Copying and Comparing: Problems and Solutions

Peter Grogono[1] and Markku Sakkinen[2]

[1] Department of Computer Science, Concordia University
Montreal, Quebec
grogono@cs.concordia.ca

[2] Software Systems Laboratory, Tampere University of Technology
Tampere, Finland
(On leave from the Department of Computer Science
and Information Systems, University of Jyväskylä
Jyväskylä, Finland)
sakkinenm@acm.org

Abstract. In object oriented programming, it is sometimes necessary to copy objects and to compare them for equality or inequality. We discuss some of the issues involved in copying and comparing objects and we address the problem of generating appropriate copying and comparing operations automatically, a service that is not provided by most object oriented languages and environments. Automatic generation appears to be not only desirable, because hand-coding these methods is mechanical and yet error-prone, but also feasible, because the form of the code is simple and largely predictable.

Some languages and some object models presented in the literature do support generic copying and comparing, typically defining separate "shallow" and "deep" versions of both operations. A close examination of these definitions reveals inadequacies. If the objects involved are simple, copying and comparing them is straightforward. However, there are at least three areas in which insufficient attention has been given to copying and comparing complex objects: (1) values are not distinguished from objects; (2) aggregation is not distinguished from association; and (3) the correct handling of linked structures other than trees is neglected.

Solving the third problem requires a mechanism built into the language, such as exists in Eiffel. Building such a mechanism without modifying the language requires a language with sufficient reflexive facilities, such as Smalltalk. Even then, the task is difficult and the result is likely to be insecure.

We show that fully automatic generation of copying and comparing operations is not feasible because compilers and other software tools have access only to the structure of the objects and not to their semantics. Nevertheless, it is possible to provide default methods that do most of the work correctly and can be fine-tuned with a small additional amount of hand-coding.

We include an example that illustrates the application of our proposals to C++. It is based on additional declarations handled by a preprocessor.

Keywords: Copying, cloning, equality, complex object structures.

1 Introduction

In object oriented programming, it is sometimes necessary to copy the contents of one object to another object, make a new copy (or clone) of an object, or to decide whether two objects are equal. Previously [5], one of us (PG) has argued that copying should be needed only rarely in object oriented programs; nevertheless, copying is sometimes necessary, and most object oriented programming languages provide some kind of facilities for copying. Equality testing is certainly sometimes necessary, too, but the details of how to perform the comparison can be subtle.

For the purposes of this discussion, we assume that programmers need to make copies of objects and to compare objects. The two operations are related in that we expect "`X` is a copy of `Y`" to imply "`X` is equal to `Y`", with appropriate interpretations of "copy" and "equal". Complexity arises because the desired meanings of "copy" and "equal" depend on features of the objects involved and on the circumstances in which the operations are used. We will argue:

- there are several plausible meanings that can be assigned to "copy" and "equal";
- a programming language cannot provide appropriate default versions of "copy" and "equal" for all situations; and therefore
- a programming language should provide a set of standard operations that help programmers to construct appropriate copying and comparison operations with minimal effort and maximal security.

Section 2 provides a basis for the discussion. Section 3 discusses the problems involved in copying and comparing objects, including the special problems that arise with cyclic structures. Section 4 describes the facilities for copying and comparing provided by various programming languages. Section 5 demonstrates a possible implementation of our proposals and, finally, Section 6 presents our conclusions.

2 Background

In this section, we introduce an object model that provides a basis for the subsequent discussion and some terminology. The object model can be seen as the "greatest common divisor" of most object oriented programming languages, as simple as possible for presenting the topics of this paper. It is most suited to typed, class-based languages.

In our object model, an object contains zero or more attributes. Each attribute is one of:

- a basic value;
- an object; or
- a reference[1] to an object.

[1] This covers both "reference" and "pointer" in `C++` terminology.

A *basic value* is an indivisible attribute in the object model. In most programming languages, Booleans, characters, and integers are basic values. A string might be considered either as a basic value or as a vector of characters.

The model permits nested objects: an object may be an attribute of another object. Some programming languages (e.g., `C++` and Eiffel) provide nested objects but most object-oriented languages allow an object to contain only references to other objects. The model does *not* allow for references to nested objects, a situation that can arise in `C++` but not in most other languages. The model permits an object to contain a reference to another object. Cycles are possible: a chain of one of more references may lead back to the original object.

In this paper, we assume that the objects under discussion belong to a single identifier space. The additional complications of copying and comparing in distributed object systems are beyond the scope of the present work.

2.1 Essential and Accidental Attributes

We distinguish essential and accidental attributes of an object.[2] An *essential attribute* is indisputably a part of the object; an *accidental attribute* is another object that is related in some way to the object in question but is not a part of it. For example, if the object in question is an instance of class `Car`, we would consider the attribute `engine` to be essential but the basic value `distanceTravelled` and the reference `owner` to be accidental. The distinction between "accidental" and "essential" is orthogonal to that between "reference" and "containment". The model permits all four possibilities. When an attribute is represented by a reference, it is the referent object itself, not the reference, that is the accidental or essential attribute.

Accidental attributes are intended as a generalization of associations. An association is a "structural relationship between peers" where "peers" are classes at the same conceptual level [3]. An association is a kind of accidental attribute but it is not the only kind. Associations are usually implemented as references to other full-fledged objects, although more elaborate implementations have been proposed. But objects may also contain counters, flags, descriptors, and other attributes that are needed by the application software but are conceptually not part of the object.

The distinction between essential and accidental is not always obvious. As a rule of thumb, the relationship between two objects is an association (and therefore accidental) if destroying one object does not logically entail destroying the other, otherwise one object is an attribute of the other. Similarly, an attribute is accidental if removing it from the object does not destroy the basic integrity of the object.

Example 1. Figure 1 provides a simple if rather contrived example of this terminology. The class declaration introduces a `Detector` which is responsible for monitoring the performance of a pump. Each detector has: a pointer to its own

[2] This distinction is based loosely on Aristotle's categories.

```
class Detector
{
public:
    . . . .
private:
    Counter *counter;
    Clock *clock;
    Pump *pump;
    long startTime;
};
```

Fig. 1. The class `Detector`

`Counter` object, used to count events; and a pointer to a unique `Clock` object, shared by all detectors. It has a reference to the pump it is monitoring and, finally, a basic value `startTime`. The attributes `counter` and `startTime` are essential; the attributes `clock` and `pump` are accidental because they are not part of the object.

2.2 Values and Objects

Following MacLennan [14], we distinguish *values*, which are immutable abstractions, and *objects*, which are containers with mutable attributes. For example, `true` and `false` are immutable, Boolean values. In contrast, a `Switch` object might have a mutable attribute with a Boolean value that, at different times, is either `true` or `false`.

