J. P. Rosen

What Orientation Should Ada Objects Take?

Ada has often been blamed for not being *really* object oriented, in the sense that it provides only static inheritance (through derived types), but not a full inheritance mechanism. Many proponents of the so-called object-oriented languages (OOLs, see [8] or [13] for example) have claimed that a full inheritance mechanism is absolutely necessary for true object orientation.

On the other hand, Booch [1] has defined a method he calls object-oriented design (OOD), which does not use inheritance. Many Ada teams have used this method successfully, feeling that their designs are really *oriented* according to abstractions of real-life *objects*, and still do not feel a need for inheritance. This makes them quite uneasy when reading about the necessary features of object orientation. To make things more complicated, Booch has advocated inheritance in his recent book [2]. The 9X revision of the standard will introduce inheritance into Ada, but not in the same ways as *pure* OOLs; Ada opponents will certainly take this opportunity to claim that the magic label *object oriented* still does not apply to Ada.

In this article, we will discuss various forms of object orientation and how they apply to Ada, and see how the new features of Ada 9X will impact the design of Ada programs. We will address the issue here only from the point of view of a language that is directed toward software engineering. Other uses of the object paradigm, especially in organizing data for AI applications or data bases are not within the scope of this work. We will compare Ada mainly to two challengers: C++ because it is a language that is currently attracting considerable attention, and Eiffel because it is a language that has specifically been designed to embody the concepts of inheritance in a software engineering approach.

Object Orientation

What actually makes software "object oriented" is that the basic conceptual units are abstractions of real-life objects:

> Object-oriented design is the construction of software systems as structured collections of abstract data type implementations [7].

Since an object can only be described by a set of properties and a set of actions it has on other objects, an abstraction of an object is a programming unit that gathers both data structures and program structures. This is achieved through the use of abstract data types (ADTs) which provide encapsulation and locality, a property that simplifies maintenance since all properties of a given aspect of the problem domain belong to a single piece of software.

From this point of view, Ada is *more* object oriented than current OOLs, because it allows definition of types that model more accurately the constraints of real-life objects. In languages such as Eiffel or C++, basic data types are still determined by the underlying hardware, and it is not possible to define types whose constraints are those of the real-world objects they are supposed to represent. Consider, for example, a need as simple as an integer type that is known to range from 0 to 40,000, something the Ada programmer would simply declare as:

```
type MY_TYPE is range 0 .. 40_000;
```

In Eiffel or C++, checking that values are within bounds would require making it a class, and redefining all operations, including assignment, to preserve value-semantics on assignment. Moreover, there is no portable way of ensuring that the underlying integer type used to represent the value can accommodate the whole range, therefore forcing the use of the longest available integer type on the machine, and almost certainly wasting space . . . Needless to say, *in practice,* nobody makes the effort, and good old type INTEGER (Eiffel) or int (C) is used throughout the programs.

ADTs, however, are not sufficient to make object orientation effective. In a large project, there are many objects, and some kind of organization as well as factoring of common properties is needed. How this factorization is achieved is what differentiates OOLs from the Booch approach.

In a classical OOL, objects are organized into a hierarchy of classes. General classes contain features com-

mon to many subclasses, and specialized subclasses implement only behaviors that are common to a subset of the parent class's objects. We will call this method "Object orientation by classification first" (or simply stated *classification*). This organization is akin to the classification of biological species [6]. For example, a class GRAPHICAL_OBJECT would gather all the properties of graphical objects, and a class RECTANGLE (derived from GRAPHICAL_OBJECT) will contain only those aspects that are peculiars to rectangles. Of course, since a RECTANGLE *is* a graphical object, it must still have the properties of GRAPHICAL_OBJECT. It is said to inherit from its parent type, GRAPHICAL_OBJECT.

OOD provides an alternative method by recognizing that objects are made of different parts, and that the same parts can be used to make a variety of higher-level objects. Each object is designed as an assembly of lower-level components. We will call this method "Object orientation by composition first" (or simply stated *composition*). An important feature of composition is that there is no need to know how a component is made in order to use it. A set of parts that form an object constitutes an abstraction layer. Therefore, the basic structuring concept in OOD is organization of objects according to abstraction layers. This is what is generally used in engineering, and has been especially successful in the electronic industry.

