On Exceptions as First-Class Objects in Ada 95

Thomas Wolf
Paranor AG, Switzerland

e–mail: twolf@acm.org

***Abstract**. This short position paper argues that it might be beneficial to try to bring the exception model of Ada 95 more in–line with the object–oriented model of programming. In particular, it is felt that exceptions — being such an important concept for the development of fault–tolerant software — have deserved a promotion to first–class objects of the language. It is proposed to resolve the somewhat awkward dichotomy of exceptions and exception occurrences as it exists now in Ada 95 by treating exceptions as (tagged) types, and exception occurrences as their instances.*

Keywords. Object–Oriented Exception Handling, Software Fault Tolerance

1 Introduction

The exception model of Ada 95 [ARM95] is still basically the same as it was in the original Ada 83 programming language; although some improvements did occur: the introduction of the concept of exception occurrences, and some other, minor improvements concerning exception names and messages.

Exceptions are a very general and powerful tool to build reliable applications. They provide a way to separate normal flow of control cleanly from exceptional error handling. This makes a proper separation of concerns possible, and helps in properly structuring and designing software components.

Ada 95 is an object–oriented programming language, and it has a long success history in the development of reliable software. The object–oriented features had been integrated rather smoothly into the already strong typing system of Ada 83. It is all the more surprising that exceptions apparently did not get too much attention in this integration process: they follow a somewhat awkward model and are completely set apart from the powerful and clean semantics of other types in the programming language. In fact, the programming language doesn't handle exceptions as types at all.

This may once have been a sensible design decision under the aspect of not creating too big a gap for the transition from Ada 83 to the then new language Ada 95. However, I believe the time has come to reconsider these choices made nearly 10 years ago, especially in view of the experiences made since then in other successful programming languages that show a much better integration of object orientation and exceptions, most notably Java.

In the remainder of this paper, I'll try to make a case for better integrating exception into the object–oriented paradigm in Ada 95. Section 2 discusses some problems with the current model of exceptions. Section 3 argues for an improved exception model that would solve these problems, and section 4 presents a brief summary.

2 Exceptions in Ada 95

Exceptions in Ada 95 are seen as some kind of "special objects": even their declaration looks more like that of an object than a type declaration:

```
Some_Exception : exception;
```

Such exceptions can be raised (or thrown, as other languages call it) through a `raise` statement:

```
raise Some_Exception;
```

or through a procedure that passes a string message along with the exception:

```
Ada.Exceptions.Raise_Exception
   (Some_Exception'Identity, "Text");
```

Any `begin–end` block can have exception handlers:

```
begin
  ...
exception
  when E : Some_Exception =>
    Ada.Text_IO.Put_Line
      (Ada.Exceptions.Exception_Message(E));
  when others =>
    --  Whatever...
end;
```

The handler for `others` catches any exception not caught by one of the other exception handlers in this block. The 'E' in the first handler declares an `Exception_Occurrence` that will hold the caught exception instance and that is needed if the handler needs to get at the exception message associated with this instance.

The exception message (and also the exception name) are things added in Ada 95 to rectify deficiencies found in Ada 83. But still, limiting the additional information that may be passed along with an exception to a simple character string is too restrictive. There are many cases where one would like to add even more (and typed and structured!) data to an exception, and for such uses, the single string, whose length may even be truncated to 200 characters if the exception is re–raised [ARM95, §11.4.1(18)], is simply not sufficient. I'll show one example in section 2.1 below.

Also, Ada 95 has no means to instantiate generics with exceptions: there are no formal generic exceptions. (As there are no generic formal task types, which in some rare cases also might be convenient.) Generic units thus have to raise exceptions declared elsewhere, or to declare their own exceptions. In the latter case, exceptions resulting from two different instantiations also are different [ARM95, §11.1(3)], and in the former case, the generic unit (not the place where it is instantiated!) needs to have visibility of the exception declaration. There is a work–around (see section 2.1), but it is somewhat low–level.

Finally, exceptions in Ada 95 cannot be organized in hierarchies. This, too, can be quite useful in many cases. Exception hierarchies go together with the ability to handle either individual exceptions or a whole (sub–)tree of such a hierarchy. This helps insulate users of a software component against changes in the exception raising behavior of that component, which is all the more important in Ada since exceptions are not part of the interface of operations. Consider a component that is extended or otherwise changed and then either raises a new exception it didn't raise before, or some operations now raise different (but already declared) exceptions. Without exception hierarchies, it would then be necessary to make according changes in the exception handlers of all other software components using the changed component. With hierarchies, this may not always be necessary. (It also may still be necessary in some cases, of course.) If a software component structures its exceptions, and in particular related ones, in exception hierarchies, users of the software component have the choice of handling the whole hierarchy instead of individual exceptions. Such hierarchy–wide handlers would not necessarily have to be adapted in this case.

