Teaching Object-Oriented Programming in Ada*

Mordechai Ben-Ari
Department of Science Teaching
Weizmann Institute of Science
Rehovot 76100 Israel
ntbenari@wis.weizmann.ac.il

Abstract

This paper reports on experience teaching object-oriented programming in Ada, with emphasis on constructs that support full OOP in Ada 95. In Ada 95 this support is achieved through the integration of individual language constructs, rather than through a single syntactic entity. These constructs are easy to understand and teach, but the instructor must ensure that students comprehend that object-oriented programming is a programming paradigm, and that arbitrary use of the language constructs does not constitute OOP.

1 Overview

This paper is based on experience teaching Ada for over ten years, both in universities and in industry, in particular, teaching the new version of the language Ada 95 [14].[1] Ada 95 provides full support for object-oriented programming, including type extension and dynamic polymorphism which were lacking in the original standard—Ada 83. Ada's support for OOP is structured rather differently from the support found in languages like Smalltalk and Eiffel that were designed primarily for OOP. (For a detailed comparison see [2].) OOP in Ada is a paradigm that is implemented by integrating individual constructs. The advantage is flexibility in the design of program structure; the disadvantage is that teaching OOP in Ada can be challenging because the paradigms must be taught along with the language constructs.

The paper is divided into the following sections:

- An overview of my experience teaching Ada.

*Appeared in *Journal of Object-Oriented Programming*, 11(6), October 1998, 39–45. A preliminary version of this paper was presented at the Symposium on Teaching Object Technology, Santa Barbara, CA, August 1996.

[1] Elementary Ada textbooks are [7] and [8]; advanced textbooks are [4], [1] and [6].

- A discussion of conceptual difficulties exhibited by students learning the elementary concepts of OOP (supported already in Ada 83).
- A short tutorial on type extension and dynamic polymorphism in Ada 95.
- A report on experience teaching these concepts in Ada 95.
- A note on use of the GNAT compiler by the students.

For readers who do not know Ada, footnotes will be used to explain points of the language and make comparisons with C++.

2 A decade of teaching Ada

Contrary to popular opinion, Ada is not a difficult language to teach or to learn. The reason is that the language is structured as a set of more or less orthogonal constructs. One can start off with Pascal-like data declarations, statements and procedures, then encapsulate code into packages, create abstract data types using private types, and finally introduce advanced constructs as needed:

1. Unconstrained data types.
2. Exceptions.
3. Static polymorphism - generics and overloading.
4. Hierarchial units (child packages).
5. Exceptions.
6. Floating-point and fixed-point calculations.
7. Dynamic polymorphism.
8. Concurrency.
9. Hardware interfaces.

The instructor has a lot of flexibility in constructing computer science courses, using Ada as a common tool but emphasizing different aspects such as software engineering, systems programming, object-oriented programming, information systems and numerical analysis. Furthermore, each of these topics requires only a modest amount of additional syntax, so you can concentrate on teaching the concepts and techniques.

Of course this advantage doesn't come free. The theme of this paper is: because of the flexibility of Ada, the instructor must invest extra effort to teach *paradigms*. It is probably easier to teach OOP in Smalltalk, concurrent programming in occam, numerical methods in APL and string processing in Icon, simply because these languages were designed for a specific paradigm. Ada is a multiparadigm language which scales easily from DSP's [10] to the largest applications, but judgement is required to choose the proper constructs.

The paper also describes some of the misconceptions that students have when studying object-oriented programming in Ada, misconceptions that stem not so much from the language as from the paradigm. The paper should be of interest to instructors teaching OOP in other languages because they are likely to encounter similar, though not identical, problems.

The material is taken in part from mistakes uncovered when checking programming assignments undertaken by the students. The mistakes discussed in this paper were carefully selected for their value as indicators of possible conceptual difficulties, and should not be interpreted as a criticism either of the Ada language or of my students! Almost all of them master Ada during the semester and submit very good final projects.

3 Conceptual difficulties in Ada 83

In the late 1970's when Ada was designed, support for full OOP was considered too risky for a language intended for the development of reliable, efficient software for embedded computer systems. However, the influence of OOP is apparent in Ada 83's support for strong type checking, encapsulation (hierarchial packages), abstract data types (private types), inheritance (derived types) and reuse (generics). The level of support for OOP in Ada 83 has been called *object-based programming*. The following subsections survey some of the difficulties that can arise in understanding abstract data types.

