon and assumes a representation. The implication is weak because the abstract-representation approach can significantly reduce code that depends on representation details. Abstract classes further illustrate the weakness of the implication.

- Code inheritance strongly implies interface inheritance. All code has an interface, and inherited code's interface cannot be very different in its original and new locations.
- Representation inheritance strongly implies interface inheritance. This is true because representation inheritance strongly implies code inheritance, which in turn strongly implies interface inheritance.
- Interface inheritance implies neither representation nor code inheritance — a class can inherit all the interface of a superclass but implement the behavior in a completely different way (both representation and code). Abstract classes illustrate this situation: they can be used to define a behavior that is then implemented in many different ways.

These relationships are depicted below:



3.1.5 Issues of Multiple Inheritance

To our knowledge, interface, code, and representation are the only syntactic components of class definitions for which inheritance has been provided in programming languages. Beyond how those kinds of components can be inherited, the greatest variety in inheritance comes from whether or not the components can be inherited from multiple sources and, if so, what happens if there is a conflict. Possible conflicts include two superclasses providing an operation having the same signature, two superclasses providing a slot having the same name and type, and the same superclass serving as a multiple source (e.g., if A is a superclass of both B and C, and B and C are superclasses of D, then from the perspective of D, A serves as a source twice).

Clearly, the root of such conflicts is the possibility that two components inherited from multiple sources may be indistinguishable according to the visibility/overloading rules of

the language. In fact, the main issue in devising a multiple-inheritance scheme is in dealing with such situations, which we call *inheritance ambiguity*, or *ambiguity* for short. Following is a range of methods that can be employed to address this problem:

- *Disallow ambiguity* — disallow inheritance of indistinguishable components.
- *Resolve ambiguity at definition time* — allow indistinguishable components to be *available* from multiple sources, but somehow actually inherit only one of them.
- *Resolve ambiguity on use* — allow inheritance of indistinguishable components, but somehow disambiguate upon each *use*.
- *Combine* — allow inheritance of indistinguishable components, but somehow *combine* them into a single, new component.

These methods can be further refined to apply differently when the ambiguity involves multiply-inherited components originating from different sources and multiply-inherited components originating from the same source.

SIMULA and Smalltalk disallow multiple inheritance altogether. Some extended Smalltalk systems support a weak kind of multiple inheritance that is really just a slightly automated form of copying and may not track changes made to each superclass [Borning and Ingalls, 1982]. Owl requires the user to resolve ambiguity at definition time (by designating the desired source for a given operation); a component is not considered ambiguous if its ultimate source is unique and it is simply being inherited via multiple paths.[12] CLOS also disambiguates at definition time by applying a standard rule ("depth first, left to right, up to join"). This works by defining a standard ordering of all direct and indirect superclasses of a class, and inheriting from the source that occurs first according to this ordering. Stroustrup [Stroustrup, 1987] has proposed a multiple inheritance scheme for C++ in which ambiguity is resolved on use by requiring the user to state which inherited component is being referred to in each case. The proposal also includes a mechanism, called *virtual classes*, for situations involving multiply-inherited components originating from the same source; the user can indicate that the inheritance should give rise to only one set of such components.

[12] Owl's rules are probably less interesting than they sound, since all the signatures for an ambiguous operation must be the same — only the code can vary.

The Flavors system supports combination of operation code. The basic idea is that the final code body consists of a concatenation of the inherited code bodies. Classes called *mixins* are built specifically to be combined with other classes, acting as abstract classes. Flavors provides considerable flexibility in the order in which the code bodies are concatenated. For example, one can define a "wrapper" mixin to do special things both before and after the basic code body with which the wrapper is being mixed. Interesting wrappers include locking protocols (lock before, unlock after) and graphic-display artists. MELD [Kaiser and Garlan, 1987a; Kaiser and Garlan, 1987b] illustrates another way of combining code, based on data flow.

In the discussion above we have mainly concentrated on inheritance of code. Multiple inheritance of representation affords the same options for dealing with ambiguity, except that combination does not seem to make any sense unless the sources provide slots having the same type, or the language does not have type declarations for slots. We are not aware of languages that combine individual operation signatures in any way; this is probably because there are not any sensible and useful ways to combine signatures.

3.2 Varieties of Subtyping

We now consider some of the ways two types may be related so that one is called a *subtype* of the other. Recall that our definition of subtyping is in terms of behavior: S is a subtype of T if the behavior of type S is related to the behavior of type T such that, under any applicable constraints, objects of type S can be substituted for objects of type T without ill effect.

It is perhaps interesting to note that this substitutability condition immediately implies that the subtype relation is transitive: if S is a subtype of T and T a subtype of U, then S will be a subtype of U under any reasonable interpretation of "without ill effect". The relation is also (trivially) reflexive: since S may be substituted for S, S is a subtype of S. It should be clear also that subtypeness is not symmetric, and is generally anti-symmetric. That is, if S is a subtype of T, then T is probably not a subtype of S. In fact, if the subtype relationship holds both ways, S and T are clearly equivalent.

Another useful observation is that, while it is quite possible for a type to have multiple supertypes, this does not imply any particular relationship between the supertypes

themselves. For example, DEQueue as defined in Section 2 is a subtype of both Stack and Queue, but neither is Stack a subtype of Queue nor vice versa.

Although we have been able to describe some properties that the subtype relationship must have, there is still much room for variation, which we now consider. These variations can be broadly grouped into those that resemble *subset* relationships and those that are based on *behavior.*

3.2.1 Subset Subtyping

The varieties of subtyping that resemble subsets are all quite similar in that the instances of a subtype are always also instances of the supertype. They also share the property that, to the extent that the operations of the supertype remain defined on the subtype, the behavior of the operations is the same on both types.

Subranges and Embedded Subtypes The subranges of Pascal and Ada have been called subtypes. An example is 1..100 in Pascal, which defines an integer subrange type. Such types possess the same operations as their parent types, and can be considered as being defined via inheritance. Clearly, 1..100 is a subtype of Integer according to our definition. The "applicable constraints" in this case are that we stay within the range of 1 through 100.

A more subtle form of subtype relationship is that of Integer as a subtype of Real in Pascal. Following [Bruce and Wegner, 1987], Integer would be called an *isomorphic subset subtype* of Real; we use the term *embedded subtype* for the same relationship. Integer is a subtype of Real because invocations of Integer operations that have defined Integer results act the same as the corresponding operations on Real when presented with corresponding arguments; for example, 1 + 2 = 3 and 1.0 + 2.0 = 3.0. Addition, subtraction, and multiplication indeed act the same. Division presents a bit of a problem, but if we define Integer division so that the result is defined only when it is an exact integer (e.g., 4 / 2 = 2) and undefined otherwise (e.g., 3 / 2), then Integer operation results still correspond to Real results whenever both are defined. Thus, under this definition of division, Integer is a subtype of Real.

The primary difficulty with subranges and embedded subtypes is addressing the issues of closure and boundedness. For example, some languages (most LISP dialects, Smalltalk,

Owl) provide unbounded integers, also called *bignums*. These are closed over addition, subtraction, multiplication, and other operations. Division requires some decision to be made — whether or not to retain closure, thus producing rational or real results, or to offer multiple division operations with different behaviors. Bounded integers, such as 1..100, are not closed even under addition, and thus have subtly different behavior.

The real issue is the complexity of the "applicable constraints" that must be satisfied to obtain the same behavior of the subtype and supertype. Subranges and embedded subtypes frequently have constraints that depend in complicated ways on multiple arguments. Considering Integer and Real with the exact division rule suggested above, we find an example of this: Integer is a subtype of Real provided one does not attempt inexact integer division, a constraint that depends strongly on the values of both operands to the division operation. Similarly, addition on 1..100 presents a complex constraint: the sum of the arguments must not exceed 100. Such constraints cannot be expressed in terms of the arguments individually, and thus generally cannot be checked statically.

