Ada 2005 syntax-sensitive eclipse plugin Design Justification

Document : cross-references draft

written by Didier Garcin

February 2015, 11th

Chapter 1

Generalities

1.1 Accessibility rules

Accessibility rules deal with authorized reference between two ***Basic Declarations*** of differents ***Library Unit Declaration*** and between***Library Unit Declarations*** themselves as well.

1. All public***Declarative Items*** are accessible.
2. A private***Library Unit Declaration*** is accessible by other private ***Library Unit Declarations*** only.
3. To access an accessible ***Basic Declaration*** or a ***Library Unit Declarations*** itself, a***Library Unit Declarations*** must use a ***with*** clause of the declaring ***Library Unit Declarations***. To limit the accessibility to private part only, ***private with*** must be used.

1.2 Visibility rules

Visibility rules deal with declarations that can be directly referenced without the help of no special clause.

1. A ***Declarative Item*** can see ***Declarative Items*** declared before in its ***Library Unit Declaration***.
2. A ***Declarative Item*** can see the ***Basic Declarative Items*** of its parent library unit.

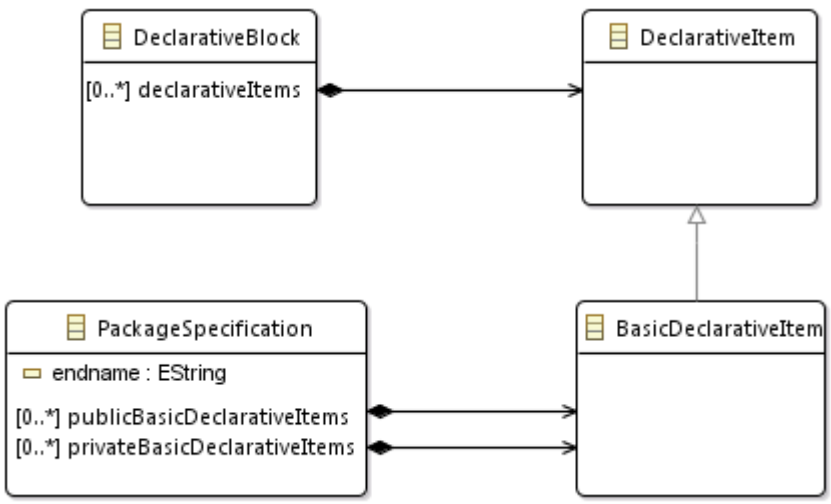

Figure 1.1: Declarative Items

3. Only ***Declarative Items*** of a ***Package Body*** can see ***Basic Declarative Items*** in the private section of its ***Package Specification***.
4. The ***Declarative Items*** of a ***Body*** or a ***Separate subunit*** are hidden from outside.

1.3 Type resolution

Subtyping: A subtype is a subset of its base type. By extension, it can be said that a type is its own subtype.

Derivation: A derived type is a new type defined with the help of a type named the base type.

Universal type : Literal constants are assigned an anonymous universal type from which all types are subtype.

Type resolution's rules on primitive types:

1. Universal type x Type is Type
2. Type x Universal type is Type
3. Subtype of Type x Type is Type
4. Type x Subtype of Type is Type

Class assignation's rule : An object of a given class can be assigned by an object of a subclass of this class. As a result, assignement rule acts as subtyping.

1.4 Subprogram's call resolution

A subprogram is identified by its name and by the type or class of its parameters. In particular, formal parameters must be assignable with the actual parameters. If a subprogram is a primitive[1] of a type then an implicit version of this subprogram exists for every derived type.

1.5 Referencing different kind of object with an unique type

1.5.1 Statement of problem

In Ada syntax, there are some situations when a name may designate different kind of objects :

- function call / variable reference in expressions.
- subprogram's and package's libraries in with and use clauses.

This examples are contrary to an unique type authorized by Xtext to reference to.

[1] A primitive is a subprogram whom at least one parameter is of the type just declared before

1.5.2 General strategy

The rule of the unique referenced type implies to designate a representative class from which it is possible to navigate to the actual referenced object.

1.5.3 Justification of design

Solution A

In view of the previous section, one solution consists in use as representative, a common ancestor to the different possible referenced types. The actual type of the object can then be retrieved thanks to *instanceof* test.

Solution B

A second solution consists in encapsulation in a common container. Assuming that only one encapsulation is valid at the same time, it is then possible to retrieve the actual object.