Why do so many people believe that classification is the only "true" object orientation? Perhaps because OOL designers place a heavy emphasis on inheritance, which is clearly necessary only to classification. Most books about OOLs generally address only classification; they start with a discussion about ADTs (and [8] is *really* great at that). Then, there is a short statement such as "we generally think of objects as being gathered into classes . . .", and then the discussion goes on to classification and inheritance. They just do not mention that other organizations are possible. Hence, the "natural necessity" of classification and inheritance. Few papers have advocated composition as a different, effective, object orientation [9, 10, 11].

Of course, both dimensions are present in any object. At the programming level, languages must support them. Composition is absolutely necessary even for classification-oriented languages, and Eiffel offers a syntactic construction that makes a clear distinction between composition and classification. Although it is possible to ignore classification for composition-oriented languages, classification is helpful for describing data with variant structure. Ada has many features that favor composition, but only a limited support for classification; new features of Ada 9X will add more facilities. The important factor, however, is that, at design level a main design direction must be chosen; the other aspect will remain second.

An Example

To stress the difference between classification and composition, we will take a simple example, which is actually the very example used by Turbo-Pascal® to introduce inheritance [3].

First, the notion of a mathematical point is introduced; Using Ada syntax, we can define it as follows (using Ada syntax):

```
type POINT is
  record
    X,Y : INTEGER;
    -- Of course, with Ada, INTE-
    --GER should not be used
    here...
  end record;
```

Then, the question arises of how to define a screen pixel. The explanation goes on like this: "Of course, a pixel is a kind of POINT, with an extra feature: it can be visible or not." Thus the need to extend the type POINT to make it a PIXEL appears, and inheritance comes naturally. In particular, if there is a MOVE procedure defined for POINTs, you do not need to rewrite it, since it will apply to PIXELs as well . . . as long as the behavior for POINTs suits your needs for PIXELs, which in general will not be the case. If you need a different behavior, you will have to write the code anyway.

However, is it so obvious that a pixel *is* a point? A point is a mathematical object, while a pixel is a dot on a screen . . . Quite different objects actually. An alternative approach would be to define a PIXEL as a self-standing entity, defined by various attributes. One attribute is its position (which can be described using the notion of mathematical point). Another one is whether it is visible or not. This would be the composition approach. With composition, the MOVE procedure you define on PIXELs bears no *visible* relationship to the MOVE on POINTs (anyway, the fact that a PIXEL includes a POINT is generally hidden in a private part). This does not preclude the implementation of MOVE for PIXELs from using MOVE on POINTs, therefore reusing the code for POINTs, but this dependency will remain hidden. Thanks to overloading, the same identifier (MOVE) can be used both for POINTs and PIXELs, providing uniformity from a user's point of view.

A benefit of the classification approach is that if you want to change the properties of a mathematical point, all of its descendants (including PIXEL) will be automatically updated. The drawback is that users of PIXEL *know* its relationship to POINT, and can apply POINT's methods (even those that are not necessarily meaningful for a PIXEL) to it. The implementation is visible: if the designer of PIXEL wants to change the implementation strategy, for example by deriving from something other than POINT, the designer is unable to do so without disturbing the code of PIXEL's users, since those users may have used methods inherited from POINT.

Linguistic Aspects

OOLs have brought a number of new notions into programming languages: polymorphism, dynamic binding, and inheritance. The popularity OOLs have achieved necessarily means their use carries a number of benefits. Many of these benefits, however, are most noticeable only when compared with older programming languages such as Pascal or C; the important question is how do these benefits apply to Ada? In other words, are those new notions neces-

sary to provide those benefits? We will discuss the tools provided by Ada to satisfy the same needs, both using current Ada and the 9X improvements.

Encapsulation

Encapsulation is the ability to gather into one place all aspects related to a given abstraction of a real-world object. Many benefits of object-oriented technologies come from abstraction and encapsulation, which are common to both composition and classification.

C has a very poor encapsulation mechanism (the file can be used as a primitive means for packaging entities). The class concept in C++ has provided a means of logically relating a data type and its associated subprograms, and a number of C++ projects use classes only for that purpose, without using inheritance. In Eiffel, classes are the only structuring feature, used for encapsulation as well as for defining compilation units.[1] Neither of these languages allows for nested units (i.e., all abstractions must be defined at the library level, and there is no way to define an encapsulated construct local to a given entity). It should be noted that having nested units and inheritance in the same language creates technical difficulties in order to preserve safety of the language; Ada 9X will achieve this, thanks to its strong typing features, at the cost of extra (and somewhat difficult) compilation—or even run-time—checks.