Classification Classification is another example of subtyping, deriving from taxonomy, and frequently discussed in set-theoretic terms. The traditional example of classification is taxonomy of things or animals, such as the set E, representing the set of all elephants, being a subset of M, the set of all mammals. If we devise types Elephant and Mammal to represent elephants and mammals in a computer program, we have an intuitive expectation that Elephant is a subtype of Mammal, since elephants are indeed mammals, and all operations on mammals, so to speak, should apply to them.

To the extent that classification corresponds to set theory, and to the extent that it is based on immutable properties of the objects classified (clarified in the discussion on restriction, below), classification produces quite reasonable subtypes. Unfortunately, natural (i.e., cognitive) classifications and categories do not always, or even generally, follow these rules [Lakoff, 1987]. As with subranges and embedded subtypes, classification subtypes can also give rise to additional complex constraints on when the objects can be substituted. A good example is the relationship between the type Platypus and the type Mammal, since platypuses do not possess all the attributes generally accorded to mammals. On the other hand, if a classification really is set theoretic, as with elephants, which possess all properties expected of mammals, there are no constraints on the substitutability, giving

rise to a particularly clean subtype relationship. We call such classification *pure*.

Restriction If we take a set and consider its members that satisfy some additional condition, which we call a *restriction*, we end up with a subset. Formally this is not particularly different from classification. The difference appears when we consider the nature of the subtypes formed by classification and restriction. In practice, classification usually refers to subtypes based on inherent, unchanging (immutable) properties of the items being classified, whereas restriction can be based on properties that might change. An example is Minor: those objects of type Person that represent legal minors.

Restriction is a more general form of subtyping than pure classification. For example, if Person has an IncrementAge operation, and we apply that operation to a Minor enough times, then we will eventually get a result that is not a Minor. This is similar to the closure problem exhibited by addition on 1..100, except that it now occurs with a unary operator rather than a binary operator. In any case, we end up with constraints on subtyping that depend on the current state (value) of the Minor object, which cannot usually be checked statically.

3.2.2 Behavioral Subtyping

What about subtypes that are not based on subsets? Suppose that objects of type S behave exactly like those of type T, as long as we attempt only operations defined on T. In that case we call S a *strict behavioral subtype* of T. One of the key properties of a strict behavioral subtype is that its objects can be substituted for objects of any of its supertypes without ill effect. Of course, a subtype can offer additional behavior, as long as the additions do not conflict with the behavior of the supertype. One of our earlier examples illustrates this: a DEQueue acts just like a Queue, as far as Queue operations are concerned, but a DEQueue has additional useful semantics. The essential difference between subset subtypes and behavioral subtypes is that subset subtypes are based more on state, whereas behavioral subtypes are based on behavior and need not have similar (concrete) states. Subset subtyping does take behavior into account, but does not emphasize it to the extent that behavioral subtyping does.

Type Extension One way of building a strict behavioral subtype is to take an existing type and add representation and/or operations. Thus we could take Queue and build DEQueue from it, as discussed in Section 2. We call this process *type extension.* Type extension is not as general as behavioral subtyping since a behavioral subtype can be built entirely independently of its supertype(s). That is, type extension builds a strict behavioral subtype by using a particular form of inheritance, whereas behavioral subtypes need not be built with inheritance at all.

Behavioral Similarity Consider the following modified form of the definition of strict behavioral subtyping: if objects of type S behave exactly like those of type T, under conditions C, then S is *C-similar to T.* This directly parallels our definition of subtyping, and it is easy to see that behavioral similarity subsumes all other forms of subtyping we have discussed.

A strict behavioral subtype's conditions are the least restrictive of those that include all of the supertype's behavior. Since pure classification follows the same rules, we see that it is a form of strict behavioral subtyping. That is, in both cases, S is similar to T under *any* conditions. If S is substitutable for T only under more restrictive conditions, we get a looser connection between the types, giving rise to what might be called a *loose* or *partial* behavioral subtype. An example of a partial behavioral subtype is the bounded integers as a subtype of the unbounded integers. The conditions are that the calculations always stay within the bounds. We can also view the unbounded integers as a behavioral extension, and thus a subtype, of the bounded integers.

At first it may seem inconsistent to say that bounded and unbounded integers are subtypes of each other, since subtypeness is generally expected to be anti-symmetric. We are discussing behavioral similarity, though, and its definition leaves room for different types to be considered equivalent (each substitutable for the other), depending on the conditions C and the items being considered. In fact, by adjusting the conditions C, we can trivially make any type similar to any other type. In that sense, behavioral similarity appears to be fully general. On the other hand, even though behavioral similarity may be fully general, it may not always be the simplest way to explain the relationship between two types. For example, comparing 1..100 and the Pascal Integer type, the precise similarity conditions are not easy to formulate; in this case the notion of a subrange subtype seems

more convenient.

3.2.3 Subtype Checking

An aspect of subtypes that we feel is important to review is how they affect type checking in programming languages. Some object-oriented languages provide subtyping mechanisms[13] without recognizing or using the subtype relationship in type checking. Others do integrate subtyping with type checking.

Type checking is the process of guaranteeing that an operator or operation is not applied to inappropriate arguments. There are three basic approaches: do no checking, check as operators are about to be applied (dynamic checking), and check in advance of execution (static checking). Not checking is the most efficient, but clearly the most dangerous approach. Dynamic checking is the most flexible, but least efficient approach. Static checking sacrifices flexibility for improved efficiency, and many argue that it is most effective in reducing programming errors resulting from type mismatches. On the other hand, static checking requires the compiler (or some other pre-execution tool) to check declarations and the programmer to provide those declarations. It is important to point out, however, that the amount of type information the programmer needs to provide can be substantially reduced by *inferring* types, as in done in languages such as ML [Gordon *et al.*, 1979]. It is also important to note that static checking does not preclude dynamic binding of operation invocations to actual code, Owl being an example to which we will return.

One aspect of type checking is resolution of overloaded operation invocations. In Ada, and with C++ non-virtual functions, the compiler-determined types of the arguments are used to resolve overloading.[14] In Smalltalk and Owl, and for C++ virtual functions, the type of the first argument, which is by convention implicitly the object itself, is examined at run time and used to find the right code to execute. In contrast, CLOS makes use of the types of *all* the arguments rather than just the first one.

Subtyping relates to overload resolution in that subtyping results in the definition of closely related types with overloaded operations. One consequence is that we end up with

[13] We must also consider languages that provide inheritance mechanisms that can be used to build subtype relationships.

[14] Ada actually uses result types in overload resolution as well.

circumstances in which the same operation code can be applied to objects of many types. This can occur, for example, when subtypes inherit an operation without reimplementing it. In Smalltalk, the representations are designed so that the same exact compiled code will work when inherited; in Owl the source code is recompiled in the context of the subtype, and a code body shared if the resulting code is identical. The point is that type checking is more subtle since the same operation can apply to multiple types.

When operations are reimplemented in subtypes, we end up with a significant amount of overloading to be resolved. Further, it can be argued that there are considerable benefits to deferring that particular kind of overload resolution to run time. A typical example is any kind of container type, such as Set. It would be nice to be able to print the contents of a set simply by requesting each item in the set to print itself. Doing this by requiring sets to be homogeneous (i.e., contain objects of just one type) is overly restrictive. It is much nicer to allow heterogeneity, as long as all the items are printable (i.e., supply an appropriate operation Print).