The concept of type

Abstract data types are the basis of object-oriented programming. The first concept that must be learned is that of *type*, defined as a set of values and a set of operations on those values. Unfortunately, this concept is not given the emphasis it deserves in introductory courses. In C you cannot declare new types—**typedef** only renames a type. Even if Pascal is used, the instructor might not insist that students declare new types, especially since there is no encapsulation mechanism.

Since my students did not learn Ada as their first programming language, it was occasionally difficult to convince them to use types other than integer and float. For example, one student believed that array indices must be integers and wrote: [2]

```
type Colors is (Red, Green, Yellow, Blue);
type Intensities is array(
  Colors'Pos(Colors'First) .. Colors'Pos(Colors'Last) ) of Float;
```

instead of using the enumeration type directly as the index type:

```
type Intensities is array(Colors) of Float;
```

Another example

[2]The first declaration declares an enumeration type. For any discrete (enumeration or integer) type T, T'Pos is a function that gives the position of its parameter in the sequence of values of the type. T'First and T'Last return the first and last values of the type, respectively.

```
if Colors'Pos(C1) < Colors'Pos(C2) then ...
```

shows that the student did not think of the type as including the relational operator.

Even Boolean types may not be recognized as types, as seen in the infamous

```
if A < B then C := True; else C := False; end if;
```

which students often use instead of:

```
C := A < B;
```

A similar problem is the use of **others** in a **case** statement whose alternatives already cover all the values of an enumeration type:

```
case C is
  when Red | Yellow => ...;
  when Green | Blue => ...;
  when others => ...;
end case;
```

Clearly this demonstrates a legacy of programming with integers.

It is interesting that many modern languages do not support true enumeration types, justifying the decision by claiming that they are subsumed by general abstract data types. However, I believe that enumeration types still have two important justifications:

- Practically, few programmers will bother creating an ADT for a type with just a few elementary values; instead, they will use integer types unsafely.
- Enumeration types provide a simple way of explaining the main features of types: sets of abstract values—as opposed to concrete representations—and operations which can be predefined or user-defined. Once these concepts are understood, it is a relatively small conceptual step to ADT's.

The concept of value

In fact there is a conceptual difficulty even at the level of the primitive concept of *value*. Both C and Pascal have the strange characteristic that it is possible to define a type (set of values), but the languages offer no construct for expressing values of the type! I am referring, of course, to records and arrays: you must use individual assignment statements for each component. Ada, on the other hand, supports *aggregates* which enable values of these types to be explicitly expressed:[3]

```
Days_Per_Month: array(Months) of Positive :=
  (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
```

[3] An aggregate is a sequence of values, one for each array or record component, enclosed in parentheses. There is a rich syntax for aggregates including both positional and named notation.

```
Today: Date := (Year => 1996, Month => Jan, Day => 29);
```

Students must be told again and again that aggregates are not just syntactic sugar, but an essential part of type checking because an incomplete aggregate is a compile-time error. This is complicated by the fact that students who have seen C array *initializations* do not realize that an aggregate can appear in any context where a value is expected, and that an aggregate can contain arbitrary expressions and not just constants:[4]

```
Days_Per_Month :=
  (Feb => Check_Leap(Today.Year),
  Apr | Jun | Sep | Nov => 30,
  Jan | Mar | May | Jul | Aug | Oct | Dec => 31);

Head := new Node'(Key => I, Next => Head);
    -- Add node to head of list
```

In C++ constructors can be used like aggregates:

```
struct Date
{
  int day, month, year;
  Date(int d, int m, int y) {day=d; month=m; year=y;}
};

Date  d = Date(29, 1, 1996);
      d = Date(d.day+4, d.month, d.year);
```

but, as with enumeration types, it would be preferable to introduce the concept of a compound value using arrays and records before studying constructors.

Abstract data types

An abstract data type, such as a stack, is defined in Ada by declaring a package containing a private type and a set of operations on a private type:[5]

```
package Stack_Package is
  type Stack is limited private;
  procedure Push(S: in out Stack; N: in Integer);
  procedure Pop(S: in out Stack; N: out Integer);
```

[4] In the first example, function Check_Leap is called at run-time and the returned value is used to construct the aggregate. The second example shows an allocator which uses an aggregate to give initial values to the record components.