Ada features the package—a very powerful encapsulation mechanism. Packages can be used for many purposes, including but not limited to, building abstract data types. Let us stress this point: an Ada package *is not* necessarily an ADT; a package containing the definition of a private type together with associated operations is the Ada way of defining an ADT. Being more general, the package is a bit less adapted to the particular need of building classes; but this is outweighed by its ability to provide a flexible means of satisfying any encapsulation needs, including local encapsulations.

It is unarguably true that the class mechanism brings a real enhancement to languages such as C or Pascal by providing them with an encapsulation mechanism that has been sorely missing. With the package, Ada already has a more powerful tool. Moreover, 9X will further improve it by providing a second level of organization, with the notion of hierarchical libraries. The class mechanism is clearly not indispensable for the purpose of encapsulation.

Polymorphism and Dynamic Binding; Type Extensions

Polymorphism is the ability for a variable to hold various data structures. In most OOLs, a variable may hold not only values of its own type, but also values of a type that inherits from the variable's declared type. A consequence is that the designer of a type with an OOL does not know what the actual type is that the variable will hold at run time: anyone reusing the type may add (later) new variations. In Ada, a polymorphic variable must be explicitly declared as such, using a type with discriminants and variant parts. No logical dependency is necessary among the different forms the variable can take, but those forms are fixed by the type declaration. If a new form is necessary, the original type must be modified.

Dynamic binding is the ability of an operation to perform differently according to the actual type of the value a polymorphic variable is holding. This is performed automatically with OOLs. At run time a (formal) operation dispatches to the correct implementation according to a hidden descriptor that uniquely identifies the current variant. In Ada, a dispatching operation must be explicitly provided in the form of a procedure accepting a parameter of the polymorphic type, which uses a case statement, driven by the discriminant, to call the appropriate operation. There is no need to have a one-to-one mapping between the operation on the polymorphic type and the corresponding treatment for a particular variant. However, all this dispatching must be explicitly coded.

By requiring explicit control over polymorphism and dynamic binding, the Ada solution gave complete control to the designer of an ADT over all possible uses of the type, and complete compile-time type checking. If a variant is added to a discriminant and the corresponding code is not added to the relevant case statement, the program will simply not compile. This is the reason it was chosen in the first place, at the cost of more explicit code and recompilations.

This trade-off, however, of security against ease of evolution is now being regarded as too strict for many applications. Dynamic binding provides a simpler way of adding new variants to an existing type, no change in code is necessary. This makes evolution and addition of new features easier. The drawback is that some type checking must be delayed until execution time. With 9X, both paradigms will be available, under the responsibility of the initial designer. *Tagged types* will allow for type extensions, inheritance and dynamic binding for class-wide operations. This will make applications easier to evolve, and will allow for easier interfacing with foreign environments. Nevertheless, extensions will be allowed only if the initial type is tagged: an important consequence is that those new features will be available only if the initial designer specifically gave permission; if necessary, complete control over all usages of the type can still be guaranteed.

The language will now leave the responsibility of whether security or ease of evolution should prevail as a design choice. This careful approach will be a major improvement from the point of view of program evolution, while still retaining, if required, the secure approach that makes Ada unique for many critical applications.

Reusing Algorithms

One of the needs that many previous programming languages failed to satisfy is the ability to provide reusable algorithms (i.e., algorithms that can be applied to a variety of types). The basic point is that most algorithms are not applicable to an arbitrary type. For example, the simple algorithm used to exchange the contents of two variables:

```
TEMP  := X;
```

[1]The latest version of Eiffel has brought a new packaging construct, not mentioned in [8]. Eiffel is such a moving target that any reference to it should mention the precise version used.

```
X    := Y;
Y    := TEMP;
```

asserts that at least assignment is available for the type of variables X, Y and TEMP, something that is not necessary true, e.g., for a limited type in Ada (or a file type in Pascal). The main problem when designing a way of providing reusable algorithms is how to express those common properties that are required of any type to which the algorithm is applicable.

