Ada, Interfaces and the Listener paradigm

J-P. Rosen*

Adalog
19-21, rue du 8 mai 1945
94110 ARCUEIL
jean-pierre.rosen@adalog.fr

Abstract. It is often claimed that interfaces, as provided by Java, are a must for Ada0Y. In this paper, we explain what interfaces are, and show equivalent constructs using Ada's "building blocks" approach. We focus then on one particular usage of interfaces, namely the listener paradigm. We detail various solutions to this problem, and show that interfaces are far from being the only, nor even the best, solution in many cases. We conclude that although interfaces are conceptually an interesting feature, their importance for Ada0Y should not be overestimated.

1 Introduction

We assume in this paper that the reader is reasonably familiar with Ada, but not necessarily with Java; therefore, we start by exposing what interfaces and other relevant features of Java are, then we explore the needs that are fullfilled by interfaces, and finally show how the same needs can be addressed in Ada.

This paper gives in places a view of inheritance which is quite different from what is generally considered as accepted. We understand that many people will not necessarilly agree with the ideas presented here; we only hope to provide some new food for thoughts.

2 Inheritance Revisited

When introducing the basic mechanisms of inheritance, it is customary to explain that there is a *parent* class, that provides data fields and methods, from which a *child* class is derived. The child class has the same data fields and methods as its parent; they are thus said to be *inherited*. In addition, if some of the inherited methods are not appropriate to the child class, they can be *redefined*; i.e. the child class provides a *different behaviour* for the *same* method. Incidentally, we notice that a method is either inherited or redefined, but can never disappear; all descendants of a class have (at least) the same methods as their ancestors.

This way of presenting inheritance is consistent with the so-called *delta-programming* paradigm: a child class is viewed as a slight modification in behaviour from an existing class. However, it is possible to give a different flavour to inheritance by presenting it as follows:

* Many thanks to R. Riehle for reviewing an early version of this paper

A parent class defines a set of properties that are shared by all its descendents. For example[1], all animals can eat. Therefore the class of all animals provides an `Eat` method. Then, each particular animal must define *how* it eats, by giving its own definition of `Eat`. That's why operations of a class are called *methods*: each class defines its own way of doing some basic operation. However, it may happen that all sub-classes of a given class share the same way of doing something; for example, all varieties of dogs use the same method for eating. Therefore, the `Dog` class can provide a *default implementation* of `Eat`, that will be shared by all subclasses, unless it is explicitly overridden.

In this second presentation, we focus on a *common interface* which is guaranteed to be provided by all the descendants of a class; inheritance of methods appears only as a secondary feature, a kind of default value that can, but needs not be provided by the ancestor. Actually, most OO programming languages provide *abstract methods* which are simply methods for which no default is provided, and therefore *must* be redefined by descendants.

Of these two aspects of inheritance, the guaranteed interface is clearly the most important feature; this is what allows *polymorphism*, the ability for a variable to designate values of a type that is not statically known, and still apply relevant operations to it.

3 Important Java Features

3.1 Interfaces

A Java interface is simply inheritance restricted to its first aspect: the definition of a set of provided methods, without any default values for them. A class that *implements* an interface promises to provide the methods defined by the interface. For example, a simple interface would look like this:

```
public interface Enumeration { // From package java.util
  public boolean hasMoreElements ();
  public Object nextElement () throws NoSuchElementException;
}
```

We can now define a method which is applicable to any class that implements the `Enumeration` interface:

```
//This method prints all elements in a data structure:
void listAll (Enumeration e) {
  while e.hasMoreElements()
    System.out.println (e.nextElement());
}
```

In a sense, an interface is nothing but an abstract class that defines no data fields, and whose methods are all abstract. But actually, it can be argued that

[1] Sorry for this overused example!

interfaces are not connected to inheritance at all, and are even the *contrary* of inheritance: it is just a contract that can be obeyed by several classes, *without* requiring any conceptual dependency between classes that implement a given interface. In short, inheritance corresponds to a *is a* relationship, while interfaces correspond to a *provides* relationship.