Doing static type checking in the presence of subtyping while retaining a fair amount of flexibility (i.e., dynamic binding) presents interesting conceptual, language-design, and compiler-implementation problems. In the presence of subtyping, type checking is frequently called *conformance checking*, since the issue is whether or not one type conforms to another in some way more general than strict equality. As mentioned in the introduction, conformance involves the matching of interfaces, of operation invocations to operation definitions, and of subtype operation definitions to supertype operation definitions.

Owl is an example of a language that supports subtyping, static type checking, and dynamic binding. It is instructive to examine Owl's type checking rules to gain better insight into the differences between conformance checking and traditional type checking. Owl's rules are essentially the same as those adopted for Emerald. There are two basic rules. (We omit some irrelevant details for simplicity.)

First we consider operation invocations. In the operation invocation

```
x := p (e, e1, ..., ek);
```

the static signature of p is determined by computing the static (compile time) type of the first expression, e. This can be done by inspection if e is a literal such as 37. Similarly, if e is a variable, its compile-time type is determined from the variable's declaration — all

variables must have declared types in Owl. Finally, if e involves an operation invocation, the static signature of the invocation is determined recursively, and the result type of that signature is the static type of the expression.

Once the static type of e has been determined, the definition information for that type is examined to get the signature of p. Suppose that signature is:

p (T, T1, ..., Tk) returns R

Assuming that the type of x is X, and that $S \sqsubseteq T$ means S is a subtype of or the same type as T, the conformance rules for the operation invocation are:

- R $\sqsubseteq$ X;
- ei $\sqsubseteq$ Ti, for $1 \leq i \leq k$.

Thus, each argument expression ei must conform to the signature's expected type, and the result type must be acceptable for storage into x.

The other basic rule in Owl controls what $\sqsubseteq$ means. For S to be a subtype of T, S must implement every operation that T does (S is allowed to have more operations), and the operations must conform. In particular, assume we have the following signatures for operation p:

- in T: p (T, T1, ..., Tk) returns T0;
- in S: p (S, S1, ..., Sk) returns S0.

The conformance rules are:

- S0 $\sqsubseteq$ T0;
- Ti $\sqsubseteq$ Si.

In words, the subtype result must be no more general than the supertype result. This guarantees that type checking based on the supertype signature result type will not be invalidated. The subtype arguments must be at least as general as the supertype arguments. Again, this guarantees that operation invocation checks done with the supertype signature are also correct for the subtype signature.

Note that the rule for the result is reversed from the rule for the arguments. This reversal, called *contravariance*, also appears in the conformance rules for function types in [Cardelli and Wegner, 1985]. As an alternative one might propose these *covariant* rules:

- $S0 \sqsubseteq T0$;
- $Si \sqsubseteq Ti$.

While covariance is natural-looking, it simply does not work for general static type checking. This is because the subtype operation cannot accept everything that a call statically checked with the supertype's signature might send to the subtype operation at run time. Note, however, that if $Si = Ti$ for every i (which trivially holds when there is only one argument), $S \sqsubseteq T$ under both the covariant and contravariant rules. Perhaps because of this fact, and in spite of the apparent shortcomings of the covariant rules, covariance has been adopted for Eiffel.

The Owl and Emerald conformance rules are the only known foundation for static type checking in the presence of subtypes. A number of subtleties arise, however, when considering parameterized types, and there are competing proposals in that area. Detailed consideration of the issues of parameterized types are beyond the scope of this paper. See [Cardelli and Wegner, 1985], [Meyer, 1986], and [Horn, 1987] for more discussion of subtyping and conformance with parameterized types. In the next section, we draw conclusions about desirable ways of incorporating subtyping into programming languages, and from those considerations suggest a design approach.

4 Language Design Principles

As can be seen from the review given in the preceding section, there is considerable variety in the inheritance mechanisms that have been proposed and used with programming languages. Also proposed and used are a number of notions of subtyping. Finally, static type checking in the presence of subtypes is becoming better understood. We consider this also to have been the historical development of the topic: first there were inheritance mechanisms, then the idea of subtypes began to evolve, and now we understand enough to develop some theory and to support static type checking. While this account may be

an oversimplification, it is suggestive of a developing line of thought in the programming language community, moving from mechanism towards theory, from form to meaning.

We feel it useful to organize and summarize what has been learned by suggesting some pragmatic principles of language design related to inheritance and subtyping. Not all of these principles are new, such as the Abstract Representation Principle first advocated by Snyder. Nevertheless, many of the principles have not been articulated before and we know of no language that adheres to all of the principles. We believe that the principles are likely to lead to improved language designs — improved in the sense of encouraging more reliable program construction analogous to the now well-accepted notions of data abstraction and modularity.

As we allude to above, the principles derive from the perspective that inheritance and subtyping should serve to support *behavioral abstraction* in programming languages. By behavioral abstraction we mean the treatment of types as embodying or being behaviors, rather than as syntactic constructs or data structures. Behavioral abstraction extends the notion of data abstraction by relating behaviors in terms of behavioral subtype relationships (Section 3.2.2).

In this section, we introduce the principles by describing each in turn. We then provide a more considered rationale for the behavioral abstraction approach underlying the principles. Finally, we evaluate several existing languages in terms of the principles.

4.1 The Principles

The essence of our suggested design approach is to reverse the historical flow when designing a language: consider inheritance mechanisms in the light of subtype theory. More specifically, we propose that one should design the subtype system of a language first, and then insure that the inheritance mechanisms support the subtype system well. This is analogous to saying that syntax should follow intended semantics.

Designing the subtyping system of a language first leads to the following language design principle:

> **Subtype Support Principle:** Every desired form of subtyping in a language should be supported by an easily recognized and easily used inheritance mechanism.

The Subtype Support Principle is justified on the grounds that semantic concepts in a language should be supported by clear syntax. An obvious syntactic mechanism for the semantic concept of subtyping is inheritance. The principle applies equally well to dynamically type-checked languages, such as LISP extensions or Smalltalk, as it does to statically type-checked languages, such as C++ and Owl. The only argument that can be mounted against this principle is that some particular form of subtyping may be desirable but used rarely enough that the "cost" of providing a corresponding form of inheritance is not worthwhile.

The Subtype Support Principle appears stronger than it is. In particular, the principle does not require that there be language-recognized subtyping, only that the desired subtyping schemes can be modeled easily using the features of the language. Moreover, the principle does not require that any specific subtyping scheme be provided, that inheritance be used only in support of subtyping, or that subtypes be constructed only by inheritance.

> **Exclusiveness Principle:** Inheritance mechanisms should be used exclusively to support intended subtype relationships.

Given that inheritance is a syntactic mechanism, the question must be asked as to what semantics it supports. We know of only one reasonable candidate, namely subtyping. The reason for choosing subtyping stems from the fact that inheritance establishes a strong bond among components of a system, particularly the components that are involved in defining the semantics of objects. That bond, of course, is the one that causes implicit propagation of changes from superclasses to subclasses. If such a relationship is reflected only at the level of syntax (i.e., through the inheritance structure) and so has no visible expression at the level of semantics, then we are likely to encounter serious maintenance problems. For example, how can we tell what semantic effect a change in a superclass may have, and if we cannot tell, how can we be sure that the change is the correct one? Because subtyping is the language mechanism that captures behavioral relationships among the components that define the semantics of objects, inheritance must be constrained to operate only within the context of the subtype structure.

It is clear, and a recurrent theme of this paper, that we believe that inheritance and subtyping can and should be separated. This suggests the following principle:

> **Separation Principle:** It should be possible to build subtypes and supertypes without using inheritance.