2.1 Work–Arounds

For some of the above limitations, there exist work–arounds. Instead of working with generic formal exception parameters, which don't exist in Ada 95, it is possible to use a generic formal object of type `Ada.Exceptions.Exception_Id`, which can then be used inside the generic unit to raise the designated exception not by a `raise` statement but through procedure `Ada.Exceptions.Raise_Exception`. This works, but it strangely departs from the usual high–level view taken in Ada. Working with `Exception_Ids` in this way is reminiscent of working directly with the tags of tagged types.

The usual way to add application–defined arbitrary data to an exception (occurrence) relies on a mis–use of the standard exception message. This method is exemplified by the CORBA IDL mapping to Ada 95.

In CORBA IDL, exceptions can carry arbitrary application–defined attributes. The mapping of IDL to Ada 95 must therefore somehow provide a way to add additional information to exceptions in Ada 95. This is done by defining separate tagged types to hold the exception attributes. When a CORBA exception is raised in Ada, the application must not use the `raise` statement directly, but first create an object of the appropriate type in `IDL_Exception_Members'Class` and then call a special primitive operation of this type. This operation first copies the data into some dynamically allocated (or otherwise global) object and then puts some kind of reference to this object in stringified form into the standard exception message. Then the normal Ada exception corresponding to the CORBA exception is raised with that message.

Where a CORBA exception is caught in an exception handler, the Ada application can retrieve the associated data by calling a special operation that looks at the exception message, extracts the reference from it and then returns the object associated with this exception occurrence.

While this works, it makes for a rather heavy interface that is not particularly convenient to use. Worse yet, this mechanism relies on the application following an implicit protocol for treating exceptions, and this protocol cannot be enforced. If an application directly raises an Ada exception that corresponds to a CORBA exception without adding the CORBA data, or if an application changes the exception message of such an exception, all bets are off and it is rather unlikely that the scheme will continue to work as intended. Such unsafety is in stark contrast to the rest of the Ada 95 language!

There is no direct work–around known to me for the lack of exception hierarchies in Ada 95. (Except the catch–all "`when others`", but that should really remain reserved for rare special cases. Generous use of "`when others`" is in general not a sign of a well thought–through error handling strategy. Also, it indiscriminately catches any exception and is thus not well suited for dealing with exception hierarchies.) One might attempt to use the above work–around for adding application–defined attributes to an exception occurrence to also encode somehow an attribute that further distinguishes between occurrences of a particular exception, but any such attempts will be partial and unsafe solutions only, and rather heavy and inflexible to use.

2.2 Some History

Interestingly enough, the above problems were all well known at the time the design of Ada 9X began. A very early document [Ada90a] on requests for revision identified

- no support for grouping exceptions,
- no support for application–defined exception (information) data items, and
- no generic formal exception parameters

as shortcomings of Ada 83 that were perceived as worthwhile to try to improve in the Ada 9X project.

Early drafts of the new standard–to–be did indeed cater for these concerns: the Draft Mapping Document from February 1991 [Ada91a] did contain derivable exception types in the form of something called "tagged elementary types" as well as generic formal exception parameters. That approach improved at least two of the three problem areas mentioned in [Ada90a]. However, these revision requests from [Ada90a] didn't make it into the official Ada 9X requirement document [Ada90b], and consequently derivable exception types (together with the tagged elementary types themselves) had dropped out of consideration by August 1991 already — Mapping Document V3.1 [Ada91b] doesn't mention either concept anymore. History is a bit blurred as to the reasons for this decision; unfortunately, the extensive archives on the language design process do not reveal this particular piece of information. In the Rationale [Ada91c] for [Ada91b], it is only mentioned that the revision requests for these topics were not well-founded enough (grouping exceptions) or were "not considered to provide enough user benefit to justify disturbing the language" (additional information to be passed around with an exception) and hence did not warrant the increase in complexity of implementing such a scheme. It also appears that this decision was never revisited later on, when the OO model of Ada 9X settled down.