In many languages, there is an implicit assumption that basic values are "values" and that classes define "objects" in MacLennan's sense. This leads to confusion for entities such as `String` which should arguably be implemented as immutable (in accordance with the view that strings are basic values) but is usually implemented as a mutable class for efficiency. For example, Java addresses this confusion by providing both an immutable class, `String`, and a mutable class, `StringBuffer` [2, page 172] and CLU has both mutable and immutable versions of the structured type constructors [12].

Our object model distinguishes *mutable* and *immutable* (`const` in C++). This distinction is orthogonal to that between simple and structured objects. We allow both mutable, simple objects, such as a `Counter` that contains an updatable integer, and immutable structured objects, such as a binary tree with an integer at each node.[3] Figure 2 shows examples of each of the four categories.

References are not values in MacLennan's sense because the meaning of a reference depends on the existence of its referent. Many formal object models postulate a fixed, given set of object identifiers, but they must then prohibit the use of meaningless identifiers by suitable integrity constraints. By contrast,

[3] Of course, binary trees can also be represented with mutable objects.

	Mutable	**Immutable**
Simple	`Switch`	`Integer`
Structured	`Person`	`BinaryTree`

Fig. 2. An orthogonal classification

one would not imagine a constraint saying that, depending on the total state of the object system, some integer values must not appear as any attribute of any object.

An immutable object may in general contain references to mutable objects. We use the term *strictly immutable*[4] for the important special case in which this is not allowed. The contents of a strictly immutable object is a *pure value.*

Example 2. The distinction between mutable and immutable is useful but has some subtle ramifications. Consider an application with an immutable class `Rectangle` with integer attributes `length` and `width`. An application can save space by creating one instance of each desired size of rectangle and sharing these instances amongst clients.

Suppose that the same application has a class `Point` with mutable integer attributes `x` and `y`. An instance of class `GraphicalObject` is an object with attributes `shape`, referencing a `Rectangle`, and `position`, referencing a `Point`. Using these classes, the application can create structures like that shown in Figure 3, in which `GraphicalObject`s share `Rectangle`s and `Point`s. The instances of `Rectangle` are immutable objects; that they are shared is undetectable to the program. The instances of `Point` are mutable objects; that they are shared is crucial to the behaviour of the system, because changing the coordinates of a `Point` will cause movement of all of the `GraphicalObject`s referencing it.

The use of two or more names to refer to a single object is called "aliasing" and is sometimes considered to be undesirable. Sharing of both mutable and immutable objects is an important feature of object modelling [5]. Certainly, unintended aliasing can cause serious and subtle errors. But, in many situations, multiple references to a single object are the most appropriate way to model real-world relationships.

2.3 Abstract and Concrete Values

For many objects, it is useful to distinguish the *abstract value* of the object from its *concrete representation.* For example, if the object is a set, its abstract value consists of the members of the set, but its concrete representation might be an array, a list, a hash table, or some other kind of data structure. In CLU, this idea is elevated to a principle [13], and the language provides explicit syntax

[4] "Deep-immutable" would also be suitable.

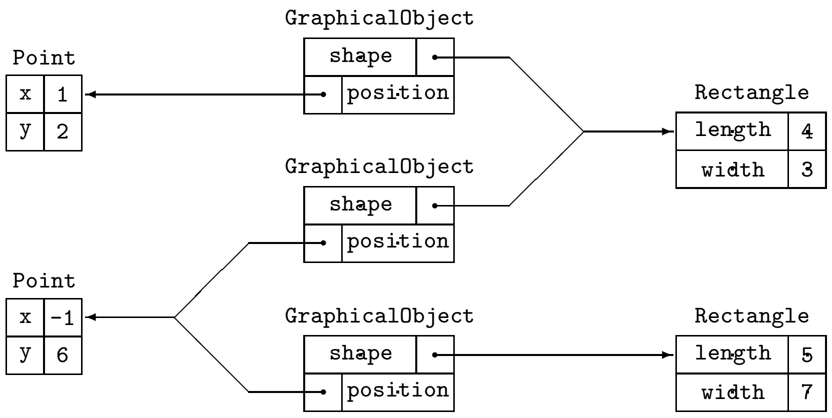

Fig. 3. Using `GraphicalObject`s

for switching between the interface, or external view, and the representation, or internal view. (We avoid the word "conversion" because abstract values do not exist directly either in the language or in programs.) Most object oriented programming languages, however, do not provide explicit syntax for this distinction, and the distinction between abstract values and their representations is mostly in the mind of the programmer.

It is quite natural to think of abstract values in connection with comparison operations. Suppose, for example, that we have two objects `X` and `Y`, each representing sets, and each using an unordered list to represent a set. We would consider `X` and `Y` to be equal if their lists were, respectively, $[2, 1, 3]$ and $[3, 1, 2]$, because both lists share the abstract value $\{1, 2, 3\}$.

It is perhaps less obvious, but equally important, that copying operations should respect abstract values. For example, we might want to represent a set as a binary search tree while we are building it and as an ordered array for retrieval operations. The conversion from tree to array could be accomplished by a copy operation that recognized the need for a change of representation but preserved the abstract value of the set.

2.4 Inheritance

It is usually best not to inherit copying and comparison operations from superclasses because the inherited methods do not process attributes introduced by the subclass. To use a familiar example, if the subclass `ColouredPoint` inherits its copy and comparison methods from the superclass `Point`, the attribute `colour` will be neither copied nor compared. Nevertheless, a copy or compare method in the subclass can often make use of the corresponding method in the superclass and should do so if the programming language provides an appropriate mechanism, as most object oriented programming languages do.

If the programming language supports multiple inheritance, copy and comparison methods from one or more superclasses can be used to construct the corresponding methods in the subclass. Object oriented programming languages with multiple inheritance typically provide appropriate mechanisms for calling methods from multiple superclasses. A few languages provide standard method combinations that do not require explicit invocation: for example, "before methods" and "after methods" in CLOS [7, page 50].

With either single and multiple inheritance, the programming language can provide some support for implementing copy and comparison methods in subclasses. Our approach, described in Section 5.3, avoids some of the common problems, such as covariant redefinition of equality. It is clear, however, that the programming language cannot generate the subclass methods correctly in all cases.

3 Copying and Comparing

In this section, we discuss copying and comparing operations in detail, the consequences of cycles in structures, and the problems of operations between instances of different classes.

3.1 Copying

The word "copy" is used loosely to mean several different things. In this paper, we use particular words for various copying operations, as follows.

– *Assign* means "update a reference" and does not involve copying the contents of objects.
– *Replace* means "copy data from an object into another object that already exists".
– *Clone* means "create a new object and copy data from an existing object into it".