Xtext demands that it can be getted the name of the object. For this purpose, Xtext allows to derive one of its runtime class ***DefaultDeclarativeQualifiedNameProvider*** and to define *protected QualifiedName qualifiedName([RepresentativeType] ref)* in order to provide a name.

Chapter 2

Ada types referencement

2.1 Statement of problem

Figure 2.1 shows classes involved in Ada typing. Meanwhile and figure 2.2 show classes that references types. The latters delegates to ***Subtype Indication*** [1] the care to reference a type. ***Subtype Indication*** always refers on a type by its name; hence, a cross-reference should be resolved.

2.2 The type of the reference

Type Declaration is not only at the top of the class hierarchy in charge of Ada typing but it defines *name* too that ***DefaultDeclarativeQualifiedNameProvider*** can handle. As a consequence, the type of ***Subtype Indication***::*subtypeMark* should be ***Type Declaration***.

2.3 Where Ada types are declared

Figure 2.1 shows that type declarations ***Data Instance Declaration*** are all ***Basic Declarative Item***.

So, they may appear in public or private section of a ***Package Specification***.

It shows as well they are ***Declarative Item*** too. As such, they may appear in any Ada body or ***Block Statement***.

As a result, all the rules of visibility and accesibility apply. See section 1.2 and 1.1 for details.

2.4 Targets of references

The problem exists because a same type may have more than declaration :

[1] This attribute could be shared by these classes thanks to a common ancestor. Perhaps, scoping could take advantage of.

Figure 2.1: Typing hierarchy

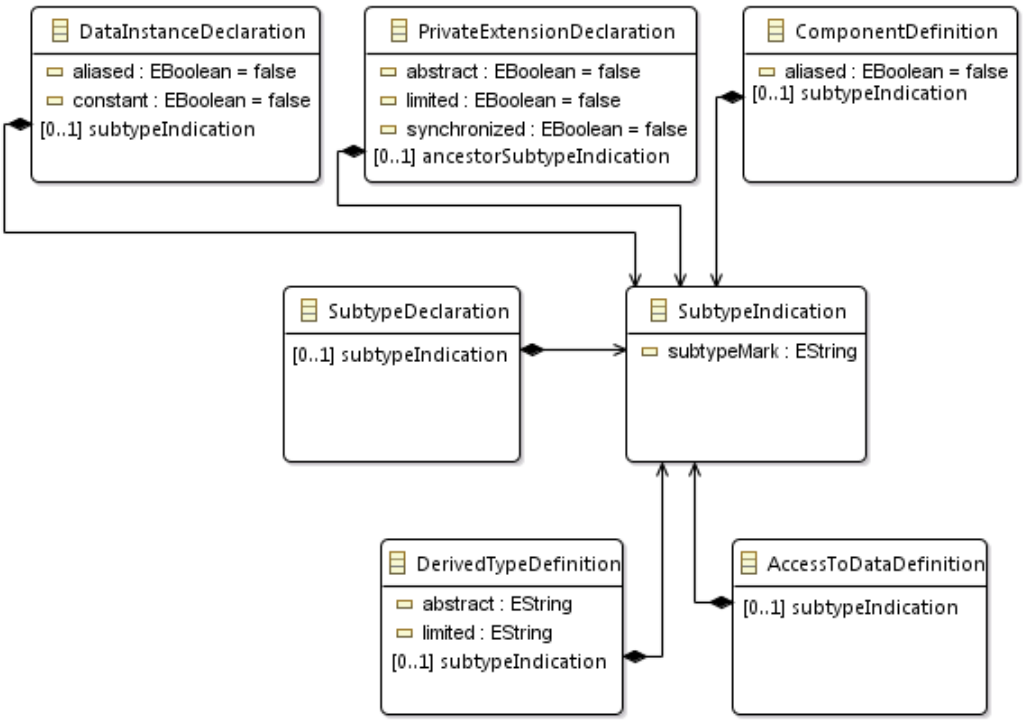

Figure 2.2: References to types

- ***Full Data Type Declatation***
- ***Incomplete Type Declaration***
- ...

The goal for ***Subtype Indication*** is to refer to the most complete definition. However, the most complete definition of the referenced type is the closest from ***Subtype Indication***. So, reverse visiting in AST solves this purpose.