The major benefit of interfaces is that variables can be declared of an interface type, and can refer[2] to any object that implements the interface. Only the methods defined for this interface are available through such a variable. For example:

```
class DataStructure implements Enumeration {...}
  ...
Enumeration E;
   ...
E = new DataStructure();
```

Of course, the implementation of the methods promised by `Enumeration` have full visibility on all the internals of the class `DataStructure`.

3.2 Inner Classes

Another important feature of Java, that appeared with Java 1.1, is the notion of *inner class*. An inner class is a class that is defined within another class. The constructor for such a class is actually a method *provided* by its enclosing class; as such, an object of the inner class must be created by applying the constructor to an object of the enclosing class. The newly created object is then in some sense hard-wired to the object that served to create it. The inner class has visibility to all components of this parent object.

Here is an example of an inner class, and of the creation of an inner object:

```
class External {
  public class Internal {...}
}
  ...
External ext = new External ();
External.Internal ei= ext.new Internal ();
```

3.3 Equivalence of Interfaces and Inner Classes

Any use of interfaces can be replaced by an equivalent structure that uses only inner classes. We'll show this on an example. Imagine an interface called `Wakeable` that provides a method `everySecond`. A class implementing this interface can register itself to a scheduler, that will call its `everySecond` method every second. The outline of this is as follows:

[2] In Java, all variables (except for elementary types) are references (i.e. pointers) to objects.

```
interface WakeableInterface {
  void everySecond ();
}

final class Scheduler {
  static void Register (WakeableInterface p) { ... }
}
```

Using interfaces, a periodic class would look like this (note that the constructor registers the current object to the scheduler):

```
class Periodic implements WakeableInterface {
  public void everySecond () { ... }
  void Periodic () { Scheduler.Register (this); }
}
```

However, the following scheme, which uses only classes (and inner classes) is equivalent:

```
public abstract class WakeableClass { // class, not interface
  abstract void everySecond ();
}

final class Scheduler {
  static void Register (WakeableClass p) { ... }
}

class Periodic {
  class localWakeable extends WakeableClass { // Inner class
    public void everySecond () { ... }
  }
  localWakeable l = new localWakeable (); // local inner object
  void Periodic () { Scheduler.Register (l); }
}
```

Note that the method `everySecond` can use the name `Periodic.this` to access the fields of the enclosing object, and has the same visibility scope, and can access the same elements, as its counterpart made from an interface. From a philosophical point of view, it is slightly different; in the interface case, the object provides the functionality (`everySecond`) itself, while in the second case the functionality is delegated to a sub-object. However, both solutions are technicaly equivalent.

4 The Ada Building-Block Approach

To understand the remaining of this discussion, it is very important to remember the basic *building block* approach of Ada. In Ada, there is no such thing as classes (in the usual sense), friends, or even interfaces defined in the language. All these usual constructs are built from the combination of basic building blocks.

The Ada approach can be understood by comparing Lego® blocks to Playmobil® pieces. The great versatility of Lego blocks comes from the fact that the same piece can be used to build very different elements; for example, the same round plate is used to build an antenna in the Space Lego, a shield in the Middle-Age Lego, or an umbrella in the Lego village. On the other hand, a piece from a Playmobil set is very specialized and cannot be used in a different context than its initial box.

Ada typically works like Lego blocks; a class is neither a package nor a tagged type. It is a *design pattern*, where a package declares only a tagged type and its associated operations. Friends (in the C++ sense) are made by declaring several related tagged types in the same package, etc. Although this approach requires a bit more effort from the programmer, it allows the building of constructs not available in other languages without new syntax.

When told about the Ada way of doing things, people often react by saying that it is only a workaround for compensating the lack of such-and-such feature in Ada; it is not. It is a different approach to building various mechanisms, that allows almost any paradigm to be used without requiring additions to the language.