[5] This is a *package specification* which defines the interface to the package (module). Declarations in the public part are accessible to users of the package, while declarations in the **private** part are not, and are used only by the compiler to generate code. The public name of the type Stack is followed by a completion of the definition in the private part of the package. The word **limited** indicates that predefined assignment and equality are not available for this type.

```
private
  type Vector is array(Integer range <>) of Integer;
  type Stack is
    record
      Data: Vector(0..1000);
      Top: Integer range Vector'Range;
    end record;
end Stack_Package;
```

I teach these concepts as a sequence of 'attempts' to create an ADT:[6]

- Encapsulate the data and operations for a *single* stack in a package; the state is maintained in the package body.[7]
- Now, suppose you need more than one stack. Create a data type by placing a type declaration in the package specification.
- But this is not abstract. So define a private type and complete it in the package *body*.
- Unfortunately, the program is now illegal because it cannot be compiled! When the stack object is created, the compiler needs to know how much memory to allocate. Since a language which cannot be compiled is of little practical use, we compromise on information hiding by placing the completion is *private* part and pretending not to see it. I claim now that a change of implementation requires only recompilation.
- This is not precisely true, however, because the distinction between shallow and deep copy and equality must be considered. Limited types prevent the problem by removing the predefined operations for the type.
- Alternatively, if a private type is implemented as an access type (pointer), the completion of the designated type may be in the body:

  ```
  package Stack_Package is
    type Stack is limited private;
  private
    type Stack_Record;
    type Stack is access Stack_Record;
  end Stack_Package;

  package body Stack_Package is
    type Stack_Record is ...;
  end Stack_Package;
  ```

  This is an approximation to reference semantics: the implementation can be changed without recompilation, at the expense of the indirect access.

[6] For details, see my textbook [4].

[7] This is an *abstract data object*; see Booch [5] and Yehudai [15].

Packages by themselves are just constructs for encapsulation and can contain declarations unrelated to an ADT, even declarations of variables. Students occasionally included variable declarations in packages, giving *state* to a package which was intended to be an ADT.[8] The instructor must repeatedly emphasize the difference between *objects* and *types*, in particular, that objects have state while types are just templates.

Teachers of C++ will encounter a similar problem: variables can be declared in file scope thereby giving state to the ADT.

Another problem is that students declared subprograms in the private part:

```
package Stack_Package is
  type Stack is limited private;
  ...
private
  ...
  procedure Init;
end Stack_Package;
```

This construct is sometimes useful,[9] but in an elementary ADT they are not needed. It is vital to stress that an ADT will (usually) have just public subprograms declared in the specification, in addition to hidden subprograms declared in the body that are needed for implementing the ADT.

The separation of specification from implementation is an extremely powerful tool for encapsulation. Languages which use classes, not only for abstraction but also for encapsulation, are probably easier to teach initially, but they offer less support for the physical aspects of building and managing a large software project.

4 Object-oriented programming in Ada 95

A language is considered to have full support for object-oriented programming if it supports encapsulation, inheritance and dynamic polymorphism. Ada 83's support for encapsulation is excellent and has been further improved in Ada 95.[10]

Inheritance was partially supported in Ada 83. If a package specification declares a type and a set of operations, another unit is allowed to derive a copy of the type, possibly overriding and extending the operations. However, the record type cannot be extended with additional components upon derivation, thus negating the principal purpose of inheritance.[11]

[8] This is the way one implements in Ada what C++ calls static data members; this requirement is relatively rare, certainly in the basic exercises assigned to the students.

[9] For example, a function for default initialization of a record component declared in the private part should itself be declared in the private part. Ada 95 adds additional possibilities (subprograms invoked from child packages and declarations of hidden primitive operations), but a discussion is beyond the scope of this paper.

[10] Child packages support flexible hierarchies of packages which improve name space control and allow packages to share private types.