Chapter 3

Variables and functions referencement

By elements of program, it has to be understood :

- data
- functions

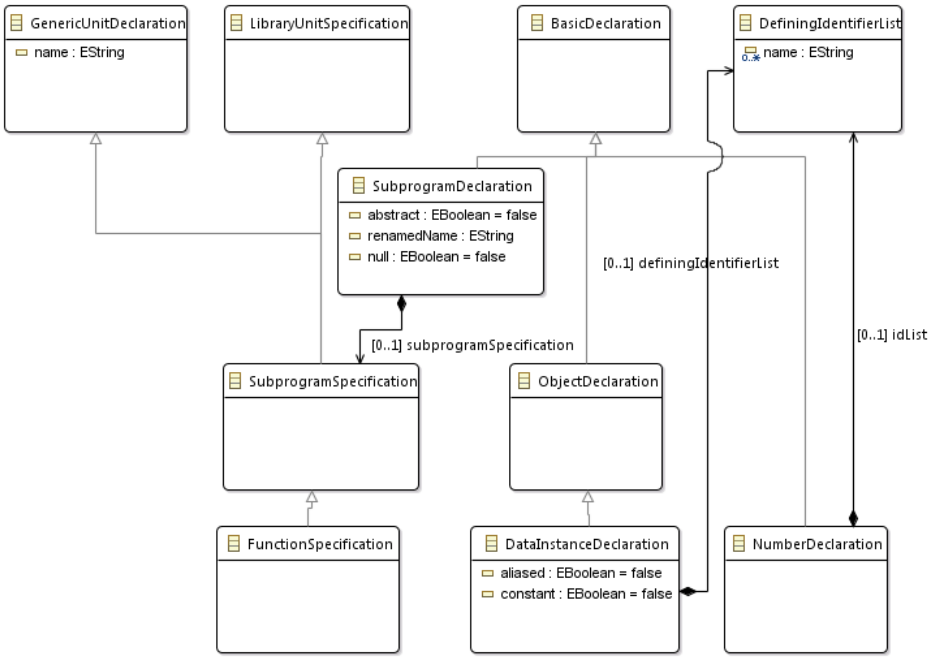

Figure 3.1: Data and Functions : initial design

3.1 Statement of problem

The figure 3.1 shows which kind of named elements may appear in an expressions :

- ***Function Specification***
- ***Data Instance Declaration***
- ***Number Declaration***
- ***Component Item***
- ***Discriminant Specification***

What we learn by this list is that an expression may address various kinds of named entities and elements. The latters are named thanks to ***Defining Identifier List***, ***Function Specification*** is named thanks to ***Generic Unit Declaration***. This raises some problems treated just after.

3.2 Getting the objects definition by their name

The question is how to reference functions, constants and variables in ***Expression*** with an unique type of reference. This is not a trival question watching this in detail.

Indeed, ***Data Instance Declaration*** and ***Number Declaration*** may gather several declarations. The name of these instances are stored in list of ***Defining identifier list***. Meanwhile, the names of functions are defined in ***Generic unit Declaration***.

3.2.1 Solving the problem of referencing to a list of strings as names

Before all, the problem is that ***Defining identifier list*** is not a candidate for being a type of reference because certainly, it defines an attribute, *name* of type ***ecore::EString*** but it is a list of names. Hence, its type, ***ecore::EString*** as primitive type must be replaced by a class, ***Identifier***. As this, it ***Named Element*** may be used as referenced type. From ***Identifier***, the definition of the element can then be accessed through containers links.

3.2.2 The need of unifying function's referencing with others named elements of expressions

Here is the second problem to address. The common way to unify is to define a new class from which ***Subprogram Specification*** and ***Identifier*** derive : ***Named Element***. Because identifiers don't include operation's signs (+, -, *, ...) but ***Function Specification***, yes, an intermediate class in inheritage must be added, ***Direct Name*** for ***Function Specification***.

The final result of these solutions is visible on figure 3.2.