3 First–Class Exceptions

Most of the aforementioned shortcomings of the current exception model in Ada 95 could be solved by a better integration of exceptions into the general type system of the programming language. In Java (and C++), exceptions are classes and thus fit much better into the object–oriented paradigm than they do in Ada 95. Even these implementations are not original: [Dony90] presented exactly such a model in the context of Smalltalk. CORBA IDL also has exceptions that can carry arbitrary application–defined attributes, although there are no exception hierarchies in IDL. (There are just two broad classes of exceptions in IDL: "checked" and "unchecked" exceptions, like in Java — IDL also, like Java, requires operations to declare the exceptions they may raise in the interface. Practicality considerations then have led to the introduction of "unchecked" exceptions that do not need to be declared in the interfaces. This undermines to some extent the whole approach, especially if lazy developers mis–use these unchecked exceptions instead of using their own, checked ones — see e.g. [RM99] for an example.)

All this strongly suggests that exceptions become real Ada types. Exception occurrences then are simply instances of these types. To allow applications to add their own attributes, exceptions should become tagged types.

This has some interesting implications. For instance, today's standard package `Ada.Exceptions`, and its special types `Exception_Id` and `Exception_Occurrence`, would become superfluous. The former is equivalent to the tag, and the latter would become a class–wide type. This might be defined along these lines:

```
package Standard is

   type Root_Exception_Type is
      abstract tagged
      --  Maybe even limited, see text.
      record
         --  Implementation defined
      end record;

   function Exception_Name
         (E : Root_Exception_Type)
      return String;
   function Exception_Message
         (E : Root_Exception_Type)
      return String;
   function Exception_Information
         (E : Root_Exception_Type)
      return String;
   -- And maybe others (e.g. Set_Message,
   -- or even Raise (see text)).

   type Constraint_Error is
      new Root_Exception_Type with
         null record;

   type Program_Error is
      new Root_Exception_Type with
         null record;

   -- And ditto for the other predefined
   -- standard exceptions

   subtype Exception_Occurrence is
      Root_Exception_Type'Class;

end Standard;
```

The Root_Exception_Type could also be made limited: in that case, additional support operations for copying exception instances (= occurrences) might be needed. (See also section 3.2 on the limitedness of Root_Exception_Type.)

Raising an exception then requires an instance of such an exception type, e.g., one would have to write

```
raise Constraint_Error'
        (Root_Exception_Type with null);
```

(if Root_Exception_Type is non–limited) or maybe even

```
raise new Constraint_Error;
```

In fact, the raise statement logically can become a primitive operation of Root_Exception_Type, and so can procedure Raise_Exception from package Ada.Exceptions.

With this approach, a straight–forward extension to exception handlers for handling class–wide exception occurrences presents itself, as illustrated by the code snippet below.

```
begin
   ...
exception
   when E : Some_Exception =>
      -- Handle an instance of type
      -- 'Some_Exception'.
   when E : Some_Exception'Class =>
      -- Handle any exception instance
      -- of type 'Some_Exception' or any
      -- type derived from it.
   when E : others =>
      -- "others" is equivalent to
      -- "Root_Exception_Type'Class".
end;
```

With class–wide exception handlers, the handling of a whole exception hierarchy can be subsumed in one handler. A rule would be needed stating that specific handlers take precedence over class–wide ones within the same block, otherwise a Some_Exception might always be caught by the second handler in the above example.

Such an approach addresses all three deficiencies presented in section 2. Exception hierarchies can be constructed, and class–wide handlers can be written. Applications can extend exceptions to carry application–specific attributes by type derivation, as for normal tagged types. And generic formal exception parameters also are possible: one would just use a generic formal parameter of an exception type.

While the benefits of such a scheme from a user perspective seem clear, the implications for implementing this would need to be evaluated carefully. However, I believe the "added value" warrants serious efforts to try to move Ada's exception facilities in such a direction.

Maybe a caveat is in order here: of course, such an integrated, object–oriented exception design is no panacea. Exception hierarchies don't solve all difficulties, but they can be a big help in designing error signalling in an application properly. Programming language features never are a substitute for proper software design! The error signalling and handling behavior of a software system must be planned just as carefully (and just as early!) as the software's normal mode of operation. Otherwise, error handling may become erratic whatever mechanisms are used. For one viewpoint on this issue, see [RM99], and also [Litke90].