In OOLs, this is obtained by assuming that all types having those common properties belong to a given class. For example, consider the algorithm used to move a figure on a screen:

```
Erase(Figure);
Set_position(Figure,
To => (New_X, New_Y));
Draw(Figure);
```

This algorithm is applicable to any graphical object that can provide an operation to erase it from the screen, one to set its current location, and a third to draw it again. In Eiffel, this is expressed as follows:

```
deferred class GRAPHIC_OBJECT
export
  MOVE, DRAW, ERASE,
  SET_POSITION
feature
  X, Y : INTEGER;
  DRAW is
    deferred
  end; -- DRAW
  ERASE is
    deferred
  end; -- ERASE
  SET_POSITION(To_X : INTEGER;
  To_Y : INTEGER) is
    deferred
  end
MOVE(To_X : INTEGER; To_Y :
INTEGER) is
do
  ERASE;
  SET_POSITION(To_X, To_Y);
  DRAW
end; -- MOVE
end -- GRAPHIC_OBJECT
```

(the **deferred** clause means that any type that inherits from GRAPHIC_OBJECT must provide a definition for the actual feature). Now, every object such as RECTANGLE or CIRCLE will inherit from GRAPHIC_OBJECT; DRAW, SET_POSITION and ERASE will be redefined for each, but the algorithm for MOVE will automatically be available.

The Ada solution expresses more directly the basic requirement: for any data type on which certain operations are available, a given algorithm is applicable. This translates into:

```
generic
  type ITEM_TYPE is
  limited private;
  with procedure DRAW(ITEM : in
  ITEM_TYPE);
  with procedure SET_POSITION
  (ITEM : in out ITEM_TYPE; TO :
  in POSITION);
  with procedure ERASE(ITEM : in
  ITEM_TYPE);
procedure MOVE(ITEM : in out
ITEM_TYPE);
```

This generic unit can in turn be instantiated by providing the corresponding types and operations. Note however that different instantiations need not bear any conceptual relationship; imagine, for example, a remotely controlled toy crane: *erasing* may be matched with an operation to pick a block up from the floor, *setting position* with moving the crane, and *drawing* with placing the object back on the floor. The MOVE algorithm would therefore be applicable to moving cubes around the floor, although a crane is obviously not a graphic object.

From the point of view of the code actually generated, both solutions are likely to be comparable. The code for the generic unit can be generated only once (no duplication) by replacing all calls to imported subprograms by indirect calls, which is exactly what dynamic binding will use.

The main difference between the two views is conceptual: with OOLS, to reuse an algorithm you have to incorporate your own type into a foreign structure. In the preceding EIFFEL example, you have to make your type inherit from GRAPHIC_OBJECT, therefore stating that your type *is* a figure. With generics, the dependency is reversed: you incorporate a foreign algorithm into the properties of your type, but you do not create any dependency *from* your type *to* the foreign structure. You buy a reusable algorithm from a vendor and incorporate it into your design, but this does not create a conceptual dependency to the provider of the abstraction.

It must be noted that many comparisons of the relative merits of inheritance and generics for code reuse only consider generic type parameters; what makes generics powerful in Ada is the ability to import a type *together* with an explicit list of required operations. Ignoring this fundamental feature simply kills the most powerful uses of generics.

Methodological Aspects

Is classification really a design method? It has long been recognized that languages are not the ultimate solution to the software crisis; development methods are the important factor, and languages are here only to help in applying a given method. Classification is the methodological end that justifies inheritance means.

Classification is not as natural, and certainly not as easy, as many purport. For example, there was a lengthy interchange of messages on the comp.lang.eiffel bulletin board on whether a RECTANGLE should be considered a kind of (i.e., inherit from) POLYGON. The issue here was that a POLYGON featured a method called ADD_VERTEX, that was clearly not applicable to RECTANGLEs. Although there was a great deal of controversy, many concluded the only safe way was to NOT use inheritance, and to consider a RECTANGLE as *not* being a POLYGON.

One must recognize there is currently no widespread methodology for applying inheritance, something akin to OOD and its derivatives (ex., GOOD [12], HOOD [4]). A methodology is taken here as a set of rules and associated tools that guide a developer during the design process, allow the cross-checking of structures, and provide the necessary documentation.

Some even advocate that inheritance should lead to abandoning the whole idea of top-down design. Since the basic idea is to design by reusing and adjusting existing components, object orientation would lead to

bottom-up design. Realistically, no project comprising several hundred thousand lines of code can be completely developed bottom-up, and it should be noted that the electronic industry *does* use top-down design, although eventually everything is built from existing components. Too often, advocating bottom-up design is just an excuse for not having any sound design methodology.

Although classification is a widespread scientific activity, it is not obvious that it can be applied to software development. The problem faced by Linné was to put some organization into *existing* species, while the software designer has to *design new objects*—quite a different task. Note that there is nothing in the electronic industry (which is certainly closer to software development than entomology!) that looks like design by classification (although classification is used to organize *existing* components, but not to *design* them).

Is Classification Used as it Should Be?