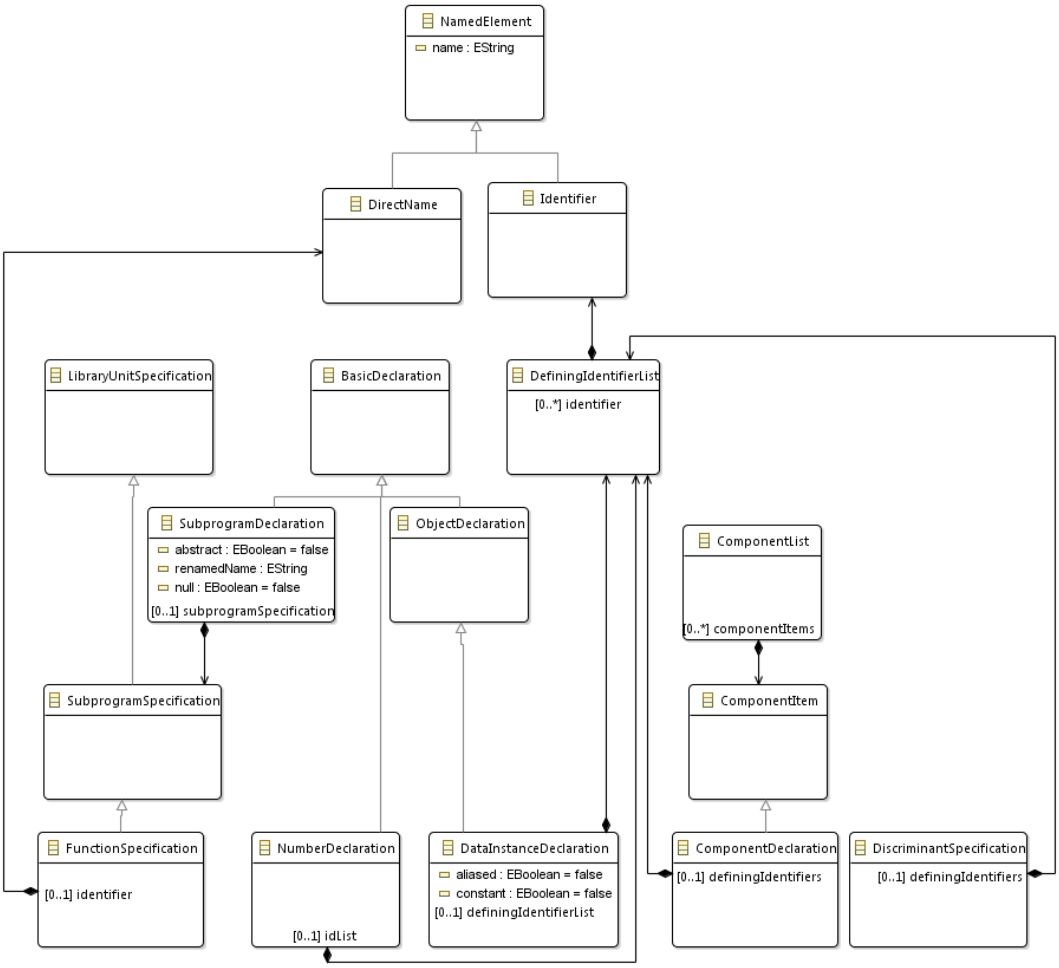

Figure 3.2: Data and Functions : final design

Chapter 4

imports : “with” and “use” clauses

4.1 Statement of problem

Figure 4.1 gives an overview of the problem.

On one part, ***With clause*** references a ***Library unit*** by its name. On the other part, ***Library unit*** doesn’t have a *name* attribute. Instead, *name* can be retrieved in its descendants or in one of their parts.

4.2 Justification of design

Provided that *name* attribute is dispatched in class hierarchy of ***Library unit***, the solution described in section 1.5.3 is the most suitable.