5 A Design Pattern for Interfaces in Ada

In this section, we show that it is possible to define a design pattern in Ada that matches closely the notion of Java interfaces - or more precisely, the inner classes that are equivalent to a Java interface.

Since we try to mimic the way Java works, we use limited types that we manipulate through pointers. Of course, other design patterns are possible that would not require these pointers. We do not address at this point the issue of interfacing with Java, but only the one of providing an equivalent feature.

Given that:

- an interface is equivalent to an abstract class with abstract methods;
- the usual way of adding a "facet" to an Ada tagged type is through instantiation of a generic;
- generic formal parameters can already express the notion that a data type must provide a set of operations;

we can express the interface from the previous example as described in the next listing[3]. Note that the body is relatively simple, but requires some care in order not to violate accessibility rules!

The type `Wakeable_Interface.Instance` expresses the properties of the interface. An instantiation of the generic package `Implements_Wakeable` on a tagged type that provides the necessary `Every_Second` procedure will provide a

[3] We use here the convention advocated in [5], where the package is given the name of the class, the main (tagged) type always has the name "Instance", and the associated pointer type the name "Handle"

```
package Wakeable_Interface is
  type Instance is abstract tagged limited null record;
  type Handle is access all Instance'Class;
  procedure Every_Second (This : access Instance) is abstract;

  --A type with an Every_Second operation can implement
  --the interface by instantiating the following generic:
  generic
    type Ancestor is tagged limited private;
    with procedure Every_Second (Item : access Ancestor) is <>;
  package Implements_Wakeable is
    type Instance is new Ancestor with private;
    type Handle is access all Instance'Class;
    function Wakeable_Of (Item : access Instance)
                          return Wakeable_Interface.Handle;
    function Full_Object_Of (Item : Wakeable_Interface.Handle)
                            return Handle;
  private
    type Inner_Type (Enclosing : access Instance) is
      new Wakeable_Interface.Instance with null record;
    procedure Every_Second  (This : access Inner_Type);

    type Instance is new Ancestor with
      record
        Inner_Instance : aliased Inner_Type (Instance'Access);
      end record;
  end Implements_Wakeable;
end Wakeable_Interface;

package body Wakeable_Interface is
  package body Implements_Wakeable is
    procedure Every_Second (This : access Inner_Type) is
    begin
      Every_Second (Ancestor(This.Enclosing.all)'Access);
    end Every_Second;

    function Wakeable_Of (Item : access Instance)
                          return Wakeable_Interface.Handle is
    begin
      return Item.Inner_Instance'Access;
    end Wakeable_Of;

    function Full_Object_Of (Item : Wakeable_Interface.Handle)
                            return Handle is
    begin
      return Inner_Type (Item.all).Enclosing.all'Access;
    end Full_Object_Of;
  end Implements_Wakeable;
end Wakeable_Interface;
```

new tagged type which is the original one with the addition of the interface. The function `Wakeable_Of` returns a handle associated to an object, which is usable to manipulate it as a `Wakeable` object; conversely, the function `Full_Object_Of` returns a handle to the full object, given a handle to the interface.

Here is how we would declare a class `Data1`, then use the generic to provide the class that implements this interface:

```
package Data1 is
  type Instance is tagged limited private;
  type Handle is access all Instance'Class;
  procedure Init (Item : access Instance; Value : Integer);
  procedure Processing (Item : access Instance);
  procedure Every_Second (Item : access Instance);

private
  type Instance is tagged limited
    record
      Value : Integer;
    end record;
end Data1;

with Wakeable_Interface; use Wakeable_Interface;
package Data1.Wakeable is
  new Implements_Wakeable (Data1.Instance);
```

Note that since the type `Data1` features an `Every_Second` procedure, it is automatically selected by the instantiation, but that, unlike Java interfaces, the operation provided for the implementation of `Every_Second` needs not have the same name; for example, the instantiation could have explicitly mentioned `Processing`. This can be handy in some situations, like having the same operation provided for the implementation of two different interfaces.