3.1 Backwards Compatibility

Backwards compatibility with the current exception model of Ada 95 could be achieved on a syntactical level by defining the existing syntax of exceptions in terms of the new exception types. For instance,

```
subtype Exception_Occurrence is
   Root_Exception_Type'Class;
```

Type Exception_Id essentially is the tag of the type of an exception instance. The current syntax notation

```
Some_Error : exception;
```

can be defined to mean

```
type Some_Error is
   new Root_Exception_Type with
      null record;
```

and

```
raise Some_Error;
```

can be seen as short–hand for

```
raise Some_Error'
        (Root_Exception_Type with null);
```

These are just sketches, but they indicate that integrating exceptions into the type system could most probably be done in a way that did not require changes in existing source code.

3.2 A Problem: Local Exceptions

One problem with this direct approach exists, though. The current semantics of tagged types do not allow derivations at an accessibility level statically deeper than that of the parent type [ARM95, §3.9.1(3, 4)]. This means that one cannot declare local exception types with the approach sketched here; however, the language currently does allow local exceptions. (And this is a useful feature in some cases, too.) The following example shall illustrate the problem:

```
generic
   X : in Natural; --  Just as an example
package Something_Or_Other is

   An_Error : exception;

   procedure Do_Something (...);
   --  May raise ‘An_Error’

end Something_Or_Other;

procedure Foo (Param : in Natural; ...) is
   package Bar is
      new Something_Or_Other (Param);
begin
   Bar.Do_Something (...);
exception
   when Bar.An_Error =>
      -- Do whatever is appropriate
end Foo;
```

The above is correct Ada 95. If exceptions became (limited) tagged types, this code would be illegal by the current rules of the programming language: An_Error would be a derivation of Root_Exception_Type, but Root_Exception_Type is declared at library level, and thus all derivations from it also must be at library level. The local instantiation of the generic package is therefore not allowed. Another aspect of the same problem can be seen in the next example:

```
procedure Do_Something (...)
is
   Abort_Backtracking : exception;

   procedure Backtracking_Algorithm (...)
   is
   begin
      if ... then
         -- Some condition detecting that
         -- further recursion is useless.
         raise Abort_Backtracking;
      end if;
      ...
      Backtracking_Algorithm (...);
      ...
   end Backtracking_Algorithm;

begin -- Do_Something
   Backtracking_Algorithm (...);
exception
   when Abort_Backtracking =>
      -- Backtracking failed, do whatever
      -- is appropriate.
end Do_Something;
```

This structure actually occurs in real code (a wildcard string pattern matcher; the “zero or more arbitrary characters” wildcard ‘*’ causes backtracking). Assume now that it was allowed to declare local derivations of tagged types, and the local exception declaration thus was fine. If we override now one of the primitive operations of Abort_Backtracking and make it access data local to Do_Something (which is possible according to the visibility rules), a serious problem occurs if that exception ever is propagated out of Do_Something, because then that local data no longer exists.

Finally, there is an awkward interaction between locally derived tagged types and class–wide types. Consider the following:

```
function Get_Exception
   return Root_Exception_Type'Class
is
   type Local_Exception is
      new Root_Exception_Type with
         null record;
   Result : Local_Exception;
begin
   return Result;
end Get_Exception;

Ex : Root_Exception_Type'Class :=
   Get_Exception;
```

This clearly doesn’t make much sense. (It gets particularly troublesome if Local_Exception extends the record or overrides primitive operations.) The problems with local exceptions as tagged types are thus three–fold:

- Declaring local derivations of tagged types is illegal under the current rules of the language. I have no canned solution for this: it remains a study topic. A change in the static semantics of the programming language would be clearly needed. Perhaps an exception from [ARM95, §3.9.1] could be made for Root_Exception_Type, or a general way around this rule can be found — there is already an AI on downward closures for subprogram access types and limited tagged types [Duff2000], which just might be a step towards resolving this issue.
- If a local exception ever got propagated out of its scope, its primitive operations might try to access no longer existing data. Perhaps this problem could be avoided by requiring that any local exception must always be caught in the enclosing scope and must never be propagated out of it: if that ever happened, it would be a programming error in virtually all cases anyway, and furthermore the exception would become anonymous and thus be of little use.
- One must avoid that an object of a local exception type can outlive its type as in the last example above. One solution is to make Root_Exception_Type limited, but that sacrifices the convenient aggregate notation for raising exceptions (aggregates cannot be limited). Yet this is probably the simplest remedy. Another idea might be to extend the accessibility

rules currently defined for access types to class–wide types, too. In this case, the assignment implicit in the return statement in the last example above would be illegal because `Local_Exception` is declared at an accessibility level statically deeper than `Root_Exception_Type`.