The object from which values are obtained is called the *source object.* The object that is changed or created by the copy operation is called the *target object.*

To clarify these definitions, suppose we have the situation shown in the left part of Figure 4 The letters `X` and `Y` are variable names in a program text; the boxes are run-time objects; the letters $\mathcal{A}$ and $\mathcal{B}$ indicate the values of the objects' attributes; and the arrows indicate the relation between names and objects. The target object is `X` and the source object is `Y`. The right part of Figure 4 shows the various situations that can arise after different kinds of copying operations. $\mathcal{B}'$ indicates a fresh copy of the value $\mathcal{B}$. We have used a neutral, procedural syntax for each operation. In each case, the first argument is the name of the target object. We note that:

– the operation `assign` (reference assignment) creates an alias — afterwards, `X` and `Y` both refer to the same object;

- after the operations `assign` and `clone`, the reference `X` has changed and the object $\mathcal{A}$ becomes inaccessible, or "garbage", unless there are other references to it; and
- `clone` could be implemented by first creating a new, uninitialized object and then using `replace` to initialize its attributes.

In `assign`, the type of Y can be a subtype (subclass) of the type of X (inclusion polymorphism). For `replace`, we require the objects `X` and `Y` to be of the same type; this restriction will be discussed in Section 3.4.

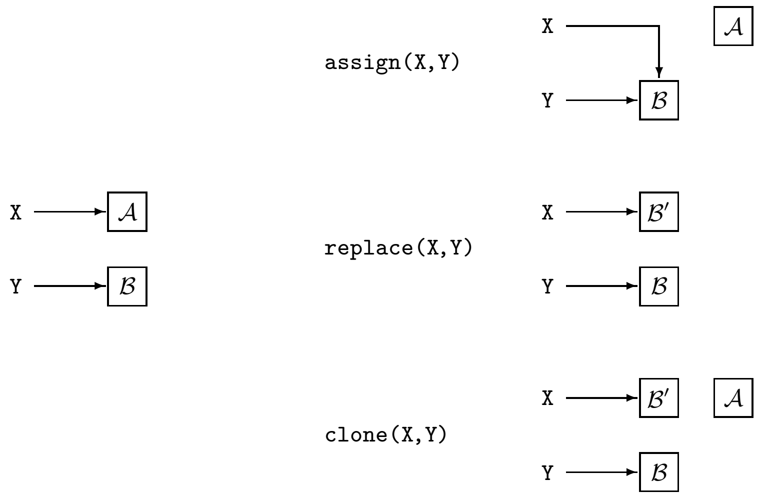

Fig. 4. Copying Operations: before (left) and after (right)

We can further categorize copying operations by their "depth". A *shallow* operation copies, but does not trace, references. A *deep* operation traces references and applies a copy operation to their referents. The distinction between shallow and deep does not apply to reference assignment. A "shallow replace" operation replaces attributes in the target object but not in objects referenced by the target object. A "deep replace" operation replaces non-reference attributes in both the target object and in objects referenced by the target object. Deep replacement requires the source and target object to be isomorphic structures. For example, deep replacement of one list by another list would require both lists to have the same number of items. It would be possible to provide a deep replacement operation that succeeded if the source and target had isomorphic structures and failed otherwise.

Swapping can be seen as a generalized form of replacing. The naive implementation (which requires a temporary intermediate object, one clone operation, two copy operations, and a deletion) is inefficient for complex objects. A more efficient, customized swapping method could be implemented automatically.

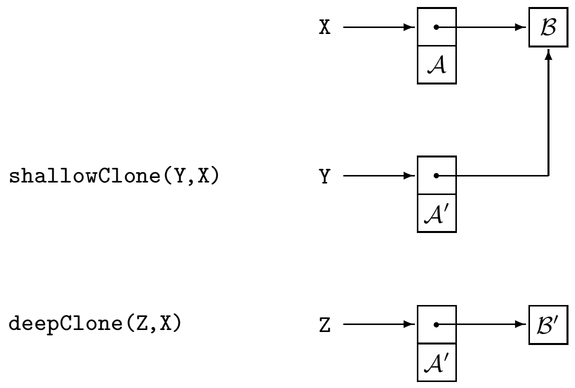

Fig. 5. Shallow and deep clones

Shallow cloning and deep cloning are distinct operations. In Figure 5, `Y` is a shallow clone of `X`. As in Figure 4, primes indicate newly created values. Note that the object $\mathcal{B}$ is shared by `X` and `Y`: shallow cloning may introduce aliasing. `Z` is a deep clone of `X`; an object and its deep clones should normally be disjoint. Approximately the foregoing definitions of shallow and deep cloning have been used in the literature of object oriented systems for many years [8]. Unfortunately, the definition of deep cloning breaks down when cyclic structures are involved! We discuss the application of these definitions to cyclic structures in Section 3.3.

In principle, there is an infinite number of possible ways of cloning a structure. We can define $\texttt{clone}^K(\texttt{X}, \texttt{Y})$ as follows: $\texttt{clone}^0(\texttt{X}, \texttt{Y})$ is the same as $\texttt{assign}(\texttt{X}, \texttt{Y})$; and $\texttt{clone}^K(\texttt{X}, \texttt{Y})$ for $K > 0$ creates a new object and assigns it to `X`, copies value attributes, and performs $\texttt{clone}^{K-1}$ on reference attributes.

Languages that provide cloning operations usually provide $\texttt{clone}^1$ (shallow copy) or $\texttt{clone}^\infty$ (deep copy). However, the Smalltalk method `deepCopy`, described in Section 4.6, performs $\texttt{clone}^2$. The shallow and deep operations are not generally useful. In most cases, "shallow" is too shallow[5] and "deep" is too deep. In order to be generally applicable, copying operations should respect the semantic properties of objects rather than merely their syntactic properties.

3.2 Comparing

There are two principal ways of comparing objects: we can check equality or identity. If `X` and `Y` are objects that have similar structure and whose corresponding attributes are equal, we say that `X` is equal to `Y`. Note that this definition is recursive but terminating: we are defining equality of complex objects in terms of equality of simpler objects. An identity comparison, on the other hand, de-

[5] Embedded objects can be at least a partial help.

termines whether the names X and Y refer to the same object; the attributes of X and Y are not even considered.

In their description of CLU, Liskov and Guttag state that "it should be impossible to distinguish between equal objects" [13, page 93]. For mutable objects, this implies that equality is identity. Otherwise, it would be possible to distinguish apparently equal objects by making a change to one object and then checking whether the other object changed. Two immutable objects are equal if their abstract values are the same. If X and Y refer to immutable objects with equal attributes, the programmer cannot determine whether X and Y refer to the same object or to different objects with the same value.