[11] **with** P is a context clause that makes the declarations in the specification of P accessible to package Q. In the declaration of Derived, the keyword **new** indicates derivation from an existing *base* type.

```
package P is
   type Base is record ...;
   procedure Op1(V: Base);
   procedure Op2(V: Base);
end P;

with P;
package Q is
   type Derived is new P.Base;
   procedure Op1(V: Derived);       -- Op1 is overridden
                                    -- Op2 is inherited from Base
   procedure Op3(V: Derived);       -- Op3 is new for Derived
end Q;
```

Derived types were introduced into Ada 83 as a generalization of the type structure of numeric types of different precisions. They have some significant applications (for example, they enable one to have two equivalent types with different underlying representations), but were never a central topic in teaching Ada.

The technical extensions in Ada 95 required to support full inheritance are surprisingly simple, though conceptually deep. The extension is based on a generalization of *variant records*. In Pascal one can define a variant record, where each variant extends a set of common components with its own specific components. A *tag* component is used in Pascal to distinguish among the variants, limiting access to the components of the current variant;[12] unfortunately, the tag is optional and some compilers do not implement the check. Ada requires the existence of a tag component, called a *discriminant*:

```
type Variant(Color: Colors := Red) is
   record
      case Color is
         Red => I: Integer;
         Yellow => B: Boolean;
         Green => C: Character;
      end case;
   end record;
```

and requires that the tag be checked when components are accessed:

```
V: Variant(Green);     -- Variant with Character component

V.C := 'X';            -- OK
V.B := True;           -- Error
```

The technical extension for OOP in Ada 95 is to allow *tagged records*, which conceptually are similar to variant records except that the tag is implicit. While other languages do not use this terminology, it is consistent with the canonical implementation of inheritance.

[12] C has union's but no tags for automatically checking the type of the current variant.

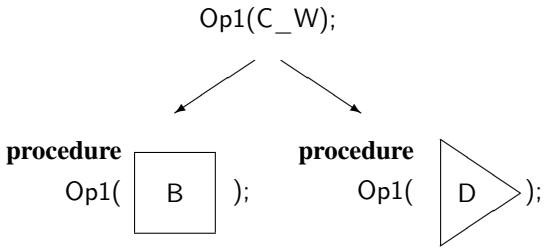

Figure 1: Dynamic dispatching

If a record type is declared as tagged, it can be extended:[13]

```
package P is
  type Base is record ...;
  procedure Op1(V: Base);
  procedure Op2(V: Base);
end P;

with P;
package Q is
  type Derived is new P.Base with
    record
      New_Component: Some_Type;
    end record;
  procedure Op1(V: Derived);       -- Op1 is overridden
                                   -- Op2 is inherited from Base
  procedure Op3(V: Derived);       -- Op3 is new for Derived
end Q;
```

A set of types derived from a common base type is called a *derivation class* (or simply *class*).[14] Given a (tagged) type T, the *class-wide type* T'Class is the discriminated union of all the specific types in the class. This means that a value of T'Class is either a value of type T or a value of some type descended from T, together with an implicit tag which is used to discriminate among the types in the class, that is, to identify the specific type at run-time.

Dynamic dispatching takes place when a value of a class-wide type is used as an *actual* parameter in a call of a primitive operation[15] with a *formal* parameter of a specific type. In the example, the operation Op1 exists in two versions (Figure 1): the original version with a *formal* parameter of type Base and the overriding version with a *formal* parameter of type Derived. The following code will cause dynamic dispatching to the

[13] The syntax **is new** ... **with** indicates that this is a derivation but with additional components specified in the following record.

[14] In C++, the term *class* is used for the construct which defines (an implementation of) a type; the terms class and type are often interchanged.

[15] Since there is no class *construct* to syntactically group operations of a type, the Ada language must specify which operations are *primitive* for a type, and thus are inherited and can be overridden. Roughly, an operation is primitive for a tagged type T if has one or more formal parameters of type T and it is declared in the same package specification as T.

appropriate version, depending on the value assigned to C_W:[16]

```
C_W: Base'Class := ...;

Op1(C_W);    -- Dispatches
```

Tagged types are fully integrated with constructs such as private types and generics, but a discussion is beyond the scope of the paper.