This principle hides within it the essence of a behavioral view, that subtypes have to do with behavioral rather than implementation properties. In that sense, the Separation Principle is saying that languages should support behavioral subtyping. An argument in favor of this principle is that it leads to natural ways of relating types that are semantically similar but implemented using distinctly different code and representations.

Note that the Separation Principle does not contradict the Subtype Support Principle, which says we *can*, but do not have to, use inheritance to build a subtype.

> **Multiple Implementation Principle:** It should be possible to build multiple implementations of the same type, having different code and representations.

This is actually a specific requirement from which we arrived at the Separation Principle. It is useful to state it explicitly, however, since it is a bit different. The main argument in support of the Multiple Implementation Principle is its utility: a considerable number of types afford more than one interesting implementation with differing performance characteristics but the same functional behavior. The primary argument that might be advanced against this principle is that it appears to require dynamic binding and has difficulty dealing with operations that access more than one object of the type at a time.

> **Abstract Representation Principle:** Types and classes should be designed with an abstract representation, defined in a behavioral way.

Following this principle leads to better abstraction and modularity by localizing the code that can observe and manipulate the representation. Representation abstraction is also important in the definition of abstract classes, as we discuss in Section 3.1.3. Further, representation abstraction facilitates the use of multiple implementations of a type. This is done as follows. First define an abstract class with signatures, but no bodies, for operations defining the abstract representation. Then define as many subclasses as desired, each one supplying a representation plus bodies for the abstract representation operations in terms of that representation.[15] One property of this technique is that it requires all operations to be broken down into operations on a single instance of the class at a time — but that is exactly the purpose of the abstract representation: to hide the actual implementation, even

[15] Exemplars provide a way to supply a representation as a number of similar interacting variants, as described in Section 3.1.3. Hence, a representation need not consist of a single object or a single physical layout.

from related subclasses. In sum, the arguments for the representation abstraction principle are that it increases abstraction and modularity, and assists in supporting simultaneous multiple implementations of a type.

An argument against this principle is that it forces more operation calls, making implementations of some operations needlessly costly. While more detailed analysis needs to be performed, we believe that the performance argument does not stand up. For example, suppose some operation does not directly access the representation, but uses the abstract representation operations (as it should). By recompiling the operation in the context of a given implementation it should be possible to expand simple representation accessing operations inline. This is not different from the usual situation with abstraction, where judicious inline expansion, and other optimizations, can be applied to overcome many performance objections.

> **Behavioral Subtyping Principle:** Programming languages should support behavioral subtyping.

An argument in support of behavioral subtyping is that it leads to simpler semantic relationships between types. Further, it allows one to recognize similarities of importance to users of types, leading to programs that are more general and extensible. Focusing on behavior rather than implementation may also tend to reduce the number of errors made in programming. It may also tend to enhance reuse.

An argument against behavioral subtyping is the same one commonly advanced against typing in general: objection to writing the declarations that define the behavior of program components. A sharper criticism is that we do not really understand good ways of describing behavior nor the compromises between precise description, conciseness, and automatic processing of descriptions. Another general kind of argument against a behavioral approach is that it appears to require dynamic binding, which some feel is too expensive.

> **Behavioral Typing Principle:** Types and type checking should be based primarily on abstract behavior, and secondarily on implementation concerns.

This embodies a strong view: that a behavioral approach is the best of the known approaches to subtyping. We have already stated some arguments in favor of a behavioral approach. One is that it leads to the simplest semantic relationships between types. Further, since a behavioral approach is semantics based, it provides the best hope for correct

programs and for reuse. Arguments in favor of its exclusive use include the fact that having a single mechanism reduces confusion and the likelihood of error, and that the presence of other subtyping approaches might undermine a behavioral approach and reduce its practical value.

4.2 Rationale for Behavioral Abstraction

Having put forth the principles, we can now summarize why we believe behavioral abstraction is both valuable and viable:

1. Behavioral abstraction subsumes and extends data abstraction, by establishing meaningful relationships between types.
2. Behavioral abstraction organizes a system by semantics and behavior. This requires behavior to be expressed and documented, and leads to reasoning in terms of behavioral rather than syntactic or physical structure. This in turn improves reuse and system malleability.
3. Behavioral abstraction supports multiple implementations of a type naturally, increasing the flexibility of systems. This also encourages more of a "building block" approach to software construction.
4. Behavioral abstraction has simple semantics, making it easy and reliable to use in practice and increasing the applicability of formal tools.

The prospects of formal tools deserves more detailed comment. The language and theory described in [Sheard and Stemple, 1988a], [Sheard and Stemple, 1988b], and [Stemple *et al.*, 1988] is an example application of automated reasoning about program correctness in realistic situations. Stemple and Sheard start with primitive data types and data type constructors that have a formal theory, and from them derive the theory of any type built in terms of those primitives and constructors. This applies to abstract data types as well, since their language, ADABTPL, also has a relatively simple formal semantics. They are able to prove representation invariants in realistic situations, specifically, that database integrity constraints are preserved by transactions against the database. Their language has a behavioral subtyping structure and they speak of the inheritance of *theory* by subtypes from supertypes. Such an automated reasoning capability can be used to support sophisticated optimization in addition to demonstrating correctness. The formal approach

of Stemple and Sheard may also indicate useful ways to extend the definition of interfaces beyond simple operation signatures.

We feel that the benefits of behavioral abstraction are great, especially for large systems. The disadvantages of the approach appear to be minimal, although there are the usual questions of efficiency and achieving adequate optimization from compilers. Even if efficiency is an overriding concern, to the extent, say, that dynamic binding is judged too costly for a given language or application, the value of behavioral abstraction is not undermined. Rather, one is just finding a different tradeoff between efficiency and behavioral abstraction. Still, this issue does raise questions as to the application domains to which behavioral abstraction is most important or is of greatest benefit.

In supporting behavioral abstraction, we have been motivated primarily by problems of programming-in-the-large. Indeed, behavioral abstraction does address programming-in-the-large concerns, such as reuse, evolution, maintenance, and correctness. The main observation that we would like to make is that almost all programming *is* programming-in-the-large, in the sense that programming is done in the context of a large system. For example, a considerable amount of the enthusiasm for Smalltalk has to do with the tools and components already present in the Smalltalk environment, rather than with the language itself. On workstations, we make use of considerable library packages and applications frameworks for dealing with graphics and interactive interfaces, as well as the traditional operating system features such as files.

It is reasonable to conjecture that one of the main routes to supporting exploratory programming and rapid prototyping, where the essence of the effort is to find the "right" behaviors, is providing a rich library of types and tools on which to build. Experimental systems can be quite large [Wileden *et al.*, 1988] and would benefit from judicious application of behavioral abstraction. The nature of the tools might be a bit different — more forgiving and less rigid at the outset, but helping to record and reduce inconsistencies as design and exploration progresses. Thinking and programming behaviorally, and having a crisp semantic model available, will still help. We do admit that there is still considerable room for increasing the flexibility of type systems and of bridging the gap between the relatively restrictive realm of static type checking and the relatively insecure realm of dynamic type checking.

It is interesting to consider whether behavioral abstraction is a useful principle outside the domain of programming languages. In fact, it almost certainly underlies notions of standards and interfaces used in a variety of domains and may help point the way to new styles of specifying standards and interfaces that allow considerable implementation freedom while supporting interoperability and reuse. Networking and operating systems come to mind as particular domains in computer science that benefit from these ideas, and it seems reasonable to speculate that the ideas have application broadly throughout many engineering disciplines. Closer to home, what about artificial intelligence, and its programming and cognitive modeling languages? This seems like much shakier ground. It is not at all obvious that what is good for compilers and provability is necessarily much like accurate or useful models of intelligence. Behavioral abstraction, as we intend it to be used in programming languages, relies on formal semantics that can be automatically manipulated. There are those who argue strongly that logic and formal semantic systems, such as those on which behavioral abstraction relies, are definitely inadequate as cognitive models. At present, the applicability of behavioral abstraction to notations such as knowledge representation languages must be considered an open question.