Finally, here is an example of a program using this data type. Note that the value is created as a `Data1`, and then manipulated as a `Wakeable`; we also demonstrate that we can go back from a `Wakeable` to a `Data1` :

```
with Data1.Wakeable, Wakeable_Interface;
procedure Example is
  use Data1, Data1.Wakeable, Wakeable_Interface;
  Handle_1 : Data1.Wakeable.Handle;
  Handle_2 : Wakeable_Interface.Handle;
begin
  Handle_1 := new Data1.Wakeable.Instance;
  Init (Handle_1, 5);

  Handle_2 := Wakeable_Of (Handle_1);
  Every_Second (Handle_2);

  Processing (Full_Object_Of (Handle_2));
end Example;
```

With this design pattern, the *declaration* of the interface is more verbose (complicated?) than its Java counterpart; declaring that a type implements the interface just requires a simple instantiation; and *using* the data type, either normally or as an interface, is extremely simple. This is the usual trade-off with the building-block approach: the extra complexity required by not having the construct readily available is charged on the *designer* of the abstraction, but there is almost no cost to the *user*.

6 What Are Interfaces Used For in Java?

The previous section showed a design pattern that allows the same programming style in Ada as in Java. However this does not imply that all uses of interfaces in Java *must* use this pattern in Ada. In Java, interfaces are used for different purposes:

Flags. Java features some empty interfaces, i.e. interfaces that declare no methods. Classes can implement these interfaces just to flag that they allow certain operations. For example, every object features a `clone` method to duplicate itself, but the method will throw an exception unless the class specifically allowed it by declaring that it implements the interface `Cloneable`. This is clearly more a programming trick than a proper usage of interfaces: what does it mean for a class to promise that it implements... nothing?

Simple interfaces. These are truly interfaces, i.e. a promise that the class features some properties. Interfaces like `Serializable` (the class can be written to a file) or `Comparable` (objects can be compared) fall into that category. It is easy to understand that, for example, a file object can handle any object that implements `Serializable`. The above design pattern can be used in that case, although often the same effect can be achieved using simple generics.

Restricted multiple inheritance. In Java, a class can inherit from only one class, but can implement any number of interfaces. *If* we give the meaning to an interface that every class that implements it is considered as belonging to a class of objects, then this usage can be considered as an implementation of multiple inheritance. The benefits over full MI is that since all methods of an interface *must* be redefined, there can't be two default implementations for the same method at the same time, therefore avoiding the issue of repeated inheritance. Once again, this kind of usage can be achieved in Ada by using the previous programming pattern.

Listeners. This is a paradigm where there are occurrences of some events in the system, and one or several objects need to take some action in response to the event. They are said to *listen* to the event. The construct that will trigger these actions is called the *notifier*: it can be a programming object, but also a hardware interrupt for example. Typical listeners are found in the management of interrupts, mouse events, clock events, dialog buttons... The notion of *event* is not necessarily connected to an external event; it can be a simple state change in an object[3], as in OO data bases.

7 Various Implementations of the Listener Paradigm

Note first that from a conceptual point of view, listeners are *active* objects: they can have actions that are invoked asynchronously, and care must be taken about concurrent access to the methods that they offer.

Two basic mechanisms are associated to the listener paradigm: there must be a *registration* process (with the associated deregistration process), to make the notifier aware of all possible listeners, and a *delivery* process to trigger appropriate actions in each listener when the event occurs. Of course, the notifier should have no *a priori* knowledge of who the listeners are. Typically, it will maintain a linked list of listeners, and call the delivery method on each of them each time the event occurs.

7.1 Call-Backs

Historically, the first examples of the listener paradigm were for handling interrupts, or for dealing with events from user interfaces, like connecting some processing to a mouse click on a button. Most of the time, the programming language used (assembly, C) had no tasking feature. Therefore, the first mechanism was *call-backs*: a procedure is associated to the event. Registration consists in passing a pointer to the procedure to the notifier, and the notifier issues an indirect call to the procedure whenever the event happens. This mechanism is still the most common one for hardware interrupts.