5 Teaching object-oriented programming in Ada 95

While the rule for dynamic dispatching may seem complicated, it can be explained with the help of a diagram such as Figure 1. Hold out your hand and say:

> You are holding a value, but it is of class-wide type so you don't know what the specific type of the value is, only that it is within the class. Since the type *is* in the class, Op1 is defined on values of this type either by inheritance or overriding.[17] The code generated by the compiler automatically checks the tag which identifies the specific type of the value and calls the operation appropriate for this type.

In other languages there does not seem to be a term for the tree (or dag with multiple inheritance) of types derived from an ancestor. In Ada the concepts of class and class-wide type facilitate teaching dynamic dispatching, because there is no need to talk about implicit type conversions as one does in C++.

In general, students quickly comprehend these concepts. The following subsections describe some of the problems that have appeared.

Inheritance overkill

Since inheritance is 'in' there is a tendency to declare every ADT as tagged. For example, some students declared tagged types without any primitive operations. Students must be made to understand that inheritance is only one possible way of constructing data types, and that composition (available in all languages with records) is often simpler and more efficient.

Rosen [13] gives a compelling argument that composition is almost always to be preferred over inheritance, while Meyer [12] almost always prefers inheritance. Teachers should carefully point out the advantages and disadvantages of each of these structuring methods so that students can begin to develop their skills in object-oriented *design*.

[16] Note that unlike C++ which uses a *distinguished receiver* syntax object.operation, Ada uses ordinary subprogram invocation operation(object). The is more flexible (especially when overloading binary operators), but the flexibility must be restricted for efficient implementation.

[17] This holds because you can never *remove* an inherited operation, only override it. Dynamic dispatching is both safe and efficient in languages such as Ada 95 and C++: the appropriate operation for any specific type can be identified at compile time and used to build a jump table.

Class-wide-type overkill

The natural progression is to teach inheritance (type extension and overriding of operations), and only then to teach class-wide types and dynamic dispatching. By this time, some students will have forgotten that an operation declared at the base of a class of derived types can, if not overridden, be invoked for all types in the class (Op2 in the example above). They will use operations with parameters of class-wide types for every operation common to all derived types, and even be seduced into using explicit run-time type information where overriding is called for:[18]

```
procedure Class_Op2(V: Base'Class) is
begin
  if V'Tag = Derived'Tag then ...;
  else ...;
  end if;
end Class_Op2;
```

The correct use of operations with formal parameters of class-wide types is to *create* class-wide values that can be used as actual parameters for dynamic dispatching:

```
procedure Activate(V: Base'Class) is
begin
  Op1(V);   -- Dispatches
  Op2(V);   -- Dispatches
  Op3(V);   -- Dispatches
end Activate;
```

A motivating example can be found in the implementation of a heterogeneous priority queue. Operations such as Insert and Remove will have parameters of class-wide types, and internally dispatch on a relational operator to determine priority.

Explicit tags should always be left to the very end of the sequence of lectures on OOP and the instructor must emphasize that the use of explicit tags is generally indicative of poor design.

Indefinite types

In Ada an *indefinite type* is a type for which you cannot declare an object without providing a constraint to inform the compiler how much memory to allocate. An elementary example is the predefined String type:[19]

```
type String is array(Positive range <>) of Character;
```

A typical use of the type is as follows, where we note the explicit computation of the index constraint (1..2*S1'Length) of the string S2:

[18] V'Tag returns the specific type of the value in V. It can be converted to a string for display. The construct is similar to typeid in C++.

[19] The notation Positive **range** <> means that only the index type Positive is given, not the index bounds which must be supplied by an explicit constraint, or implicitly by an initial value or actual parameter.

```
function Create_Palindrome(S: String) return String;

S1: String := "Hello world";          -- Constraint from initial value
S2: String(1..2*S1'Length);           -- Compute constraint

S2 := Create_Palindrome(S1);
Put(S2);
```

However, the entire computation can be trivially written without variables if you understand that a function result is a value that can be an actual parameter of a further subprogram invocation:

```
Put(Create_Palindrome("Hello world"));
```

Students found it difficult to think functionally and insisted on computing in small steps, storing each value in a temporary variable.