As with copying, we can define shallow, deep, and "depth-K" equality comparisons in the sense of the preceding section. A shallow comparison compares references but does not trace them, a deep comparison traces references and recursively compares their referents, and a "depth-K" comparison traces at most K pointers from the original object. As with copying, this conventional but naive approach works only as long as the object structures do not contain cycles.

We use four binary operators to denote comparison relations:

$\mathtt{X} =^0 \mathtt{Y}$ means "X and Y are references to the same object";
$\mathtt{X} =^1 \mathtt{Y}$ means "X and Y are shallow equal";
$\mathtt{X} =^\infty \mathtt{Y}$ means "X and Y are deep equal";
$\mathtt{X} \cong \mathtt{Y}$ means "X and Y are structurally equal".

Figure 6 compares the evaluation of these relations for different pairs of objects. We note that identity implies shallow equality and shallow equality implies deep equality. In general, equality at depth K implies equality at all depths greater than K. Examples (4) and (5) show that deep equality and structural equality are independent. The distinction between deep and structural equality has rarely been mentioned in the literature. Even such recent work as [1] does not consider structure equality at all but just studies three different formalizations of conventional deep equality.

3.3 Cyclic Structures

Objects that contain direct or indirect references to themselves introduce a further complication into copying and comparing. Figure 7 is a simplified version of a diagram by Meyer [17, page 249]. The original object is X; the objects Y and Z represent possible outcomes of making a copy of X. Using our definitions:

- $\mathtt{Y} =^1 \mathtt{X}$, implying $\mathtt{Y} =^\infty \mathtt{X}$, but not $\mathtt{Y} \cong \mathtt{X}$;
- Y is a shallow clone of X; and
- $\mathtt{Z} =^\infty \mathtt{X}$ and $\mathtt{Z} \cong \mathtt{X}$.

According to the definitions of Section 3.1, a deep clone of X would be an infinite list.

	X $=^0$ Y	X $=^1$ Y	X $=^\infty$ Y	X $\cong$ Y
(1)	`true`	`true`	`true`	`true`
(2)	`false`	`true`	`true`	`true`
(3)	`false`	`false`	`true`	`true`
(4)	`false`	`false`	`false`	`true`
(5)	`false`	`false`	`true`	`false`

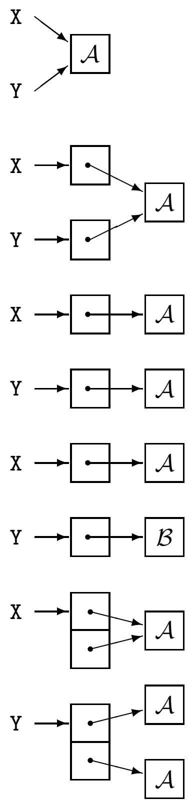

Fig. 6. Four kinds of equality

According to the definitions of Eiffel [16,17], in contrast, `Z` is a deep clone of `X`. Clearly, a complete implementation of deep cloning in Eiffel must detect cyclic references and create corresponding cyclic references in the target object.

A similar problem occurs when the object structure contains no (directed) cycles but there are several distinct paths to one object. In this case, conventional deep copying does not get into infinite recursion but produces a target object with a different structure than the source object. For example, in Figure 6(5), `Y` is a deep copy of `X`.

The distinction between shallow and deep cloning, at first glance, seems to be related to the preservation of structure but is in fact almost orthogonal. To see this distinction, observe that, in Figure 7:

– `Y` and `X` have the same reference attribute values but different structures;
– `Z` and `X` have the same structures but different reference attribute values.

One could argue that even the definition of shallow cloning and shallow equality should be modified to take self-references into account. Such structural

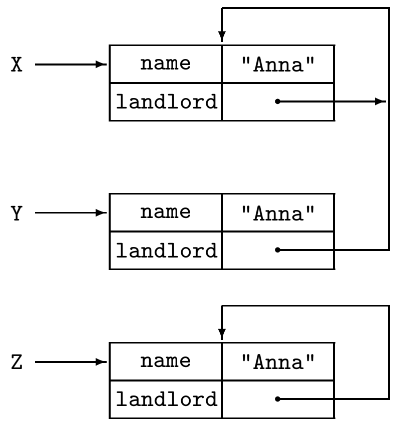

Fig. 7. The problem of self-reference

shallow equality would imply structural equality, just as conventional shallow equality implies deep equality. In that case, `Z` would be a structural shallow clone of `X` but `Y` would not be. Similarly, structural variants of depth-K cloning and equality can be defined.

In general, we can draw a distinction between *value-preserving* cloning operations and *structure-preserving* cloning operations. Which operation is the most appropriate in a particular situation depends on the nature of the application and cannot be deduced simply by examining the objects.

If the kind of deep replacement operations described in the previous section are required, it would be useful to be able to compare the structures of two objects without considering the values of their non-reference components. Because structural comparisons can be expensive, it would also be useful to offer also a three-valued comparison operation: equal by structure and leaf values; equal by structure only; and unequal by structure.

Meyer [16, 305–307] notes that comparison of cyclic structures introduces problems similar to those involved in cloning cyclic structures. In Figure 8, we would like both $\mathtt{A} =^{\infty} \mathtt{B}$ and $\mathtt{C} =^{\infty} \mathtt{D}$ to be true, but it is not obvious how to arrive at these results. (The attributes `f` and `b` are intended to suggest "forward link" and "backward link" respectively.) Meyer's technique for deep comparison of objects `X` and `Y` is to recursively compare references from `X` and `Y` under the assumption that $\mathtt{X} =^{\infty} \mathtt{Y}$. For example, in the deep comparison of `A` and `B` in Figure 8, since we have

$$\mathtt{A.b} =^{0} \mathtt{B.b} =^{0} \mathtt{Void}$$

and

$$\mathtt{A.f.f} =^{0} \mathtt{B.f.f} =^{0} \mathtt{Void}$$

we need only show that

$$\mathtt{A.f.b} =^{0} \mathtt{B.f.b}.$$

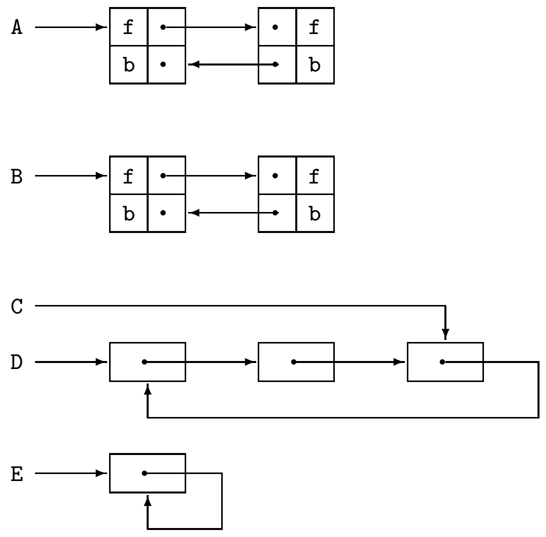

Fig. 8. Deep comparisons

In fact, `A.f.b` $=^0$ `A` and `B.f.b` $=^0$ `B`, and the result follows from the assumption `A` $=^\infty$ `B`. This technique is, of course, analogous to a proof of the correctness of a recursive function in which we use the correctness of the function as an induction hypothesis. Note that even Meyer's definition of deep equality does *not* require structural equality: for instance, `D` $=^\infty$ `E` in Figure 8.