It is easy to see that effective behavioral abstraction depends on the ability to describe, compare, and relate behaviors. Thus, it implies the need for certain programming language features. Going beyond specific features, we believe that behavioral abstraction, and the principles we have presented in this section, suggest a strategy for designing the subtyping and inheritance features of programming languages. These matters are taken up in Section 5.

4.3 A Look at Existing Languages

We now consider the extent to which a number of existing languages follow the principles we have presented. We have attempted to cover a broad range of significant and popular languages, but do not claim that to offer a comprehensive survey. The results are summarized in Table 1, with additional commentary appearing in the text.

Smalltalk

References: [Goldberg and Robson, 1983; Borning and Ingalls, 1982].

	Subtype Support	Exclusiveness	Separation	Multiple Implementation	Abstract Representation	Behavioral Subtyping	Behavioral Typing
C++	E	E	P	G	G	F	G
CLOS	G	P	G	G	P	NA	NA
CommonObjects	G	P	G	G	E	NA	NA
Eiffel	E	E	P	G	E	G	G
Emerald	P	NA	E	E	E	G	E
Flavors	F	P	G	G	P	NA	NA
Owl	E	E	P	G	E	G	G
SmallTalk	G	P	G	F	F	NA	NA

E—excellent; G—good; F—fair; P—poor; NA—not applicable

Table 1: How Several Existing Languages Adhere to the Principles.

Background notes: Smalltalk has a very weak notion of type, one that is heavily tied to implementation rather than behavior. Partially as a result, it has no language-recognized notion of subtype. Subtyping relationships can be expressed in terms of coincidental operations that may or may not be built by inheritance. Those relationships, however, cannot be validated by the language system.

- *Subtype Support Principle:* If used with discipline, Smalltalk's inheritance mechanism supports a tree subtype structure (single immediate supertype) fairly well. The multiple inheritance scheme of [Borning and Ingalls, 1982] extends the structure from a tree to a directed acyclic graph, though it is based upon automated copying.

- *Exclusiveness Principle:* Because of Smalltalk's weak notion of type, proper use of inheritance within a subtyping structure cannot be enforced.

- *Separation Principle:* It is possible to define subtypes independently of inheritance, but such subtypes exist "only in the head" of the programmer.

- *Multiple Implementation Principle:* Multiple implementations can be created by building multiple subclasses of a Smalltalk class. The subclasses, however, would all share the common representation of the superclass (although they can add to it).

- *Abstract Representation Principle:* While no special features are provided, and typical Smalltalk style does not follow this principle, abstract representations can indeed be devised and used. Abstract classes appear frequently as sources for inheriting and structuring code. While Smalltalk does prevent the representation of objects from being manipulated at will,[16] it does not protect them from subclasses.

[16] We are assuming the user does not employ some of the powerful operations that allow any object to be accessed and updated (instVarAt:, for example).

- *Behavioral Subtyping Principle:* Because Smalltalk does not have a genuine notion of type, this principle does not apply.
- *Behavioral Typing Principle:* Because Smalltalk does not have a genuine notion of type, this principle does not apply.

C++

References: [Stroustrup, 1986; Stroustrup, 1987].

Background notes: C++ is superficially similar to Smalltalk (with multiple inheritance). It differs primarily, and significantly, in that it supports a strong notion of type, static type checking, and overloading of operations. It also differs in that it permits finer control over what is inherited by and what is visible to subclasses. C++ supports a weak form of subtyping that requires exact matching of operation signatures, although more flexible subtyping relationships can be simulated using overloading. Moreover, the subtyping structure is dictated by the inheritance structure, rather than the other way around, and therefore C++ is considered to adhere very well to the Exclusiveness Principle, but not so well to the Separation Principle.

- *Subtype Support Principle:* C++ has one form of subtyping and it is supported explicitly by inheritance.
- *Exclusiveness Principle:* In C++, inheritance cannot be used other than to support its single notion of subtyping.
- *Separation Principle:* C++ does not adhere to this principle, since a subtype always inherits from its supertypes, except that alternate code and representation can be supplied.
- *Multiple Implementation Principle:* Multiple implementations can be created by defining an abstract class and then a subclass for each different implementation.
- *Abstract Representation Principle:* An abstract representation can be created by hiding the representation from subclasses, using the private construct, while providing those subclasses with appropriate abstract representation operations. This technique, however, is used at the discretion of the developer.
- *Behavioral Subtyping Principle:* C++ adheres to the extent that signatures represent behavior, but the conformance rules are quite restrictive.

- *Behavioral Typing Principle:* C++ type checking is based on the subtype structure and therefore C++ adheres well to this principle. Its adherence is not quite as good as Emerald's, however, because the subtype structure is dictated by the inheritance structure.

Owl

References: [Schaffert *et al.*, 1986].

Background notes: Owl and C++ are remarkably similar in their adherence to the principles. The major difference lies in the fact that Owl's conformance rules are much more flexible than those of C++.

- *Subtype Support Principle:* Owl has one form of subtyping and it is supported explicitly by inheritance.
- *Exclusiveness Principle:* In Owl, inheritance cannot be used other than to support its single notion of subtyping.
- *Separation Principle:* Owl does not adhere to this principle, since a subtype always inherits from its supertypes, except that alternate code and representation can be supplied. In other words, the subtyping structure is dictated by the inheritance structure, as in C++.
- *Multiple Implementation Principle:* Multiple implementations can be created by defining an abstract class and then a subclass for each different implementation.
- *Abstract Representation Principle:* Owl adheres extremely well, since all representation access is mediated through operations whose bodies can be replaced. Further, through the **private** and **subtype_visible** attributes, access by subtypes to representation and code of supertypes can be controlled.
- *Behavioral Subtyping Principle:* Owl adheres well, to the extent that signatures represent behavior.
- *Behavioral Typing Principle:* Owl type checking is based on the subtype structure and therefore Owl adheres well to this principle. Its adherence is not quite as good as Emerald's, however, because the subtype structure is dictated by the inheritance structure.

Emerald

References: [Black *et al.*, 1987].

Background notes: Emerald and Owl are very similar in terms of their realization of the notion of subtyping. They differ primarily in two ways: First, in Owl the subtype/supertype structure is established through an explicit syntactic construct, while in Emerald the relationships are implicitly based on interfaces. Second, Emerald does not support inheritance.

- *Subtype Support Principle:* Emerald does not provide an inheritance mechanism.
- *Exclusiveness Principle:* Emerald does not provide an inheritance mechanism.
- *Separation Principle:* Emerald does not provide an inheritance mechanism.
- *Multiple Implementation Principle:* Multiple implementations are a key feature of Emerald. They are created by using different so called *object constructors* to construct objects that conform to the same type; object construction, and hence implementation, is separate from the typing structure.
- *Abstract Representation Principle:* Subtypes are not given access to the representation of a supertype.
- *Behavioral Subtyping Principle:* Emerald adheres well, to the extent that signatures represent behavior.
- *Behavioral Typing Principle:* Emerald type checking is based on the subtype structure and therefore Emerald adheres well to this principle.

CLOS

References: [Bobrow *et al.*, 1987].

Background notes: CLOS is similar to Smalltalk in that it has a very weak notion of type that is heavily tied to implementation rather than behavior, does not have a language-recognized notion of subtype, and allows the expression of subtype relationships in terms of coincidental operations that may or may not be built by inheritance. The inheritance mechanism of CLOS, however, is much more powerful than Smalltalk's.