Even though classification at first sight appears to be a scientific activity, in practice it appears that inheritance is actually used for a very different purpose. Actually, almost every book, paper, or article that presents classification is self-contradictory. It always starts by explaining classification, where it appears that a project is well-organized from top-level, general classes, to derived, more specialized classes. The topology of the project is thus said to follow the general scientific scheme of species classification.[2] It will be explained later that as projects evolve, there is no need to change existing classes: new classes are derived from older ones, in order not to disturb parts of the software that depend on them; as needs evolve, new behaviors are provided in the newly derived classes. This means that the topology of the project no longer reflects a logical classification, but the evolution of requirements over time. The inheritance graph becomes a kind of stack of archeological layers, and it is impossible to understand it unless you know the complete story of the project.

Another issue is that inheritance is often used not for classification, but just to grab pieces of code. Jaulent [5] gives an example in which he makes a class PARROT inherit from the class HUMAN_BEING because it needs the property CAN_SPEAK . . . This kind of misuse of the inheritance mechanism, especially when multiple inheritance is available, is extremely tempting to the programmer who just wants to "reuse" existing code, while actually introducing a terrible mess in the dependencies between objects.

The Case for Reuse

Of course, reuse is not limited to the case of reusing algorithms mentioned earlier. It extends to reusing objects, subsystems, even full designs. This is certainly a prime concern, and Ada was designed for reuse. What is not generally said is that there are different kinds of reuse, and reuse in the Ada sense is very different from the kind of reuse OOLs provide.

Reusability, as defined in OOLs, is targeted toward *delta* coding. The basic idea here is that no two different projects will ever need *exactly* the same components. Therefore, if the exact abstraction you need is not in the program library, inheritance lets you choose one that is close enough to what you want to do, and just modify (I dare not say "patch") the difference (the delta) between the behavior provided by your ancestor and the one that is needed. This is an efficient means of rapidly developing components tailored to specific needs from existing pieces.

On the other hand, composition promotes standard, opaque components. There is no insight on how they are designed internally, and no way to change the behavior except where explicitly provided for (through generics). This keeps uniformity among all uses and the ability to change the implementation of the component without affecting users in any way (even recompilation is not necessary if you change only a body). This has an unpleasant consequence for the programmer: if no component fits exactly the programmer's needs, then these needs must be adjusted to fit existing components (and the programmer must resist the temptation of designing a specifically tailored component). This is quite a new constraint in software design, but very common in all other engineering disciplines. If you need a 3.456KΩ resistor, what do you do? Use a 3.7KΩ and adjust your design. Of course, working with a set of standard frozen components is much more cost-effective for maintenance and validation.

Let's face it: the kind of reuse offered by Ada is less appealing to the programmer, since it puts constraints on the design phase that will only pay later on. But Ada is intended for long-term projects, not quick prototyping. In the slums, people build houses by reusing old items such as pieces of wood and tires, and adjusting them to their needs. This kind of reuse is often extremely clever, and if you absolutely need a shelter rapidly, it is certainly an efficient method. But will the house withstand a storm? This is certainly not industrial reuse, which is building houses from well-engineered, standard prefabricated components.

Is it Possible to Reconcile Composition and Classification?

Since composition and classification both have drawbacks and benefits, it is tempting to try and reconcile them in a common framework.

Booch tried to define a method in [2] that used composition and classification on equal footing. However, we do not believe such efforts can be successful, and this can be shown by examining the properties of the structural graph of the project.

The graph of an Ada project reflects the dependencies in terms of "with" clauses (i.e., it describes uses of units by other units). This graph is fundamentally *nontransitive:* what this means is that, to understand and act on a given unit, it is necessary to understand the unit itself (of course), the specifications of all "withed" units, and no more. Since adding a new unit adds dependencies only to the unit's immediate neighbors, it can be said that the overall complexity of the full graph grows linearly

[2]Species classification and graphical objects are not the *only* examples taken in *all* books about classification. Is it possible to apply it to anything else?

with N (the number of units).

On the other hand, an inheritance graph reflects classification and is transitive: a given unit depends on its immediate ancestors, on the ancestors' ancestor, and so on. To understand the behavior of a given unit, it is necessary to understand the whole inheritance subgraph that leads down to the given object. This was perceived by Booch since he wrote [2, p. 101]:

There is a very real tension between inheritance and encapsulation. To a large degree, the use of inheritance exposes some of the secrets of an inherited class. Practically, this means that to understand the meaning of a particular class, you must often study all of its superclasses, sometimes including their inside views.