One of the reasons that cycles can occur in a structure is that components of the structure may have "back-pointers" to components that contain them. "Threads" in a threaded tree are an example. For copying or comparing the entire structure, the rules that we have given apply correctly. Problems arise, however, if we try to copy or compare a component of such a structure: for example, a leaf of a threaded tree. In such situations, it is not feasible to provide general methods, and we must resort to class-specific methods, as described in Section 5 below.

3.4 Operations Across Classes

As mentioned in Figure 3.1 and Figure 3.2, both replacement and comparison are more complicated if the objects belong to different classes. Suppose that the objects are `X` and `Y` and that their classes are C_X and C_Y, respectively. There are four cases to consider for `replace(X, Y)`.

C_Y **is a subclass of** C_X**.** In this case, `Y` will in general have at least the attributes of `X`, and possibly additional attributes. Copying from `Y` to `X` is safe in the sense that all of the attributes of `X` will be defined, but information in the additional attributes of `Y` will be lost ("sliced"). This is what happens by default in C++ and some other languages. The abstract value of the object may thus not be conserved; a better solution would be to require an explicit slicing operation (i.e., a projection) before the assignment.

C_X is a subclass of C_Y. Copying from `Y` to `X` is undesirable and not allowed directly in any well-known language, because the additional attributes of `X` will be left undefined (retain their old values). Such copying is possible in C++ if the object `X` is just addressed through a reference of type C_Y.

C_Y and C_X have a common ancestor class C_Z. In this case, only the attributes inherited from C_Z will be copied. This is still less likely to make sense than the previous case, of course. Even this kind of copying will happen in C++ if both objects are addressed through a reference of type C_Z.

C_X and C_Y are not related by inheritance. It may be feasible to define sensible copying and comparison operations, but this will have to be done by the programmer.

Similar considerations apply to comparison: only the attributes common to the classes C_X and C_Y will be compared, which will in most cases not give the results that the class designer would like to have. The equality operator is *not* generated automatically in C++, but if it is defined for a class, it will be inherited by all subclasses that do not redefine it.

Evidently, the only simple and easily understandable way to handle replacement between objects of different classes is the principle we chose in Section 3.1: the operation should fail. Thus, `replace` must either give a failure/success code as a return value, or be able to raise an exception.

In comparison, the equivalent simple principle is that objects of different classes can be compared for equality but the result is always `false`. Non-trivial comparison is performed only between objects of the same exact class, so we have a kind of “type-safe covariance” (cf. Section 2.4) for the equality operation.

4 Language Comparison

In this section, we discuss the facilities for copying and comparing provided by several popular or otherwise relevant object-oriented languages. The languages are introduced in alphabetical order.

4.1 BETA

BETA [15], like its precursor Simula [9], does not provide copying or comparing operations automatically. This is a reasonable design decision because, as we have shown, it is not possible to provide the right functions in all contexts.

4.2 C++

C++ leaves most of the work of copying and comparing to the programmer.[6] The compiler provides a default assignment operator and a default copy constructor

[6] Information about C++ was obtained from the ISO/ANSI Draft Standard available at `http://www.cygnus.com/misc/wp/` and elsewhere.

for a class if the programmer does not define these methods. In our terminology, the default assignment operation "replaces" and the copy constructor "clones". (More precisely, the copy constructor initializes an object that has already been created.) Both operations are shallow ("memberwise") and will give rise to memory management problems if the programmer relies on them for classes that contain references. Consequently, `C++` programmers are encouraged to define their own assignment operators and copy constructors for classes with reference (pointer) attributes.

`C++` provides the operator `==` for built-in types but does not provide a comparison method for user-defined classes. The programmer must provide an overloaded version of the `==` operator for any class whose instances must be compared. The operator `==` with pointer arguments can be used for identity comparisons. Caution is required because, in the presence of multiple inheritance, a `C++` object may have several different addresses [18, page 57].

There is a bias in `C++` in favour of copying objects rather than sharing them. If data members are not pointers and objects are allocated on the stack, then assignment, argument passing, and returned objects all involve copying. The default assignment operators and copy constructors provided by `C++` are consistent with this bias. As classes get more complex, however, complex objects are more likely to be represented as pointer structures. The default methods are inappropriate for pointer structures and programmers must manage copying for themselves. Lalonde and Pugh [11] consider the bias towards copying to be a deficiency of `C++` and discuss the implications of this bias for `C++` programmers. Style guides for `C++` programmers often recommend the use of pointers and heap allocation rather than containment and stack allocation to avoid the overhead and complications of copying [10, 19].

4.3 DSM

Although the language DSM [20] is no longer widely used, we include it in this discussion because it is one of the few languages that addresses the issues raised in this paper in a comprehensive manner. Rumbaugh points out that there are a variety of operations that should be propagated to the objects of a structure. These operations include not only `copy` (clone), which we have discussed, but also `save`, `destroy`, `print`, and others. On the other hand, replacing and comparing are not treated because the approach applies only to unary operations.

In DSM, reference attributes *in* classes are replaced by relations (relationships, associations) *between* classes; this approach is quite common in data modelling but less common in programming languages. Propagation is controlled by *propagation attributes* that are part of the semantic relationship. (Obviously, Rumbaugh is using "attribute" in a different sense than that of this paper.) For example, for the class `Car`, the `part-of` relation has the attribute `propagate` and the `made-by` relation has the attribute `shallow` in an example of [20]. Consequently, when we copy a `Car`, the new object will contain copies of the parts of the original `Car` but a reference to the same manufacturer as the original `Car`.

A relation in DSM is always bidirectional, corresponding to a reference attribute and its inverse. In the `Car` example, the `made-by` relationship has the `propagate` attribute in the inverse direction, indicating that copying a manufacturer copies all the cars manufactured by it. We would certainly prefer the `none` attribute, meaning that the new company must start without any cars manufactured yet. However, our disagreement with Rumbaugh shows only that the semantically correct propagation of cloning cannot be decided automatically by the language or compiler, since even reasonable designers can have different opinions on it.

