Experience in 40 Years of Teaching Ada

Jean-Pierre Rosen
Adalog, 2 rue du Dr Lombard, 92130 Issy-Les-Moulineaux, France. Email: rosen@adalog.fr

Abstract

In this era of newsgroups, forums, and MOOCs, is there a place for traditional face-to-face training? After years of teaching Ada, we present our experience about what is easy and what requires special emphasis in the various aspects of Ada training. We conclude that "passing the message" is what most requires human presence.

Keywords: Ada, training, software engineering.

Back in 1979, shortly after Green was selected by the DoD, Ichbiah gave the first presentation of this brand new language – preliminary Ada. I attended this presentation, and since at that time I was a teacher in a French engineering school, I immediately returned that presentation to the students. I never ceased teaching Ada ever since then. This paper presents my experience after 40 years of teaching Ada.

1 The students

My experience relates to a quite specialized sample of students: they were all engineers already working in a company, mostly with some years of experience, some of them freshly out of school. Most of them were software engineers, although I received some QA people or project managers sometimes.

None of them were complete beginners as far as programming was concerned. Actually, the knowledge of at least one programming language is a prerequisite to my course. This implies that I assumed that the basics of programming were known (although I had surprises at times). This also meant that quite often, students may have had bad habits that needed to be fixed...

2 Teaching the language

The common language basis known to all students is C, plus varying knowledge of other languages from the C-family (C++, Java, C#…).

The previous knowledge of students evolved over time. Some years ago, many students had used Pascal or Pascal-like languages (at least as a first introductory language), making the syntax look more familiar. More recently, a significant number of students had exposure to VHDL, making the language look even more familiar. Python is often mentioned, but not as frequently as one would expect given the current claimed popularity of this language.

Few students have difficulties with the syntax. Moreover, the syntax differences between Ada and the languages students know are easily overcome, given the excellent error messages provided by the Gnat compiler for syntax errors.

When showing the syntax of the basic statements, it is important to stress what's special in Ada (completeness of **case** statements, safety of the **for** loop) and to show that these peculiarities are here on purpose, to serve goals of software engineering. It is also useful to take this opportunity to stress the uniform and consistent design of the language.

Since most students have not been exposed to any block structured language, the notion of being able to declare any construct at any place (like a subprogram or a package within a subprogram) is not easily grasped. It is necessary to explain in great details the nesting of program units, visibility rules, hiding...

Almost all students (except the senior ones) have been trained to object oriented programming with C++ or Java, although none of them really understand where the terms "class" or "method" come from. A bit of methodological introduction, showing the principles of functional vs. object oriented programming is useful, and helps justify the strategy adopted by Ada, since Ada's OOP model is quite original, and, to be honest, not straightforward. Newcomers are surprised that "class" is not a reserved word! With appropriate explanations, it is reasonably easy to explain the difference between a specific type and a class-wide type.

Exceptions are no more a new concept, and many students had (some) exposure to concurrency.

Note that the teacher should not hesitate to cheat a little bit in places, to provide an usable and easy to understand model, even if it is not the exact truth from a language lawyer point of view. For example, I state (tongue in cheek) that the resolution of integer literals is just a kind of overloading resolution (to avoid having to talk about universal integers), or that a package specification cannot contain any form of body (it can, if it contains a generic instantiation).

It is of course important to make comparison with other languages, not to tell that other languages are bad, but rather that they correspond to different requirements about the very purpose of a programming language. C, for example, was intended to be a "portable assembly language" (and it is very good at that); Ada is intended to keep the programmer away from the machine. I often stress this by saying "*C is the best language to program a*

computer; Ada is the best language to develop a software application"[1].

3 Teaching how to use the language

A good Ada course should not teach how to program *with* Ada, but how to program *in* Ada.

At first glance, Ada looks like most other programming languages, and it is easy to use it just like any other language: using only predefined types, using packages just for separate compilation (without any consideration for information hiding), ignoring generics altogether, etc.[2] The first challenge is to have the students understand that the Ada way of *thinking* is radically different from other languages.

For example, the need for information hiding is not obvious to many. This has to be explained first, then packages come as the tool to enforce information hiding.

The habit of writing specifications without even thinking about the implementation is far from obvious, and must be enforced in the exercises. Students tend to think about implementation first ("how can I do this?"), then write the specification as a way of exporting what they have written. It must be explained that the specification expresses client requirements, and that the body is the implementation of the requirements.

Ada has a rich set of data structures, some with no equivalent in other languages (discriminated types, f.e.). The importance of an appropriate choice of data structures has to be stressed, and more generally the need to *design* types and data structures the same way as algorithms. Although students have no issue with understanding the possibility of defining one's own types, they have difficulties to put it into practice. The natural trend of many is to do everything with types Integer and Boolean only.