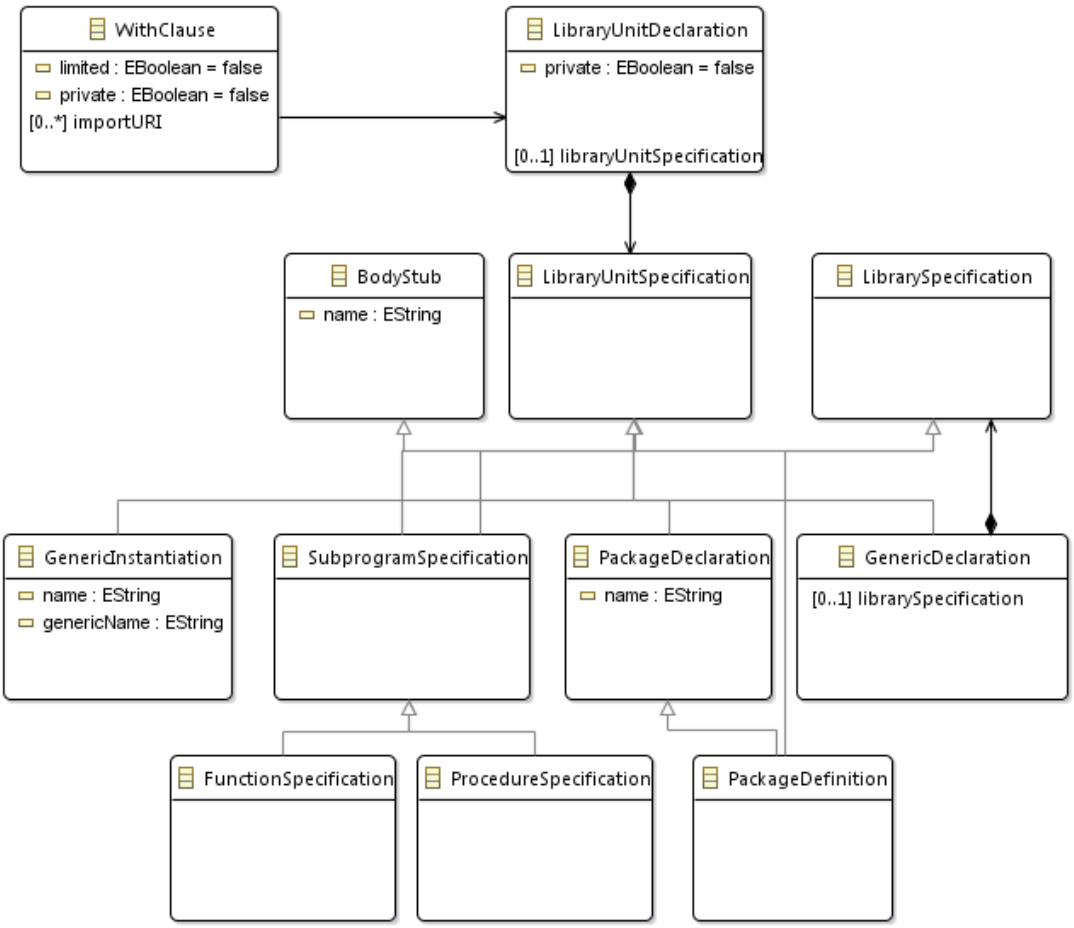

Figure 4.1: Library units

Chapter 5

Class *Name*

5.1 Statement of problem

The syntax by the Ada reference manual tells a ***Name*** is :

- either a direct name (identifier or operation symbol)
- or an explicit dereference (deferencement of an access)
- or an indexed component (a cell of an anrray)
- or a slice (a subarray)
- or a selected component (the component of a record)
- or an attribute reference (of a name)
- or a type conversion
- or a function call
- or a character literal (to designate ASCII indexes by their character literals. I can see nothing else.)

5.2 shows that Name is used to designate entries (with class ***Requeue Statement***), tasks and exceptions too.

As a result, a Name designates either a data, a type, a subprogram, an entry, a task or an exception.

This point is a problem for references because identifiers in class ***Name*** must refer to an unique type.