The DSM principle could obviously be applied also to reference attributes. It is more fine-grained and complicated than our proposal, especially because the attributes are specified separately for each operation. Consequently, a programmer might inadvertently define very strange semantics for a relation. DSM also takes into account the complete graph structure of objects and relations, as we do in the proposals of Section 5.

4.4 Eiffel

Attributes in Eiffel classes are usually represented by references. It is possible to declare an attribute of a class (or a variable in general) as `expanded`, in which case it will contain an embedded object without separate identity, rather than a reference. It is also possible to declare an entire class as `expanded`, in which case all variables whose type is that class will contain objects rather than references to objects. Basic types are regarded as special cases of expanded types.

The Eiffel type hierarchy is bounded above by `ANY`, an ancestor of all classes, and bounded below by `NONE`, a descendant of all classes. Methods for copying and comparing are defined in class `ANY` and may be inherited or redefined by user classes. Most of these methods exist in three forms. The basic form is the one that we consider here. In addition to the basic form, there may also exist: a "frozen" form, indicated by the prefix `standard_`, that cannot be redefined; and a form that allows either or both of its arguments to be `Void` — that is, a null reference.

In Eiffel [16, 295–307]:

– If `X` and `E` have expanded types, the statement `X` := `E` is a copy operation.
– If `X` and `E` have reference types, the statement `X` := `E` is a reference assignment.
– The method `is_equal` tests for shallow equality. (The operator "`=`" may be used for the same purpose.)
– The statement `X.copy(Y)` makes `X` a shallow copy of `Y` and ensures that the expression `X.is_equal(Y)` yields `TRUE`.
– The expression `clone(X)` creates a new object, `Y`; shallow copies the attributes of `X` to `Y`; and returns a reference to `Y`. The (useless) expression `clone(X).is_equal(X)` yields `TRUE`.
– The method `deep_equal` tests for deep equality.

– The methods `deep_copy` and `deep_clone` "replicate an entire data structure, starting at the source and creating new objects as needed". Both methods ensure that `deep_equal` is satisfied. The distinction between `deep_copy` and `deep_clone` is not clear in the original specification of Eiffel [16]. Probably, `deep_copy` was intended to effect a replacement of non-reference attributes and a deep cloning of reference attributes. It is not mentioned at all in later descriptions of the language [17].

After `X.copy(Y)` or `X:=clone(Y)` has been executed, the objects `X` and `Y` may contain pointers to the same object. After `X.deep_clone(Y)` or `X.deep_copy(Y)` has been executed, `X` and `Y` are disjoint data structures. Deep operations detect cyclic references and process them appropriately.

Eiffel provides an unusually large collection of methods for copying and comparing, and is the only language we know besides DSM that can handle cyclic structures correctly. Nevertheless, the probability that one of the methods provided would be precisely correct for copying or comparing complex objects seems rather small, and the programmer has no mechanism for incrementally modifying these methods without losing the built-in handling of cyclic structures.

4.5 Java

Java [2] provides single inheritance with the class `Object` at the root of the class hierarchy. A class other than `Object` inherits either `Object` or another class — Java does not provide multiple inheritance for classes — but it may inherit any number of *interfaces*.

The root class `Object` provides a method `clone` that creates a new object and then shallow-copies the attributes of the source object to it. A class can support `clone` in its basic form, redefine it, or prevent its instances from being cloned.

A class that requires a copying method different from `clone` obtains it by redefining the default implementation. The object must first execute `Object.clone`, which allocates space for the new object and initializes its attributes as described above, and then overwrite any attributes for which the default values are incorrect.

Although Java does not provide a general method for copying other than `clone`, some of the standard classes (e.g., `Vector` and `String`) provide specialized operations.

Java provides the operator `==` for identity comparisons. The class `Object` provides a method `equals` which defaults to identity comparison. In Java standard classes, `equals` is redefined to provide an appropriate value comparison. User-defined classes can use the default version of `equals` or provide an implementation of `equals` that is appropriate to their needs.

4.6 Smalltalk

A Smalltalk[7] object is *immutable* if its class has no methods that change the values of its attributes. An *identity object* has exactly one instance. For example, a Smalltalk implementation provides an identity object `true` that is never cloned; you are allowed to "copy" `true` but all you will get is a reference to the original object. All identity objects are immutable.

The Smalltalk statement `X` := `E` evaluates the expression `E`, obtaining an object, and makes `X` a reference to that object. All Smalltalk implementations provide methods `copy` and `shallowCopy`, both of which return a shallow copy of the source object. Some Smalltalk implementations also provide a method called `deepCopy` that returns a new object in which references have been replaced by shallow copies of their referents. Note that this is $\texttt{clone}^2$, not a deep copy, in the sense of Section 3.1 of this paper.

Smalltalk provides two comparison operators: "`==`" for identity comparison and "`=`" for equality comparison.

Identity comparison is an ordinary object operation in Smalltalk, which means that it can be redefined in a user class. (It is a Smalltalk programmers' *convention* that users do not redefine "`==`" but the language does not prevent redefinition.) This approach is unfortunate, for two reasons. First, identity is a basic object concept whose meaning should not be changeable [8]. Second, there is a performance penalty for what should be a very rapid test, because object operations are dynamically bound.

5 Semantic Copying and Comparing

The traditional classification of copying and comparing that we have presented — reference, shallow, and deep — is based on the representation of objects rather than on their abstract values. Practical applications frequently require reference operations that are usually provided by the language. The shallow and deep operations provided by the language, if any, are often not suitable in practice. In general, a compiler cannot infer the abstract value of an object from the program text and therefore cannot provide appropriate copy and comparison operations for complex classes. The question that we address is: what facilities should a language provide to simplify the task of defining these operations?

What does it mean, for example, to copy an instance of `GraphicalObject` (see Figure 3)? The target object should presumably share the attribute `shape` of the source object but should have its own attribute `position`. Alternatively, if the target object is to be a member of a group of objects, its position should be that of the group.

[7] Information about Smalltalk was obtained from the FAQ by David N. Smith available at `http://www.dnsmith.com/SmallFAQ/`.

5.1 Attribute Classification

In practice, the distinction between “essential” and “accidental” that we made in Section 2.1 is not always sharp enough to use as a basis for implementation: we need to identify attributes that require special treatment. It seems useful to classify attributes as follows (better terms might be invented):

– structure (composite link, essential object reference);
– normal reference (association) or value;
– accidental reference (association) or value; and
– special attribute.