Finally, it is important to show that the compiler is of great help during the whole coding phase: "the compiler is your friend". Students tend to write everything, and compile only when the code is complete, producing a discouraging flow of error messages. Explain that the Ada compiler is a helpful companion, willing to tell the errors as early as possible, and that it saves a lot of time to compile often, like immediately after each subprogram is written, even if a lot of code is still missing.

4 Typical student errors (and how to react)

There is a small number of errors that students will make almost systematically in the course of exercises. These are good opportunities to hint on important points of the language.

Using enumerated types is not natural for many students: for example, in an exercise that involves a state machine, students tend to declare a Boolean variable for each state rather than a Current_State variable of an appropriate enumerated type.

Hint: ask the student what happens if two states are True at the same time (if they assume that it does not happen, ask if they can prove it); show that this cannot happen with an enumerated type.

When faced with a compilation error, students tend to try to make the error disappear, instead of searching for their own design error. Typically, if a student writes:

```
        D : Duration ;
begin
        D:= 1;
```

He will get an error message:

```
Expected type Standard.Duration, found an integer type
```

Which he will try to fix by writing:

```
        D:= Duration (1);
```

This shows that the student thought "Oh, the compiler wants a Duration, let's convert this to Duration", rather than thinking that Duration is a real type, and therefore that the correct fix is:

```
        D:= 1.0;
```

Hint: Explain *why* this kind of reaction is wrong: when something doesn't compile, it is important to understand the problem, not to find a workaround such that the compiler accepts it. In the end, the student should come to think:

> "Thank you, gentle compiler, for pointing out my design errors"![3]

Error messages sometimes push the students in the wrong direction. For example, students who want to declare an integer type think that the word "integer" must appear in the declaration and write:

```
type My_Int is Integer range 1..10;
```

Unfortunately, the error message points after the **is** and says "missing **new**". Students blindly follow the advice and write:

```
type My_Int is new Integer range 1..10;
```

This results in a type derived from Integer, while the correct fix would have been to delete the word Integer:

```
type My_Int is range 1..10;
```

Hint: explain that what they wrote makes their type dependent on the definition of Integer, while without "**new**

[1] Please don't quote me on the first part of this statement without the second part !

[2] I have seen projects doing this; they didn't get much gain from using Ada, and spent a huge amount of time fighting the compiler.

[3] Ada is the only language where users are happy to have compilation errors!

Integer" the compiler would choose the most appropriate integer type.

Simple tests are often written as this:

```
if Is_Present = True then ...
```

This may seem just like a slightly redundant formulation (that will be optimized away by the compiler anyway), but it actually reflects a fundamental problem: the student thought "if the variable Is_Present contains the value True", not "if my data is present".

Hint: use this kind of error to stress the need for higher level thinking: the programmer should change his mind from "programming a computer" to "expressing a solution to a problem".

5 Hard points

5.1 Vocabulary

The compiler has a special lingo. The Ada standard has a precise definition of every technical term, and the compiler is careful to use the appropriate vocabulary. However, terms may not be obvious to the casual user. For example, a common error message is "invalid use of subtype mark in expression or call"; this puzzles the student (*subtype mark*? What's this?) while it simply means that a type name has been used in place of a variable name – a very common confusion.

Some Ada terms have a different meaning than in other languages. For example, in Ada a *subprogram* is either a *procedure* or a *function*, while in Fortran, a *procedure* is either a *subroutine* or a *function*. An *object* is either a *constant*, a *variable*, or a *formal parameter*… and is not related to object oriented programming. It is up to the teacher to stress these differences to avoid confusion.

5.2 Unlearning

It is a natural move to understand new features of a language by analogy with the similar features of known languages. Unfortunately, this leads to misunderstandings when features looks similar, but are in fact quite different. Some concepts (like tasking, or object orientation) are even easier to teach to people *without* corresponding experience, because they don't have to unlearn the similar looking feature of the other language. It is part of the teacher's skill to feel when a student is biased in his understanding, and to insist on the differences between the languages.

Some habits are hard to lose, like putting useless parentheses around the condition in an **if** statement! Some students (especially those who know mainly Java or other fully dynamic language) have a hard time understanding that an object can exist simply by being declared, without calling any **new** statement.

5.4 Forget Integer and Float!

The art of Ada programming is in defining appropriate (high level) types. Although the course stresses the need to define appropriate types modeling the problem, students tend to return to the types Integer (for integer values) or Float (if there is a point in the numbers!), like if these types were actually the mathematical sets **Z** and **R**, but they are not! During exercises, the teacher should really chase these and make the students think of the very nature of the entities that appear in the program. If two "things" are different in real life, they must not have the same type. For real types, show that other languages are very poor, but that Ada offers a range of possibilities. Take the example of monetary computations, where the use of floating point types is forbidden by law!