- *Subtype Support Principle:* The powerful inheritance mechanism of CLOS has no difficulty supporting a tree (single inheritance) subtype structure. It also supports a directed acyclic graph structure, though care must be taken with respect to multiple superclasses that define operations having the same name.
- *Exclusiveness Principle:* The CLOS inheritance mechanism is designed to allow a composite object to be built by inheriting from its components[17] and thus does not

[17] CLOS advocates have been known to present such examples first in their talks on the language.

exclusively support subtyping. As we discuss in Section 2, aggregation (i.e., the "part of" relationship) should not be confused with subtyping.

- *Separation Principle:* While subtypes and supertypes can certainly be built without inheritance, it should be noted that CLOS does not have a language-recognized notion of subtype, only of inheritance. Therefore, subtype relationships cannot be adequately guaranteed.
- *Multiple Implementation Principle:* CLOS allows multiple implementations, but in the absence of any indication that the implementations are implementations of the same type.
- *Abstract Representation Principle:* Because slots are revealed to all other components, CLOS is not considered to adhere to this principle.
- *Behavioral Subtyping Principle:* Because CLOS does not have a genuine notion of type, this principle does not apply.
- *Behavioral Typing Principle:* Because CLOS does not have a genuine notion of type, this principle does not apply.

Flavors

References: [Moon, 1986].

Flavors adheres to the principles similarly to CLOS. One significant difference is the greater power and substantially more complex semantics of the inheritance mechanisms of Flavors, namely method combination. This makes Flavors adhere even less well to the Exclusiveness, Separation, and Abstract Representation principles.

CommonObjects

References: [Snyder, 1985].

In contrast to Flavors, CommonObjects adheres to our principles better than does CLOS. CommonObjects does better primarily on the Abstract Representation Principle, though its stronger stand on data abstraction improves its adherence generally.

Eiffel

References: [Meyer, 1986; Meyer, 1988].

Eiffel is identical to Owl in terms of its adherence to the principles.

5 Some Implications of a Behavioral Approach

Having proposed several language design principles and argued in favor of a behavioral approach, we now consider in more detail the implications a behavioral approach would have for programming language features and the language design process.

5.1 Effects on Types

One immediate implication of a behavioral approach is that the most important property of the notion of *type* in a programming language is that a type implies a behavior. It is instructive to consider the historical development and refinement of types in programming languages. In FORTRAN, types of variables constrain their behavior, of course, but FORTRAN types primarily specify implementation. Further, FORTRAN does not allow new behaviors to be described — that is, there are no user-defined types. ALGOL68, PL/1, COBOL, and Pascal have richer type systems than FORTRAN, but types are still primarily a means for specifying implementation. CLU, and to a lesser extent Modula and Ada, allow type implementations to be hidden, which has the effect of playing down implementation concerns and emphasizing behavior.

It is clear that the abstract data type approach is closely related to the behavioral notion of type. The main difference is that a pure behavioral approach would allow more complete dissociation of type specification from implementation: in the ideal it may be possible to describe a type without supplying any true implementation. We suspect that any effective and practical method for completely describing behavior will be some kind of implementation, such as an abstract model. On the other hand, less complete specifications might very well be possible without implementation. Ada, CLU, and Owl, for example, allow types to be *described* abstractly without providing an implementation.

A *behavioral type* is a description of behavior and need not be tied to any particular component or piece of code. In a behavioral language, a component would not so much *be* a type or *represent* a type as *have*, *implement*, or *conform to* a type. That is, if types describe behaviors, then, since two components might provide indistinguishable behavior, types cannot be identified with components. In this way *all* hints of implementation are removed from the notion of type. This seems to be a logical next step in the evolution of types in programming languages.

How to express behavior is a difficult problem, worthy of considerable study. On the one hand, signatures are relatively easy to check, but somewhat unreliable since they capture little semantics. Full semantic descriptions are more difficult to write and check (not to mention the problem of devising an appropriate notation for them), but provide maximum safety. Moreover, they better support code optimization, given a sufficiently powerful optimizer. Finding a good middle ground, or even showing how to go beyond signatures in a practical programming language, is still an open problem, though [Sheard and Stemple, 1988b] and [Perry, 1986] help point the way.

5.2 Effects on Type Checking and Call Binding

If behavior is the most important property of types, then behavioral conformance is the appropriate approach to type checking, as opposed to exact matching of types. Any specific type definition and conformance checking design takes some aspects of behavior to be significant and treats others as being irrelevant for definition and checking purposes. For example, performance attributes such as size and speed might be ignored when defining and checking types since those properties do not affect program correctness, even though they may ultimately be very important in determining if the program meets its full requirements. This observation suggests that no single notion of type conformance will suffice for all languages or circumstances. Allowing the significant attributes of behavior to be adjusted according to situation is an interesting language research problem.

One specific kind of conformance that is certainly useful is *full conformance*: a component or implementation *fully conforms to* (or *implements*) a type if the component meets the type's specification. The component may offer *more*, but it cannot offer less than the type requires, or change any aspects of the type's specification. It is obvious from this definition that a type may have a number of implementations. How to describe (or determine automatically or semi-automatically) which implementation to use when more than one is available is an open question in language design.

Full conformance implies neither the presence nor absence of dynamic binding or overloading. Owl and Emerald show that static full conformance checking and dynamic binding can be used together effectively. Using multiple implementations of the same type simultaneously from the same piece of source code does appear to require, or at least is

greatly simplified by, dynamic binding. Further, dynamic binding with static conformance checking, except of multiple implementations of exactly the same type, requires inexact matching of components to types. The Owl and Emerald conformance rules demonstrate one way of doing this.

We have argued that full conformance is useful in considering the *implements* relation: the relationship between type specifications and components that implement them. *Partial* conformance (behavioral similarity) is also important and appears more related to inheritance than to implementation. For example, it is not difficult to imagine type inheritance operators that allow one to take an existing type and modify it in ways other than pure addition, so as to produce a new type that conforms only partially to the old one. Further, specifying new types that way may be quite convenient. Supporting such inheritance operators does not undermine the Behavioral Typing Principle as long as the semantics of the operators can be clearly and crisply characterized. Behavioral similarity is a topic deserving further study.

In addition to full and partial conformance, type parameters to components or type definitions present interesting situations and problems. Even CLU and Ada have ways of imposing behavioral requirements, in terms of signatures, on types used as parameters. For an example of such a requirement, consider a sort component for sorting items of type T, T being a parameter of the component definition. The component may require that T offer comparison operators if it is to be an acceptable parameter. A more subtle situation is one in which the parameterized type has some property depending on whether the parameter(s) have a corresponding property. The parameterized type Array demonstrates this situation: it is possible to print an array if and only if the type of item contained in the array also has a Print operation. Owl has some support of parameterized types along these lines, but Horn's work [Horn, 1987] goes into more subtle situations, such as parameterized abstract classes, and should be examined closely before undertaking future designs. As mentioned above, conformance for parameterized types and components is not fully understood and is another topic deserving additional research.

We have pointed out that taking a behavioral approach to conformance checking is somewhat independent of the issue of static versus dynamic binding. We also should point out that behavioral typing and conformance checking are independent of this issue as well; a behavioral approach prescribes the *nature* of the check but not *when* it is done. Granted,

having type specifications, especially behavioral ones, is conducive to static checking, but in some kinds of programming, run-time type checking may be preferred.