We give below the equivalent DSM propagation modes (propagation attribute values, see Section 4.3). Note that in DSM these apply only to reference attributes.

The structure attributes (composite links, DSM: `propagate`) are those which are to be followed in both copying and comparing, using standard algorithms for directed graphs to obtain or check graph isomorphism (structure equality). They need not be restricted to point from a composite to its parts; thus directed cycles are possible.

Normal attributes (DSM: `shallow`) are copied or compared as such, in the sense of shallow copying or comparing.

Accidental attributes (DSM: `none`) are not compared at all. In copying, an accidental attribute is assigned a default value if such is given in the class definition. Otherwise, an accidental reference attribute is set to null. Some value *types* may also have natural default values, but, for example, it is questionable to use zero as a default value for all integers.

The special attributes (DSM: `none`) would not be touched by the standard operations. If special attributes are present, the standard operations must invoke class-specific methods to handle them. However, the class-specific methods could access also other attributes and call other methods. Defining them would be allowed even when there are no special attributes.

We can divide the actions of both copying and comparison on an object into three phases:

1. the propagation of the operation over the composite links (if any) to the adjacent objects;
2. handling the normal and accidental attributes; and
3. the class-specific actions.

The execution order between the first part and the other parts is immaterial in cloning and comparing; they could even proceed concurrently. In comparison, the order between the second and third parts is also immaterial, except possibly for performance reasons. In cloning, the order can be important, because the special actions can access also the normal and accidental attributes.

In replacing (and swapping, if that operation is desired), phase 1 must first be propagated to the end, and only if the whole structures are found to be equal are phases 2 and 3 performed for all involved objects. In some cases there might be something that must be done with the old attribute values of the target object. This is straightforward if the language provides garbage collection but can be very awkward otherwise.

5.2 General Proposals

As a partial answer to the problems raised in the preceding sections, we propose the following guidelines.

1. The language should provide:
   - a built-in reference assignment operator; and
   - a built-in identity comparison method.

   The assignment operator allows programmers to create multiple references for a single object — in other words, to introduce aliasing. Although *unintended* aliasing may be harmful, the object model sometimes requires multiple references to an object. The identity comparison enables a programmer to determine whether two names refer to the same object.
2. The language should *not* provide separate public methods for shallow and deep copying and comparison, but only one copy method and one comparison method for each class. The designer of the class, rather than its clients, should choose appropriate semantics for these methods.
3. The language should provide syntactic mechanisms for:
   - distinguishing mutable and immutable classes;
   - distinguishing essential attributes and accidental attributes; and
   - specifying when deep copies or comparisons of reference attributes are required.

   An explicit distinction between mutable and immutable classes enables the compiler to make a number of optimizations. An explicit distinction between essential and accidental attributes enables the compiler to generate the default copy and comparison methods described in item 4 below. The depth specifications enable the programmer to customize these methods.
4. The implementation should provide:
   - an optional default copy procedure for each user-defined class; and
   - an optional default comparison method for each user-defined class.

The copy procedure shallow-copies each essential attribute of the source object, unless the programmer has indicated that an attribute should be deep-copied. The copy procedure does not copy accidental attributes but should provide appropriate default values for accidental attributes in the target object. Similarly, the comparison method shallow-compares essential attributes, unless the programmer has requested deep comparison, but does not compare accidental attributes.

5.3 Applying the Proposals

To illustrate the application of these guidelines, we suggest ways in which they might be incorporated into C++ in a way would fulfill our requirements. We do this to demonstrate the feasibility of our proposals rather than in the realistic expectation of their adoption into C++. In particular, we do not address the more arcane aspects of C++, such as private or protected inheritance. But note that static attributes are irrelevant to copying and comparing and that, for other attributes, it does not matter whether they are public, protected, or private.

Standard C++ already contains some of the features that we need:

– the operator "`=`", used with pointer arguments, provides reference assignment;
– the operator "`==`", used with pointer arguments, provides identity comparison; and
– the keyword `const` can be used to distinguish between mutable and immutable objects (and, in fact, enables mutability distinctions of finer granularity than this).

There are four ways in which we might make the distinctions between essential and accidental attributes, and between deep and shallow operations.

1. We could introduce new keywords. This is incompatible with C++ style, which prefers to overload existing keywords (e.g., `static`) in order to avoid breaking old code.[8] Another disadvantage of new keywords is that the compiler must be modified.
2. We could introduce a convention for marking variable names. For example, we could use the prefix `a_` to indicate an accidental attribute. We could also use the prefix `e_` to indicate an essential attribute, although this would be redundant.
   Similar conventions have been proposed for other purposes. For example, Lakos [10, page 91] suggests using the prefix `d_` for class data members and `s_` for static members. The problem with conventions, however, is that the compiler will not act on them.
3. We could use pragmas to make the distinctions. Pragmas are less offensive than keywords[9] and a compiler is allowed to ignore a pragma that it does not recognize.
4. We could design arbitrary extensions and use a preprocessor to translate the extended language into C++.

We are currently applying the fourth approach, using a preprocessor for C++. The preprocessor, which is based on earlier work [4] and will be described in detail elsewhere [6], performs a number of tasks of which the following are relevant for this discussion:

[8] "Proposing a new keyword . . . . never fails to cause a howl of outrage" [21, page 152].
[9] But: "Too often, `#pragma` seems to be used to sneak variations of language semantics into a compiler and to provide extensions with very specialized semantics and awkward syntax" [21, page 425].

```
class Detector
    public void startPump () { ... }
    deep Counter *counter;
    long startTime;
    accidental Pump *pump;
    accidental Clock *clock;
```

Fig. 9. Preprocessor input for class `Detector`

- Programmers can indicate that attributes of a class are `deep` or `accidental` in the sense of Section 2.1 of this paper.
- The preprocessor constructs default copying and comparing methods that follow the conventions that we have described with respect to normal, accidental, and deep attributes.
- Programmers can override the actions of the copying and comparing methods for special attributes.
- The preprocessor supports multiple, non-virtual inheritance. Virtual inheritance can be supported but we do not describe the (somewhat messy) details here.

Figure 9 shows the class `Detector` of Figure 1 as it would be presented to the preprocessor. We have included a method, `startPump`, to show how public features are declared; in practice, of course, there would be other functions, including a constructor and a destructor. The listing also shows the syntax for distinguishing shallow/deep attributes and essential/accidental attributes; there are no keywords for “private”, “shallow”, or “essential” because these are defaults.

First, we describe the way in which the preprocessor handles the qualifiers `deep` and `accidental`. Unqualified attributes are shallow-copied and shallow-compared. For `deep` attributes, the structure graph is explored until either non-reference attributes or reference attributes not marked `deep` are encountered. Attributes marked `accidental` are ignored by default comparison methods; the copying methods set copied attributes to a suitable default value, such as `NULL`.