5.3 Packages and modularity

When given an exercise that involves providing a package, students tend to hurry into writing the body of the package. One of the benefits of Ada is that you can specify a package, then use the specification (therefore ensuring that the specification meets the needs of the user of the package), and only then turn to the body. The teacher must therefore look after the students to make sure that they always follow these steps:

1. Write the specification of the package, and compile it to check that it is correct Ada.
2. Write the main/test program that uses the package, and compile it to check that the specification implements the requirements.
3. Only then, generate the body skeleton (Gnatstub is instrumental for that), fill it, and try the program.

Things to look after in this process:

- Make sure that bodies are not written too early
- Make sure that once the specification is validated (i.e. the main program is compiled against the package specification), the specification is not changed (except possibly for the private part). Students tend to put declarations in the specification that belong to the body. Explain that validating the specification is like signing a contract, you are not allowed to change a contract after it has been signed!

5.4 Concurrency

Sudents have varying experience with multi-tasking. Those with some experience generally wrote programs at quite a low level, using pthreads, condition variables, interrupt handlers… The concept of logically independent and concurrent tasks is of a higher level, and students have difficulties to mentally figure that several frames of control execute at the same time. This needs a special mental process, analogous to understanding recursivity. Surprisingly enough, the concept of a rendezvous is not easily understood by those with previous experience, who tend to view the accept statement like a kind of interrupt handler that is activated when a user task needs it rather than as a statement executed according to the path of the owning task. Related to this, some have difficulties understanding that **requeue** terminates the current rendezvous or protected call.

6 Exercises

Computers are provided for the exercises, and students are allowed to bring their own laptops if they prefer. However, I always advise them to pair with a colleague for the hands-on sessions. Working in pairs triggers discussions to find the solution to a problem, avoids being blocked by a small error, and in the end, allows the participants to go farther in the exercise and make it more profitable.

Appropriate exercises are very important. Here are some qualities of a good exercise.

- An exercise must be interesting, in order for the student to be motivated in seeing the result. "Hello world" exercises are a waste of time.
- An exercise must have a first step that everybody should normally complete (to avoid frustration), and further developments for those who achieve the first step quickly, in order to exercise more advanced features.
- An exercise should be targeted to demonstrate a particular feature of the language. For example, I've seen no student really understand generics from the slides alone, but after a good exercise, they really grasp how they work.
- An exercise should include a number of traps for students to learn and think about how to solve the problem.[4]

Following these principles, the course offers six exercises:

- A first exercise to get started. It shows the general features of the language, requiring the writing of a package and the definition of some types. The exercise can be solved without using the type Integer; it is a good opportunity to watch the students and chase uses of Integer instead of user defined types!.
- An exercise on generics, showing that once a generic works in one (simple) case, it works in all cases.
- An exercise on access types (a simple linked list), showing it is possible to manipulate pointers without any core dump! It also shows that implicit dereference, which looks weird when first explained, makes a very natural notation.
- An exercise on OOP and tagged types. The course of the exercise is such that a procedure that operates on a class-wide type is written *before* the concrete classes that belong to it. This shows that a subprogram can operate on objects whose type has not yet been written, emphasizing the flexibility of the approach[5].
- An exercise on tasking with rendezvous, where many have difficulties understanding that the **accept** statement is executed sequentially, and not as some kind of call-back.
- An exercise on tasking with protected types and requeues.

In addition, the course includes a short presentation of annex E (distributed systems) with a demo where various clients print to a shared or duplicated print server. Those who had no previous experience with distributed systems find the feature very nice, but those who had to tackle with this kind of development are absolutely stunned by the ease of using and reconfiguring the system!

7 (Other) lessons learned

The *real* new thing that most students discover from Ada training is the art of modeling the problem, of thinking in problem terms and not in terms of machine representation. In the end, most of the difficulties are not with the language, but with the lack of knowledge in basic principles software engineering. An essential part of teaching Ada is not the technical details, but the message of software engineering: that programming should move to higher levels of abstraction, that specifications and implementations should be kept separated, and that defining proper types needs more thinking than writing code.

If you explain these principles, and show how Ada was designed to support them, students are easily convinced on the benefits of using Ada. A common question at the end of the course is: "with all these benefits, how comes that Ada is not more widely used?" Part of the answer is that Ada is not easy to approach in a casual manner: it has to be taught, the benefits explained; it is an industrial tool, and using an industrial tool requires training. Nobody would use an excavator without training, while anybody can pick up a shovel. Of course, it's less powerful if you have a big trench to dig…

That's why there is still a future for face-to-face training. All the technological details can be learned with internet tools, but passing the spirit of Ada requires discussion, supervised exercises, and a teacher!

The role of the teacher is especially important for exercises, because it is there, that some notions (generics, rendezvous…) are really understood, and especially through errors and the interaction with the teacher that results. Let's conclude with a proverb that everyone involved in training should keep in golden letters on his desk:

> "*I hear and I forget,*
> *I see and I remember,*
> *I do and I understand*"

[4] Some students are worried that they have difficulties in doing the exercise. I always respond: "*If I give you an exercise and you do it without any problem, it only shows that the exercise was not well chosen*".

[5] At the cost of compile-time safety, of course.