Adding a new unit to a graph will increase the overall complexity by a factor that is proportional to the size of the graph: the global complexity will therefore grow as N^2.

Thus structural graphs and inheritance graphs are incompatible, since they exhibit incompatible properties. There is no way to design using composition and classification at the same time. Although there are always some classification aspects in a composition-oriented design, and conversely, a main direction must be chosen, the other aspect will remain a second-class citizen.

Once again, composition will exhibit a lower complexity and a greater security at the cost of ease of design. For small-sized, quickly developed projects, classification can be an efficient method. But for large-scale, long-lasting projects, composition is necessary to ensure control on the overall complexity.

Conclusion

Inheritance has become very popular because it has brought a number of benefits, including abstraction, encapsulation, reusable algorithms, and more, to languages such as C or Pascal that had no appropriate tool to satisfy these needs. When inheritance is viewed in the context of Ada, the issue is quite different, since many of those needs are already satisfied by other features.

Ada 9X will introduce new mechanisms, including a purposely limited and well-controlled form of inheritance, to the language. This will allow the designer to develop new paradigms, open Ada's usage to other domains, and make interfacing with external environments based on inheritance easier; however, extreme care has been exercised to ensure that improvements in some areas are not made at the cost of other qualities of the language.

This means that Ada object orientation will still not just follow classification, because classification does not answer properly a number of requirements for Ada applications. In general, classification will favor ease of design and rapidly varying specifications such as those encountered when prototyping software, while composition will better match the needs of secure, long-lasting systems, requiring various implementations of the same abstract behavior. The problem domain Ada is intended to address requires object orientation by composition rather than classification.

Ada is sometimes touted being "more than just another programming language." We trust that Ada 9X will be "more than just another object-oriented language." It will have its own form of object orientation, adapted to the problem domain of long life cycle, secure systems, which is not necessarily the form that can be found generally in the literature. ■

References

1. Booch, G. *Software Engineering with Ada,* Second ed. Benjamin Cummings, 1986.
2. Booch, G. *Object Oriented Design with Applications.* Benjamin Cummings, 1991.
3. Borland. *Object Oriented Programming.* Reference Manual of Turbo-Pascal V5.5.
4. European Space Agency. *HOOD Reference Manual,* WME/89-173/JB, Noordwijk, the Netherlands.
5. Jaulent, P. *Génie Logiciel, les Méthodes.* Armand Colin, Paris, 1992.
6. Linné, C. *Systema naturae.* Uppsala 1735.
7. Meyer, B. *Eiffel: programming for reusability and extendibility.* Interactive Software Engineering, 1986.
8. Meyer, B. *Object Oriented Software Construction.* Prentice Hall, New York, 1988.
9. Rosen, J.P. A comparison of object oriented paradigms. *Tenth international workshop on Expert Systems and their Applications* (Avignon, May 1990).
10. Rosen, J.P. Pour une définition méthodologique de la notion d'«orienté objet». *AFCET Interfaces,* n°96, (Paris Octobre 1990). (In French).
11. Rosen, J.P. Object Oriented Paradigms: OOD vs. Inheritance. Invited lecture, *Ada—Europe Conference* (Athens, May 1991). (This document is a registered Ada—Europe document and can be obtained from the Ada—Europe Secretariat, Napier Polytechnic, 219 Colinton Road, Edinburgh EH14 1DJ, UNITED KINGDOM).
12. Seidewitz and Stark. *General Object Oriented Software Development,* NASA Software Engineering Laboratory Series SEL-86-002.
13. Stroustrupt, B. and Ellis, M. *The Annotated C++ Reference Manual,* Addison-Wesley, Reading, Mass., 1990.

CR Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object oriented progamming; D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages, methodologies*; D.2.10 [**Software Engineering**]: Design—*methodologies*; D.2.m [**Software Engineering**]: Miscellaneous—*Rapid prototyping, reusable software*; D.3.2 [**Programming Languages**]: Language classifications—*Ada, object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types, modules, packages*

General Terms: Design, Languages

Additional Key Words and Phrases: Classification, composition, inheritance

About the Author:

J.P. ROSEN is the founder of Adalog, a company specializing in high-level training and consulting in the fields of Ada and object-oriented design. His research interests encompass all aspects related to Ada and OOD, software, components, and interfaces (especially SQL). **Author's Present Address:** Adalog, 27 avenue de Verdun, 92170 Vanves, France, rosen@enst.enst.fr.

®Turbo-Pascal is a registered trademark of Borland International