5.3 Explicit or Implicit Subtyping?

There is one additional implication of behavioral typing for language designs that deserves attention. Assuming that full specification of behavior along the lines of [Sheard and Stemple, 1988b] or [Perry, 1986] will not be practical or accepted for some time, in the interim we are likely to rely primarily on operation signatures as the means for specifying types and checking conformance. Is it better to require that the programmer explicitly mark subtype relationships, as is done in Owl, or to recognize subtypes implicitly from their conformance, as is done in Emerald? Note that Owl and Emerald have the same conformance *rules*, but take opposite positions on this issue.

Since a signature is but a pale shadow of a full behavioral specification, a behavioral approach suggests that subtype relationships must be explicitly designated or marked by the programmer rather than inferred by the language system. For example, many aggregate types (Stack, Queue, Set, etc.) might quite reasonably have operations called Insert and Remove, yet they have rather different behaviors. A structural or implicit subtyping rule would allow these types to be substituted for one another inappropriately.

In the absence of full specification it seems more wise to require the programmer to note subtype relationships explicitly. Of course, the language system should check that the signatures of explicitly designated subtypes do in fact conform to their supertypes' signatures. Note that Cardelli and Wegner use implicit subtyping in [Cardelli and Wegner, 1985]. In private communication, Cardelli has indicated that this appears to produce a simpler theory than explicit subtyping; to our knowledge, no detailed theory of explicit subtyping has been developed. We feel that even if such a theory is a bit more complicated, the practical advantages of explicit subtyping justify it.

A useful view to take of conformance checking is that it is a simplified form of theorem proving — type checking in essence proves a type correctness theorem. Explicit subtype designation by the programmer can be viewed as an assertion that the appropriate behavior-conformance theorem could be proved. As we learn how to specify and check behavior more completely, we should be able to minimize the need for programmers'

assertions, and rely more on automated checking.

5.4 Static Checking and Restriction Subtypes

Recall that in Section 3, when discussing restriction subtypes, we considered an example type Person with a subtype Minor for persons that are not yet legal adults. At first glance it might appear that Minor is a full behavioral subtype of Person; but because IncrementAge prevents Minor from being a closed type, Minor is not a full behavioral subtype of Person, and only partially conforms (i.e., it conforms if we omit consideration of IncrementAge).

The closure problem arises whether we are dealing with a mutable type (Minor is mutable: it has state that can change over time) or an immutable one. In fact, a mutable type can be thought of as just a "variable" containing an immutable value, with update operations performing an implicit assignment to the variable.

There do not seem to be many alternatives in dealing with this problem. One approach, taken in the Owl and Emerald conformance rules, is to require that subtypes be closed. This has the effect of ruling out restriction subtyping, including subranges and embedded subtypes. This permits compile-time conformance checking with no extra run-time overhead. Another alternative would be to use run-time checks to verify closure. A final alternative is to use more complete semantics and prove closure for the specific uses at hand. This would include proofs that numerical code never leads to overflow, for example, in the case of bounded integers.

While one might argue that the conflict between evolution of behavior and static typing (closure) undermines our arguments for a behavioral approach to types, it is more a clarification of the roles and properties of static and dynamic approaches to types. For example, the variant record construct of Pascal shows a clear distinction between the aspects of a type that are allowed to vary during program execution (which case of the variant pertains at any given time) and those aspects that may not change (the other fields, the record properties, the fields present with each case tag, etc.).

We do wish to clarify one point with respect to restriction subtypes: while they may not be acceptable subtypes in terms of full conformance rules (including closure), they are certainly useful and are readily constructed via inheritance. Restriction subtypes are yet another of the cases where we would benefit from a theory of behavioral similarity.

5.5 Static Checking and Contravariance

There are subset subtypes where the closure problem does not arise, but static rules are still perceived as being overly restrictive. A good example is provided by considering the (unbounded) integers without division as a subtype of the rationals (again without division). Under the Owl and Emerald conformance rules the specified integer type is not a subtype of the rationals. Consider the addition operation on integers; by the contravariant conformance rule, the second argument must be a *supertype* of the rational type, not a subtype. This is partly the result of determining signatures for type-checking purposes from the first (implicit) argument alone. Still, even if that were not done, the addition operator on the integers cannot be substituted for the addition operator on the rationals, since the integer operator cannot accept rationals. It seems more appropriate to consider the rationals as a subtype of the integers, but even that may be difficult because of implementation concerns, unless an appropriate abstract representation can be devised, which seems unlikely.

It is interesting to note in this regard that CLOS provides a dynamic binding mechanism that can take into account the (run-time) types of all the arguments for an operation and locate the appropriate code to use. But it should also be noted that CLOS does not support static type checking, and type checking is the point of our discussion. Again, this issue needs further research.

5.6 Effects on the Process of Language Design

In addition to some specific implications that a behavioral approach has for features of programming languages, concentrating on behavior also suggests a new process for designing programming languages. Here are the design steps we propose:

1. Determine a model of behavior for the language. What is it that is important? For example, in most languages, an important aspect is the set of operations possessed by a type, the number and types of the arguments to those operations, and so on. Implementation and performance aspects are generally ignored.

2. Given a behavioral model, design the type system to embody it. Here is where a determination must be made as to what can actually be expressed about behavior — is it just operation signatures or can we be more precise?

3. The type checking rules for the type system should represent conformance of the behaviors that the types represent. That is, non-conforming types must have non-conforming behaviors, according to the language-specific notion of behavior. It would also be nice if conforming types always had conforming behaviors, but since conformance is likely to be approximate as previously argued, we cannot expect type checking to establish true behavioral conformance. What we *can* expect is that type checking will catch a lot of mistakes. The Owl and Emerald type checking rules, possibly extended with some of Horn's ideas [Horn, 1987], illustrate current technology in conformance checking.

4. Finally, design the inheritance mechanisms, ensuring that they preserve simple and useful behavioral properties. The point here is that an effective and safe inheritance mechanism is one that preserves key behavior, according to the behavioral model of the language — the behavioral model should guide the design of the inheritance mechanism, not vice versa. Thus, we find in Owl that inheritance works along the lines of preserving *all* language defined behavior (signatures on instance operations). This particular inheritance scheme results from Owl's definition of conformance as complete compatibility. Thus, if there are any substantial weaknesses in the Owl design, they lie in its models of behavior and conformance, not in its inheritance mechanism *per se*.

Notice that the first two steps essentially end up defining the notion of *type* for the language being designed. We think it important that one begin by deciding what is and what is not important to the concept of type in a given language, and only then should the type system be designed. However, in practice the process is undoubtedly iterative.

To sum up, we are suggesting that designers concentrate on the notion of *behavior*, and let inheritance mechanisms follow from that. The benefits of designing inheritance to match with behavior are:

- The inheritance mechanism, since it is designed to preserve significant behavioral properties, can be used to do just that. This will tend to reduce programming errors.

- The ability to reuse components will also be enhanced, in that programmers will know that important behavior will not be disturbed in the inheritance process.

- Since components that were built using inheritance will have well-defined relationships with their sources, they will be easier to understand. This also contributes to their correct use as sources of inheritance and reuse.

- Conformance checking is simplified. For example, Owl has dynamic binding, and in theory we should match *every* possible signature against the actual types when type

checking an operation call. However, the inheritance rules allow us to check just *one* signature, since we know that any other signature will conform to that one.

- Programmers can think behaviorally, rather than trying to interpret mechanical, possibly counter-intuitive, rules and "playing compiler". This will speed programming and reduce errors.

References

[Black *et al.*, 1987] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering 13*, 1 (Jan. 1987), 65–76.

[Bobrow *et al.*, 1985] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: Merging Common Lisp and object-oriented programming. Intelligent Systems Laboratory Series ISL-85-8, Xerox Palo Alto Research Center, Palo Alto, CA, 1985.