The ability to think functionally is particularly important in Ada 95 because a function can return a class-wide type. Because a class-wide object cannot have its tag changed, it is difficult to use temporary variables:[20]

```
function Remove(From: access Queue) return Item'Class;

V: Item'Class := ...;

V := Remove(From=>Q'Access);    -- Probably an error
Put(V);                         -- Dispatch
```

The assignment is correct only if the item removed from the queue happens to have the same tag (specific type) as the initial value of V.

The better programming paradigm is to avoid the superfluous variable and dispatch directly on the result:

```
Put(Remove(From=>Q'Access));    -- Dispatch
```

The value returned by Remove is of class-wide type and its result can be used to dispatch to the appropriate Put procedure.

Note that this problem is probably specific to Ada because of language design decisions. Languages like Eiffel and Java which use reference semantics do not have this problem since only (implicit) pointers are returned, never actual values. Ada (like C++) uses value semantics for efficiency, but (unlike C++) Ada does not require *explicit* pointers or references to perform dynamic dispatching. The advantage of the Ada design is that you can *use* a (heterogeneous) ADT without using explicit pointers, while an efficient pointer-based implementation can be encapsulated. The disadvantage is that you have to learn how to manipulate objects of indefinite size. Of course you can always use pointers to the class-wide type, but it is worthwhile developing proficiency with the relevant programming paradigm.

[20] V is required to be constrained by an initial value, otherwise, the compiler would not be able to allocate memory for it. An access parameter—see an Ada textbook for an explanation—is used since a function can not have an **in out** parameter.

Class-wide types and variant records

While the motivation of tagged types as implicit variant records is successful, confusion is possible because the visibility rules are different. Consider:

```
type Variant(Color: Colors := Red) is
  record
    case Color is
      Red => I: Integer;
      Yellow => B: Boolean;
      Green => C: Character;
    end case;
  end record;

V: Variant := ...
N: Integer := V.I;          -- Possible constraint error
```

V is called an *unconstrained record* and can be assigned any value whose type is in the union of types.[21] When selecting a component V.I, a check of the tag must be made. If V was in fact assigned the Red variant, the assignment will succeed; otherwise it will raise Constraint_Error. The rule is expressed as follows:

> For a selected_component that denotes a component of a variant, a check is made that the values of the discriminants are such that the value or object ... has this component. The exception Constraint_Error is raised if this check fails. [14, §4.1.3(15)]

There is no question that the component I of the variant record is visible. The record declaration is a single scope and this is the reason that the component names, *even of different variants*, must be distinct.

Students who had used variant records in Pascal or Ada 83 (or union types in C) attempted the same programming style in Ada 95, using an implicit tag instead of an explicit discriminant:[22]

```
type Base is tagged null record;

type I_Type is new Base with
  record I: Integer; end record;
type B_Type is new Base with
  record B: Boolean; end record;
type C_Type is new Base with
  record C: Character; end record;

V: Base'Class := ...;
N: Integer := V.I;          -- Compile-time error!
```

[21] Ada restricts assignment to complete records and does not allow assignment to the discriminant alone to ensure that the type system is not broken.

[22] **null record** is syntactic sugar for an empty record definition.

Here the rule is different:

> The only components ... of T'Class that are visible are those of T.
> [14, §3.4.1(5)]

Since V is of type Base'Class, it has the same components as does Base (namely none), and the selection V.I is meaningless, *even if* the variable V currently contains a value of type I_Type.

The correct programming style is to use a type conversion:

```
N: Integer := I_Type(V).I;
```

If V is in fact of type I_Type, the type conversion will succeed and a value of type I_Type will have a component I that can be selected. Otherwise, Constraint_Error will be raised on the type conversion.[23] The fact that each record type declaration is a separate scope means that components of derived tagged typed can have the same names, even if they are of different types.

This year I taught tagged types without first teaching variant records. In fact, I glossed over the meaning of tagged until I had discussed the canonical implementation of dispatching using jump tables. The experiment was successful—the students understood the concepts of inheritance and class-wide types without confusing them with variant records.

Concurrency

A unique advantage of Ada is the extensive support for concurrent programming integrated within the language. Object-oriented programming in a multitasking environment is an important topic and can be taught in Ada with no additional system-specific overhead.