Since Ada has pointers to subprograms, this solution can be used directly.

7.2 Extended Call-Backs: Interfaces

The previous solution falls short when there are several related events, with different processings that have to be registered at the same time. For example, one might want to register different call-backs depending on whether a button is clicked on with the left, middle or right button.

It is of course possible to register multiple call-backs; but a set of subprograms connected to a given entity is exactly the definition of an interface; it is therefore more convenient to define an interface, and to register the interface as a whole. For example, our `Wakeable` interface can be defined as follows, if we want various operations to be performed every second, minute, or hour:

```
interface WakeableInterface {
  void everySecond ();
  void everyMinute ();
  void everyHour   ();
}
```

It is often argued that interfaces are much better than call-backs, even when the interface features only one method. This is because *in C*, registering a call-back is done by providing the address of a function, without any kind of checking about the conformance of parameters. Interfaces ensure that the profile of the

provided functions match the required profile. But of course, in Ada, pointers to subprograms must be consistent with the expected usage, so the added value of interfaces is much smaller. Therefore, it is much more acceptable in Ada to register multiple call-backs, and interfaces are not necessary in this case.

7.3 Generics

It is often the case that there is only one listener to a given event. This idiom can be very simply implemented by making the notifier generic, with the notified procedure as a generic parameter. For example, the scheduler that handles `Every_Second` events could be declared as:

```
generic
  with procedure Every_Second;
package Scheduler is
  ...
end Scheduler;
```

Note that we could as well provide several procedures as formal parameters, if the notifier was serving several events.

7.4 Tasking

In all the previous idioms, the processing of events was done by (aysnchronously) calling a procedure. However, as noted above, this is really an active behaviour, and it seems natural to model this with tasks. Actually, if the call-back mechanism (and variants) is in so wide-spread use, it is because at the time where GUI appeared that made an intensive use of listeners, most languages had no concurrency.[4]

Polling A first possibility is for the notifier to provide a function that tells whether a new event has arrived, or a function that returns the next event, with a special `null_event` if none has occurred. The listener will actively sample the notifier for events (this is how the main loop of GUIs behaves generally) :

```
task body Listener is
  ...
begin
  loop
    Event := Notifier.Get_Next_Event;
    case Event is
      ...
    when Null_Event =>
      delay 0.1;
    end case;
  end loop;
end Listener;
```

[4] with the notable exception of Ada, of course!

This idiom is not appropriate when there are several listeners, since the notifier would have to deliver the last event that occurred to any task that requests it; however, it would be very difficult to avoid delivering the same event twice to the same listener. In practice, the notifier will rather reset the current event as soon as it has been delivered once. On the other hand, it is easy for the listener to poll various notifiers until an event is delivered, allowing it to process multiple kinds of events.

Entries A notifier can be viewed as an event server, in which case it is natural to represent it as a task or protected object. Listeners will get events in the same way as with the previous idiom; however, `Get_Event` will typically be an entry rather than a subprogram. This implies two main differences:

- The listener will block until an event is available. This means that no active loop is necessary.
- When an event occurs, the notifier can deliver the event to all listeners that are currently queued. With a protected object, it is quite easy to ensure that no race conditions can occur if one of the listeners returns to the point where it requests an event before all listeners have been notified. With tasks, this can be ensured using `requeue` or entry families, although not as easily.[5]

8 Comparison of the Various Implementations of the Listener Paradigm

Idioms based on procedure calls are *sequential* according to Booch's taxonomy[2], i.e. there must be an external synchronization mechanism to ensure that the state of the object is not modified by another procedure while the object is being notified of an event, while idioms based on tasks are naturally *concurrent*.

Idioms based on tasking have the nice feature that there is no need for a registration mechanism: the task simply requests an event when it is in a state that allows it. The generic idiom is a purely static solution: there is no need for registration either, and the region where the listener is active is entirely determined at compile time by scoping rules.