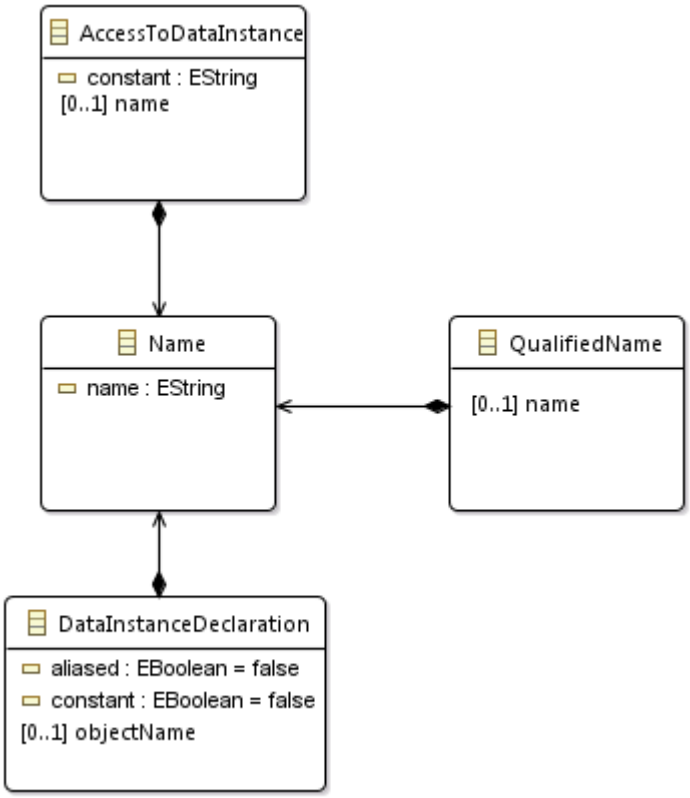

Figure 5.1: Class Name used for data

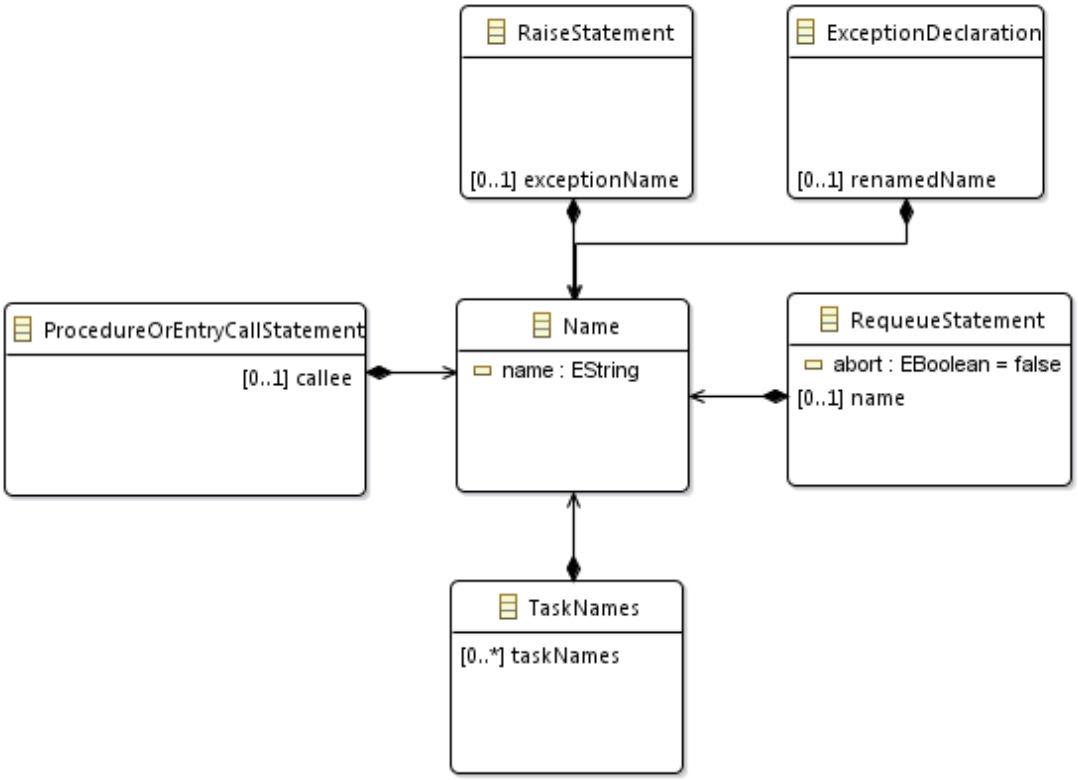

Figure 5.2: Class Name used for programs

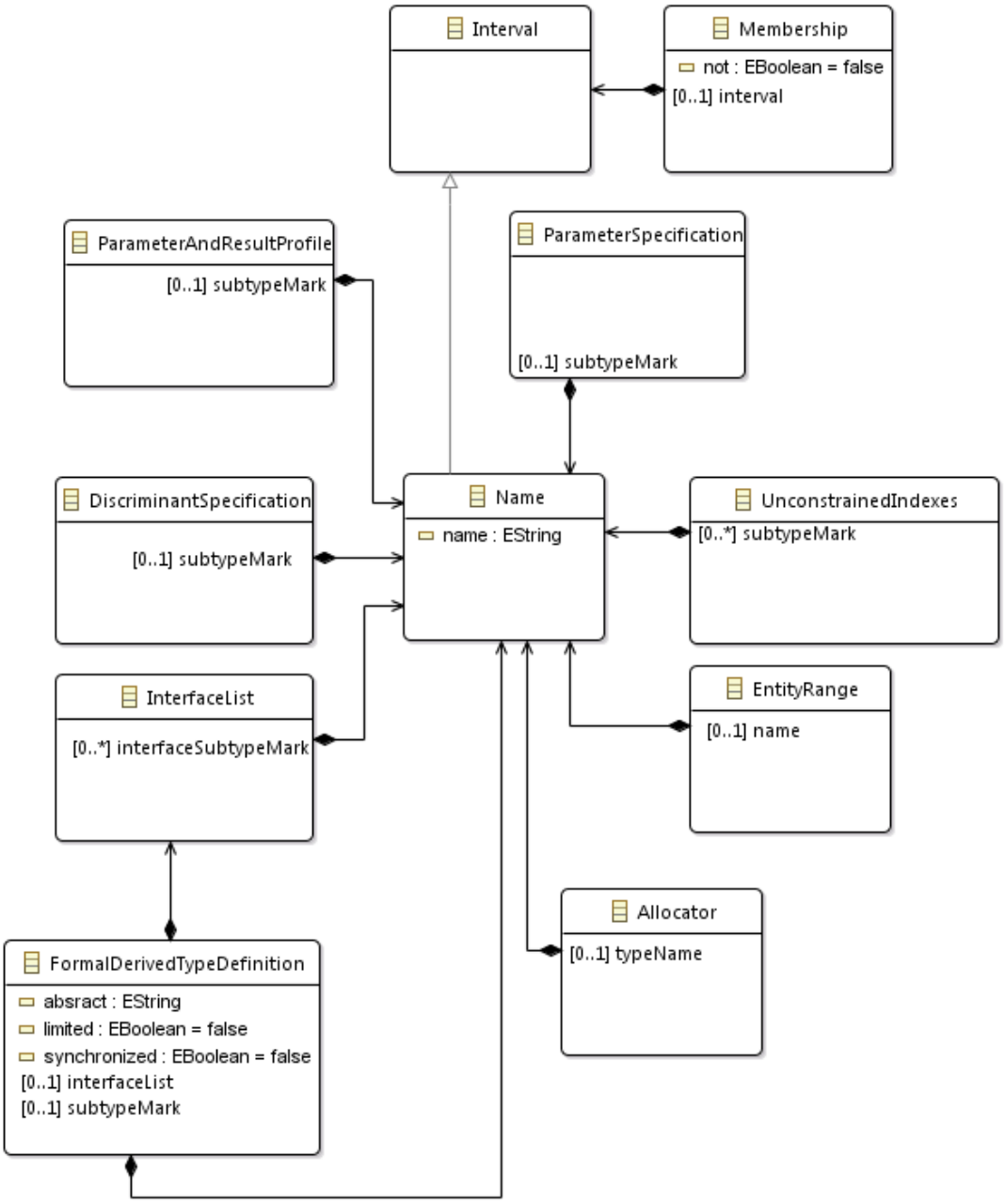

Figure 5.3: Class Name used for types

5.2 General strategy

5.2.1 Solution A

A solution consists in applying strategy of section 1.5.3. This assumes to define a common ancestor to these classes letting the cross-reference find by name the correct object.

5.2.2 Solution B

Name is very generic and used to designate many classes and all its syntax is not always used. Hence, another solution consists in splitting ***Name*** in several ***Name*** specialized classes.

Method for *Name* specialization

For each class having an attribute of type Name, determine the type of the referenced object.

For this type, localize the attribute *name*.

If the attribute *name* is defined in an ancestor, inventory its descendants.

For each of its descendant, determine the general form of its qualified name based on the syntax and semantic of its Ada declaration(s) and if a language-defined attribute has its type.

Create a new class ***Name*** rid of unnecessary syntactic constructions.

The following sections apply this algorithm.

Name designating an exception's name

5.2 shows ***Raise Statement*** and ***Exception Declaration*** that designate exception by its name of type ***Name***.

The declaration of a exception is immuable : an identifier followed by the keyword *exception*. This means that its qualified name is always the including packages names followed by its identifier : Exception qualified name : (Package identifier)* Exception's identifier.

Name designating a task

Tasks can be declared as data :

1. They can be indexed
2. referenced by an access variable
3. discriminated

It doesn't exist language-defined attribute of type task. Its name is an identifier.

As a result, ***Task Name*** doesn't need :

- Attribute designator : no predefined attribute is a task.
- operator symbol neither character literal literals

Name designating a procedure or an entry call

As a procedure and entry calls are syntactically indistiguable, their call-name are the same. Here, optimizations are analyzed separately and unified at the end of this section.

Name designating an entry An entry can :

1. be indexed
2. have parameters (i.e: a formal part)

It doesn't exist language-defined attribute of type entry. Its name is an identifier. It can't be referenced by an Ada access.

As a result, ***Entry Name*** doesn't need :

- Attribute designator : no predefined attribute is a task.
- operator symbol neither character literal literals
- .all as an entry can't be designated by a type.