A project of great educational interest is the implementation of a Linda [9] *tuple space*. (See my paper [3] and the independent work of Lundqvist and Wall [11].) The tuple space is by definition a heterogeneous data structure, which is best programmed using type extension upon inheritance of an abstract tuple. The tuple-space operations are implemented as operations with parameters of class-wide type since *any* tuple can be placed in or removed from the tuple space. Synchronization is implemented using Ada concurrency primitives such as protected objects.

6 Development environments

For many years it was difficult to teach Ada because the only available compilers were expensive commercial products. The situation is now radically changed: a team at the New York University has developed GNAT, a compiler for (all of) Ada 95 that is freely available under the terms of the GNU public license.[24] It uses the GCC code generator and hence is available on all platforms likely to be found in a school. The

[23] The is similar to dynamic_cast of a reference in C++.

[24] The GNAT development team has formed a company—AdaCore Technologies, Inc.—to provide commercial support for their Ada compilers; however, public versions are also upgraded and remain freely available.

distribution bundles integrated development environments for DOS and Windows. In addition, commercial developers offer low-cost academic licenses for their products.[25] Students were able to install and use GNAT with no difficulty.

The entire text of the Ada standard [14] is freely available online, both in ASCII format which is useful for running textual searches, and in hypertext format which makes it easy to navigate the document. I did not expect the students to study the standard in order to understand the language; however, no class time was expended on teaching the standard libraries which are clearly documented in the standard. Some students showed initiative by searching the libraries for random number generators and mathematical functions.

7 Conclusion

Like other modern programming languages, Ada is a large, comprehensive language that must be carefully studied. Fortunately, it can be studied at gradually increasing levels of complexity, starting from a clean Pascal-like language, and then incrementally progressing to advanced concepts like object-oriented programming and concurrency. Most of the language is relatively easy to teach, but you must teach programming paradigms along with the language constructs.

Acknowledgement

I would like to thank Amiram Yehudai and the reviewers for their helpful comments on the manuscript.

References

[1] John Barnes. *Programming in Ada 95*. Addison-Wesley, Reading, MA, 1996.

[2] M. Ben-Ari. *Understanding Programming Languages*. John Wiley & Sons, Chichester, 1996.

[3] M. Ben-Ari. Using inheritance to implement concurrency. *Twenty-Seventh SIGCSE Technical Symposium. SIGCSE Bulletin*, 28(1):180–184, 1996.

[4] M. Ben-Ari. *Ada for Software Engineers*. John Wiley & Sons, Chichester, 1998.

[5] Grady Booch. *Software Engineering in Ada*. Benjamin/Cummings, Menlo Park, CA, 1983.

[6] Norman H. Cohen. *Ada as a Second Language (Second Edition)*. McGraw-Hill, New York, NY, 1996.

[7] J. English. *Ada 95: the craft of object-oriented programming*. Prentice-Hall, Hemel Hempstead, 1997.

[8] M. B. Feldman and E. Koffman. *Ada 95: Problem Solving and Program Design*. Addison-Wesley, Reading, MA, 1996.

[9] David Gelernter and Nicholas Carriero. *How to Write Parallel Programs—A First Course*. MIT Press, Cambridge, MA, 1990.

[25] Up-to-date information on the availability of Ada compilers may be obtained by following the links at the web sites: http://www.adahome.com and http://www.acm.org/sigada.

[10] Patricia K. Lawlis and Terence W. Elam. Ada outperforms assembly: A case study. http://www.seas.gwu.edu/seas/eecs/Research/ada/sigada-website/lawlis.html, July 1996.

[11] K. Lundqvist and G. Wall. Using object oriented methods in Ada 95 to implement Linda. In A. Strohmeier, editor, *Reliable Software Technologies - Ada-Europe '96*, volume 1088 of *Lecture Notes in Computer Science*, pages 211–222. Springer Verlag, 1996.

[12] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall International, Hemel Hempstead, 1988.

[13] J.P. Rosen. What orientation should Ada objects take? *Communications of the ACM*, 35(11):71–76, 1992.

[14] S. T. Taft and R. A. Duff, editors. *Ada 95 Reference Manual: Language and Standard Libraries*, volume LNCS 1246. Springer Verlag, 1997. International Standard ISO/IEC 8652:1995(E).

[15] Amiram Yehudai. Data abstraction: types vs. objects. *Ada Letters*, II(2):46–48, 1982.