Whether it is useful to allow more than one listener really depends on the application; although this is a must in some contexts, it can be nice in other cases to be able to ensure that an event can be processed by only one listener.

Table 1 summarizes the properties of each solution. As usual, there is no single "best" solution to the listener paradigm. Which idiom to use should be determined by the application needs.

[5] Sorry, the algorithm is a bit long for this paper; please contact the author if interested.

Table 1. comparison of the various idioms for the listener paradigm

Idiom	Active?	Registration	Multiple listeners	Multiple events
Call-back	No	Explicit	Yes	No
Interface	No	Explicit	Yes	Yes
Generic	No	Static	No	Yes
Task, Polling	Yes, not blocking	Automatic	No	Yes
Task, Entry	Yes, blocking	Automatic	Yes	No

9 Proposal for Ada0Y

There is a proposal for an amendment (AI-00251)[1] that would allow the equivalent of interfaces in Ada. Basically, a tagged type could be derived from several types, all of which, except the first one, would be required to be abstract with only abstract operations. The canonical example from the AI is:

```
type Stack is abstract; -- An abstract interface type
  -- ... Operations on Stack

type Queue is abstract; -- Another interface.
  -- ... Operations on Queue
        . . .
type Deque is abstract new Queue and Stack;
  -- An interface which inherits from both
  -- the Queue and Stack interfaces.
  -- ... Operations on Deque
        . . .
-- An extension that implements an interface
type My_Deque is new Blob and Deque with
  record
    -- Conventional type implementing an interface.
    . . .
  end record;
```

The proposed implementation model is very close to what is proposed in section 5: the tagged type would contain pointers to the various dispatching tables.

Note that although this would be convenient, especially for interfacing to Java, current compilers are able to recognize constructs that are "close enough" to map Java interfaces. For example, the Aonix® compiler help file explains that:

> An Ada tagged limited type "claims" that it implements an interface by including in its list of components an aliased component of the interface type.

10 Conclusion

We have argued already [4] that the fact that other languages rely only on inheritance as the implementation mechanism for a number of concepts does not mean that the same should apply to Ada, given its much richer set of features. We have shown here that the fact that Java uses interfaces in various ways, and especially for all cases of listeners, should not be used to conclude that it is always the most appropriate paradigm in Ada.

Java interfaces are a useful concept, and the idea of "promised interface" is certainly an attractive feature. Moreover, since there are Ada compilers that target the JVM, it is important to define how to access Java interfaces from Ada. However, although there is nothing in Ada that maps *directly* to Java interfaces, we have shown in this paper that existing Ada constructs provide solutions to the problems that Java interfaces address. Therefore, we claim that there is no *technical requirement* for adding interfaces to Ada. It would only be syntactic sugar, since equivalent constructs can be built using current features.

But it must be remembered that sometimes, sugar (i.e. new building blocks) "helps the medicine go down"... Finding ready-to-use features in a language makes it easier to use, but having too many of them increases complexity. The real issue is therefore where to draw the line between building blocks and specialized pieces, and this is certainly not easy. Lego blocks allow much more creativity than Playmobil pieces[6], but even Lego has provided more and more sophisticated blocks over time...

In conclusion, we claim that the issue of adding interfaces in Ada0Y needs investigation, at least for interfacing with Java, but that the technical need is less important than often advocated.

References

1. http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00251.TXT
2. Booch, G.: Software Components with Ada, Benjamin Cummings Company, Menlo Park, 1987
3. Heaney, M.: "Implementing Design Patterns in Ada 95", Tutorial, SIGAda 2000 conference.
4. Rosen, J-P.: "What Orientation Should Ada Objects Take?", Communications of the ACM, Volume 35 #11, ACM, New-York.
5. Rosen, J-P.: "A naming Convention for Classes in Ada 9X", Ada Letters, Volume XV #2, March/April 1995.

[6] Admitedly, this is a personal opinion!