3.3 An Alternative Approach

Another approach that might also be worth considering has been taken in Modula–3 [Modula89], where exceptions may be parametrized by a type. This has also been proposed in [Weber94] for the Ada programming language.

If one allows exceptions to be parametrized by (accesses to) tagged and class–wide types, this approach offers enough support for adding application–defined attributes to exceptions. However, it doesn't explicitly address grouping of exceptions or building exception hierarchies, and it doesn't address generic formal exception parameters at all. Also, parametrized exceptions might be a bigger syntactic change than exceptions as tagged types.

4 Summary

This position paper has discussed some particular shortcomings of exceptions in Ada 95. As a position paper, it does not present a conclusive analysis of the topic, but rather intends to open up the discussion as to whether — and if so, how — to improve the exception model of Ada 95, and presents one possible approach. It is clear that the model outlined above is not a minor change to the language, and has many more implications than mentioned. For instance, how to pass exceptions that are tagged types across partitions in a distributed Ada program would have to be worked out carefully, and interoperability regarding exceptions between Ada and other languages (in particular Java and C++) should be studied. Also, the interactions with controlled types — can exceptions have controlled components, and if so, when are these finalized? — must be worked out, and finally the problems with local exception types need to be solved.

References

[Ada90a] Ada 9X Requirements Team: *Ada 9X Revision Issues;* April 1990. Available at URL `http://www.adaic.org/pol-hist/history/9x-history/reports/issues1-Apr90.9x.zip.gz`. [Feb. 6, 2001].

[Ada90b] NN: *Ada 9X Requirements;* Office of the Under Secretary of Defense for Acquisition, Washington D.C.; Dec. 1990. At URL `http://www.adaic.org/pol-hist/history/9x-history/requirements/req-Dec90.txt.gz`. [Feb 6, 2001].

[Ada91a] Ada 9X Mapping/Revision Team: Ada 9X Draft Mapping Document, 21 February 1991. URL `http://www.adaic.org/pol-hist/history/9x-history/reports/map-Feb91.9x.zip.gz`. [Feb 6, 2001].

[Ada91b] Ada 9X Mapping/Revision Team: *Ada 9X Mapping, Vol. II: Mapping Specification, V3.1*, 23 August 1991. URL `http://www.adaic.org/pol-hist/history/9x-history/reports/map-spec-Aug91.9x.zip.gz`. [Feb 6, 2001].

[Ada91c] Ada 9X Mapping/Revision Team: *Ada 9X Mapping, Vol. I: Mapping Rationale, V3.1*, 23 August 1991. At the URL `http://www.adaic.org/pol-hist/history/9x-history/reports/map-rat-Aug91.9x.zip.gz`. [Feb 6, 2001].

[ARM95] Taft, S. T; Duff, R. A.: *"Ada 95 Reference Manual — International Standard ISO/IEC 8652:1995(E)"*, published as *LNCS* ***1246***; Springer Verlag 1995.

[Dony90] Dony, Christophe: "Exception Handling and Object–Oriented Programming: Towards a Synthesis"; in *Proceedings of ECOOP–OOPSLA '90; SIGPLAN Notices* ***25(10)***, pp. 322 – 330; October 1990.

[Duff2000] Duff, R. A.: "Downward Closures for Access–to–Subprogram Types"; *AI–00254* of the Ada Rapporteur Group; November 2000. URL `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00254.TXT`. [Mar 26, 2001]

[Litke90] Litke, J.D.: "A Systematic Approach for Implementing Fault–Tolerant Software Designs in Ada"; *Proceedings of TRI–Ada '90*, pp. 403 – 408; ACM; December 1990.

[Modula89] Cardelli, L.; Donahue, J.; Glassman, L.; Jordan, M.; Kalsow, B.; and Nelson, G.: "Modula–3 Report (revised)"; *Report #52*, DEC Systems Research Center; Nov. 1989. URL `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-052.html` [Mar 26, 2001].

[RM99] Robillard, M. P.; Murphy, G. C.: "Regaining Control of Exception Handling"; *Technical Report TR–99–14*, Dept. of Computer Science, University of British Columbia, Vancouver BC, Canada; December 1999.

[Weber94] Weber, M.: *Proposals for Enhancement of the Ada Programming Language: A Software Engineering Perspective;* PhD Thesis #1227, Dept. of Computer Science, Swiss Federal Institute of Technology in Lausanne, Switzerland; 1994.