Second, we describe the methods generated by the preprocessor for a general class. Figure 10 shows the file generated for a class `Z` with two parents, `X` and `Y`.[10] The classes `HAX` and `HAY` are the highest ancestors of `X` and `Y`, respectively. These could be the same class (fork-join inheritance) but, in general, both `X` and `Y` could have more than one highest ancestor.

The preprocessor generates functions as required. In Figure 10, the user has provided none of the functions and the preprocessor has generated a complete set. However, if the user had provided `operator=`, the preprocessor would not have generated `operator=` and would have generated `defaultReplace` only if

[10] The preprocessor actually generates a header file, a definition file, and a documentation file. It also includes more white space than shown here.

```
class Z : public X, public Y {
public:
    virtual Z * clone();
    virtual Z & operator=(Z & other);
    virtual bool operator==(const HAX * other) const;
    virtual bool operator==(const HAY * other) const;
private:
    int iz;
    virtual Z & defaultReplace(Z & other);
    virtual bool defaultEqual(const Z * other) const;
    };

Z * Z::clone() {
    Z *result = new Z;
    *result = *this;
    return result;
    }
Z & Z::defaultReplace(Z & other) {
    ( (X &) *this ) = other;
    ( (Y &) *this ) = other;
    iz = other.iz;
    return *this;
    }
Z & Z::operator=(Z & other) {
    return defaultReplace (other);
    }
bool Z::operator==(const X * other) const {
    return
        typeid(* this) == typeid(* other) &&
        defaultEqual((Z *)other);
    }
bool Z::operator==(const Y * other) const {
    return
        typeid(* this) == typeid(* other) &&
        defaultEqual((Z *)other);
    }
bool Z::defaultEqual(const Z * other) const {
    if ( ! ( (X *)this == (X *)other ) ) return false;
    if ( ! ( (Y *)this == (Y *)other ) ) return false;
    if (iz != other->iz) return false;
    return true;
    }
```

Fig. 10. Declaration and implementation of class Z

the user's `operator=` called it. The defaults for comparison are similar. Note that objects of different classes are never considered equal.

We have not yet addressed the issue of cyclic structures. It is not possible to decide statically whether a structure may contain cycles, but the assumption that every structure might be cyclic introduces considerable overhead because mark bits are required. In the general case, a global view of the object structure is needed. Such a view can in principle be built on an existing language if it has sufficiently powerful reflective facilities, as does Smalltalk (although it is surprising that the standard methods of Smalltalk are so defective in this respect). In the static variety of object oriented languages, the basic mechanisms must be built-in, as they are in Eiffel but in no other well-known language. It should be possible to define incremental methods to adjust the copying and comparison to the exact semantics required for specific classes.

Our attribute classification scheme partly solves the problem of cyclic references because only cycles consisting of *deep* references are relevant, and these should not be very common. If deep references are used only for composition, there will be no directed cycles. If sharing of parts is not allowed, there will not even be undirected cycles.

6 Conclusion

Copying and comparing are operations that cannot be generated automatically from a syntactic object description. Nevertheless, most object oriented programming languages provide some support for copying and comparing, either in the form of default methods in root classes, or in some other way.

The subtleties involved in copying and comparing non-trivial objects are such that simple-minded attempts by the compiler to provide suitable methods are likely to be of little use. Consequently, some languages provide only a basic set of methods and leave the rest of the work to programmers.

We have shown that copying and comparing methods for a class can be generated automatically if the compiler or preprocessor is given a few hints about the way in which the attributes of the class are used. In large systems with thousands of classes, automatic generation of these methods could save a considerable amount of work.

The correct handling of object structures that are not simply trees is impossible or at least very cumbersome (depending on the language) to program on the class level. Therefore, this facility should be offered by the language (or standard libraries), but with suitable hooks for the class-specific handling of some attributes.

Acknowledgments. Peter Grogono's part of the research described in this paper was supported by the Natural Sciences and Engineering Research Council of Canada. A significant part of Markku Sakkinen's work was performed at the Department of Computer Science and Information Systems, University of Jyväskylä.

References

1. Serge Abiteboul and Jan Van den Bussche. Deep equality revisited. In T.W. Ling, A. O. Mendelzon, and L. Vieille, editors, *Deductive and Object-Oriented Databases: Fourth International Conference, DOOD '95, Singapore, December 4-7, 1995, Proceedings*, number 1013 in LNCS, Berlin and Heidelberg and New York, 1995. Springer-Verlag.
2. Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, second edition, 1998.
3. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language: User Guide.* Object Technology. Addison Wesley, 1999.
4. Peter Grogono. Issues in the design of an object oriented programming language. *Structured Programming*, 12(1):1–15, January 1991.
5. Peter Grogono and Patrice Chalin. Copying, sharing, and aliasing. In *Colloquium on Object Orientation in Databases and Software Engineering (ACFAS'94)*, Montreal, Quebec, May 1994.
6. Peter Grogono and Markku Sakkinen. A view and interface generator for C++. Technical report, Department of Computer Science, Concordia University, November 1999.
7. Sonya E. Keene. *Object-Oriented Programming in COMMON LISP.* Addison-Wesley, 1989.
8. Setrag N. Khoshafian and George P. Copeland. Object identity. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 406–416, September 1986.
9. Bjørn Kirkerud. *Object-Oriented Programming with Simula.* Addison Wesley, 1989.
10. John Lakos. *Large-Scale C++ Software Design.* Professional Computing Series. Addison-Wesley, 1996.
11. Wilf LaLonde and John Pugh. Complexity in C++: A Smalltalk perspective. *J. Object-Oriented Programming*, 8(1):49–56, March/April 1995.
12. Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual.* Number 114 in LNCS. Springer-Verlag, Berlin and Heidelberg and New York, 1981.
13. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.
14. Bruce J. MacLennan. Values and objects in programming languages. *ACM SIGPLAN Notices*, 17(12):70–79, December 1983.
15. Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language.* ACM Press/Addison Wesley, 1993.
16. Bertrand Meyer. *Eiffel: the Language.* Prentice Hall International, 1992.
17. Bertrand Meyer. *Object-oriented Software Construction.* Object Oriented Series. Prentice Hall, second edition, 1997.
18. Scott Meyers. *Effective C++.* Addison-Wesley, 1992.
19. Steven P. Reiss. *A Practical Introduction to Software Design with C++.* Wiley, 1999.
20. J. Rumbaugh. Controlling propagation of operations using attributes on relations. In N. Meyrowitz, editor, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 285–296, September 1988.
21. Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994.