[Bobrow *et al.*, 1987] Daniel Bobrow, David Moon *et al.*. *Common Lisp Object System Specification.* American National Standards Institute, Washington, DC, 1987. ANSI X3J13 Document 87-002.

[Borning and Ingalls, 1982] A. H. Borning and D. H. H. Ingalls. Multiple inheritance in Smalltalk-80. In *Proceedings of the National Conference on Artifical Intelligence* (Pittsburgh, PA, Aug. 1982), American Association for Artificial Intelligence, pp. 234–237.

[Bruce and Wegner, 1987] Kim B. Bruce and Peter Wegner. An algebraic model of subtype and inheritance. In *Proceedings of the Workshop on Database Programming Languages* (Roscoff, France, Sept. 1987), Altaïr-CRAI, pp. 107–131.

[Cardelli and Wegner, 1985] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv. 17*, 4 (Dec. 1985), 471–522.

[Dahl and Nygaard, 1966] O.-J. Dahl and K. Nygaard. Simula – an Algol-based simulation language. *Commun. ACM 9*, 9 (Sept. 1966), 671–678.

[Goldberg and Robson, 1983] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Gordon *et al.*, 1979] M. J. Gordon, A. J. R. G. Milner, and C. P. Wadsworth. *Edinburgh LCF*, vol. 78 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1979.

[Horn, 1987] C. Horn. Conformance, genericity, inheritance, and enhancement. In *Proceedings of the 1987 European Conference on Object-Oriented Programming* (Paris, France, June 1987), AFCET, pp. 269–280.

[Jategaonkar and Mitchell, 1988] Lalita A. Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes (preliminary version). In *Proceedings of the 1988 ACM Conference on* LISP *and Functional Programming* (July 1988), pp. 198–211.

[Kaiser and Garlan, 1987a] Gail E. Kaiser and David Garlan. MELDing data flow abd object-oriented programming. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Orlando, FL, Oct. 1987), ACM, pp. 254–267.

[Kaiser and Garlan, 1987b] Gail E. Kaiser and David Garlan. Melding software systems from reusable building blocks. *IEEE Software* (July 1987), 17–24.

[Kristensen *et al.*, 1983] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Abstraction mechanisms in the BETA programming language. In *Proceedings of the Tenth ACM Symposium on Principles of Programming Languages* (1983), ACM.

[Kristensen *et al.*, 1987] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Classification of actions, or inheritance also for methods. In *European Conference on Object-Oriented Programming* (Paris, France, June 1987), AFCET, pp. 109–118.

[Lakoff, 1987] George Lakoff. *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind.* University of Chicago Press, 1987.

[LaLonde *et al.*, 1986] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. An exemplar based Smalltalk. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, OR, Sept. 1986), ACM, pp. 322–330.

[LaLonde, 1987] Wilf R. LaLonde. Designing families of data types using exemplars. School of Computer Science SCS-TR-108, Carleton University, Ottawa, Canada, Feb. 1987.

[Lieberman, 1986] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, OR, Sept. 1986), ACM, pp. 214–223.

[Meyer, 1986] B. Meyer. Genericity versus inheritance. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, OR, Sept. 1986), ACM, pp. 391–405.

[Meyer, 1988] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, 1988.

[Moon, 1986] David A. Moon. Object-oriented programming with Flavors. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, OR, Sept. 1986), ACM, pp. 1–8.

[Perry, 1986] Dewayne E. Perry. The Inscape program construction and evolution environment. Technical report, AT&T Bell Laboratories, August 1986.

[Schaffert *et al.*, 1986] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, OR, Sept. 1986), ACM, pp. 9–16.

[Schrefl and Neuhold, 1988] Michael Schrefl and Erich J. Neuhold. Object class definition by generalization using upward inheritance. In *Proceedings of the Fourth International Conference on Data Engineering* (Los Angeles, CA, Feb. 1988), IEEE, pp. 4–13.

[Shaw, 1984] Mary Shaw. Abstraction techniques in modern programming languages. *IEEE Software 1*, 4 (Oct. 1984), 10–26.

[Sheard and Stemple, 1988a] Tim Sheard and David Stemple. Automatic verification of database transaction safety. *ACM Trans. Database Syst.* (1988). To appear. Also available as University of Massachusetts, Department of Computer and Information Science, Technical Report 88-29.

[Sheard and Stemple, 1988b] Tim Sheard and David Stemple. The precise control of inheritance and the inheritance of theory in the ADABTPL language. In *International Conference on Computer Languages '88* (Miami Beach, FL, Oct. 1988), IEEE. Submitted.

[Snyder, 1985] Alan Snyder. Object-oriented programming for Common Lisp. Tech. Rep. ATC-85-1, Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, CA, 1985.

[Snyder, 1986] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, OR, Sept. 1986), ACM, pp. 38–45.

[Stemple *et al.*, 1988] David Stemple, Adolpho Socorro, and Tim Sheard. Formalizing objects for databases using ADABTPL. In *Proceedings of the Second International Workshop on Object-Oriented Databases* (Germany, Sept. 1988), ACM. To appear.

[Stroustrup, 1986] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[Stroustrup, 1987] Bjarne Stroustrup. Multiple inheritance for C++. In *Proceedings of the EUUG Spring Conference* (Helsinki, May 1987), pp. 189–208.

[Wileden *et al.*, 1988] Jack C. Wileden, Lori A. Clarke, and Alexander L. Wolf. Three techniques supporting the development of large prototype systems. In *Proceedings of Third International IEEE Conference on Ada Applications and Environments* (May 1988), IEEE Computer Society Press, pp. 28–37.

[Wolf *et al.*, 1988] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden. The AdaPIC toolset: Supporting interface control and analysis throughout the software development process. *IEEE Transactions on Software Engineering* (1988). (to appear).

[Wolf, 1985] Alexander L. Wolf. Language and tool support for precise interface control. COINS Technical Report 85–23, University of Massachusetts, September 1985.

A. Glossary

Behavioral abstraction	the treatment of types as embodying or being behaviors, rather than as syntactic constructs or data structures
Class	an encapsulator of program-entity declarations; when used as the source of inheritance, a class is said to be a superclass, and when used as the target of inheritance, a class is said to be a subclass
Code	one of three primary components of a class definition (the others being interface and representation), forming the executable-statement portion and often broken down into (statement) bodies for the operations defined in the class
Component	a portion of a program that is syntactically distinct from other portions, such as a class definition, a type definition, a signature, or a slot
Field	synonym for slot
Inheritance	a means by which new system components can be constructed from old system components such that changes to the definitions of the old components can have an effect, subject to certain constraints, on the definitions of the new components
Interface	one of three primary components of a class definition (the others being code and representation), forming the specification of the class and often broken down into signatures for the operations defined in the class
Method	synonym for operation
Object	instance, or run-time realization, of a class or type definition
Operation	a separately executed portion of code that represents a well-defined manipulation of an object; it usually consists of a signature and a (statement) body
Representation	one of three primary components of a class definition (the others being code and interface), forming the data structure portion and often broken down into slots
Signature	a specification for an operation consisting of a name for the operation and a list of the types of the operation's arguments and results
Slot	a portion of the representation component of a class, usually acting as a variable or as a reference to an object

(Statement) body	a portion of the code component of a class, usually consisting of the executable statements of an operation
Subclass	see class
Subtype	see subtyping
Subtyping	a means by which the behavior of one object can be established or asserted as being similar to the behavior of another object such that the first object can be used, subject to certain constraints, in place of the second object; the type of the first object is said to be a subtype of the type of the second object, while the type of the second object is said to be a supertype of the type of the first object
Superclass	see class
Supertype	see subtyping
Type	a behavioral abstraction of the objects that are its instances