Name designating a procedure A procedure can :

1. be referenced by an Ada access
2. be indexed indirectly through an Ada access.
3. have parameters (i.e: a formal part)
4. can adopt the language-defined attribute's notation

Its name is an identifier.

As a result, ***Procedure Name*** doesn't need :

- operator symbol neither character literal literals

Conclusion As a result, A procedure/entry call-name doesn't need :

- operator symbol neither character literal literals

Chapter 6

Reference to Ada entry

Figure 6.2 shows the classes that reference entries :

1. ***Accept Statement***
2. ***Procedure Or Entry Call Statement***
3. ***Entry Body***

6.1 The scope

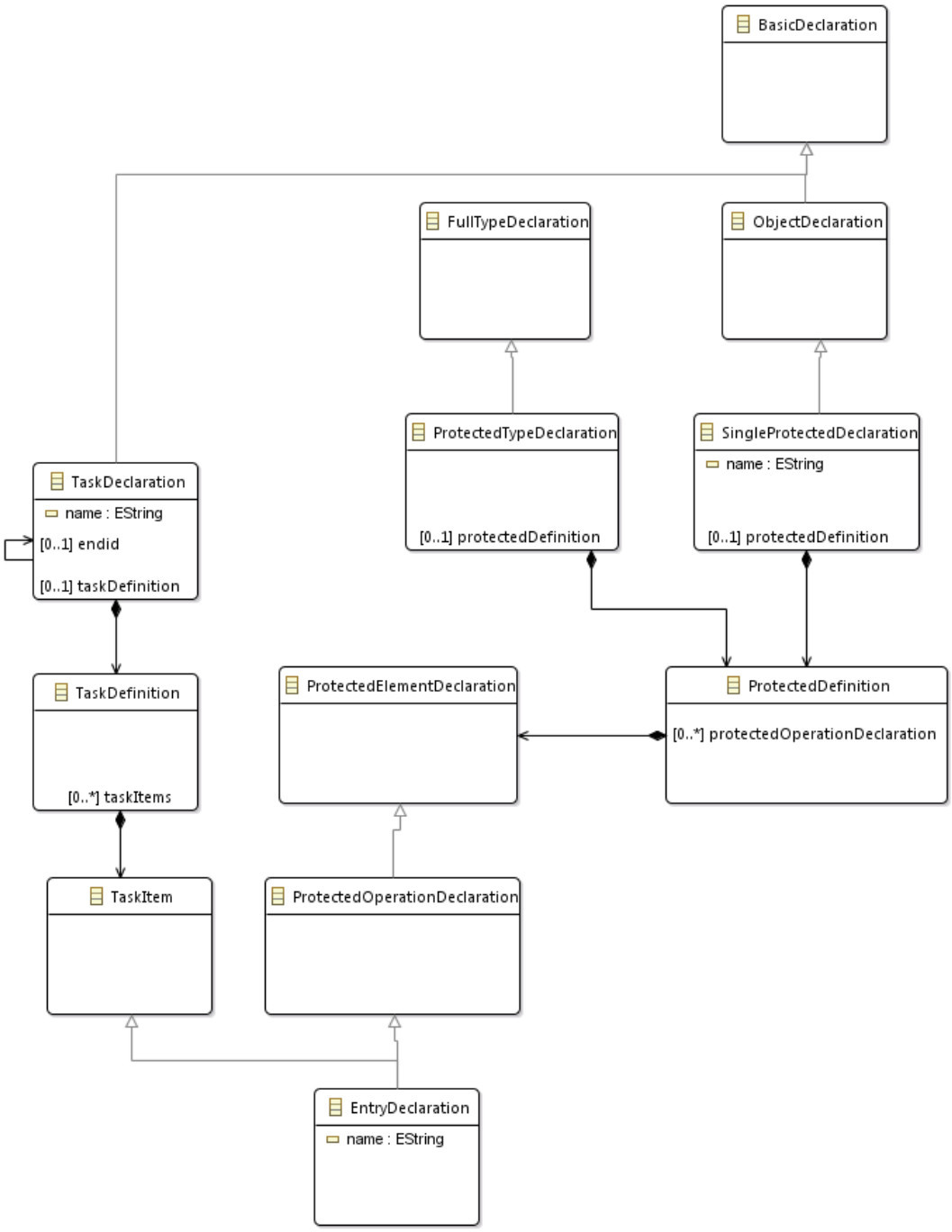

Figure 6.1: Entry declaration

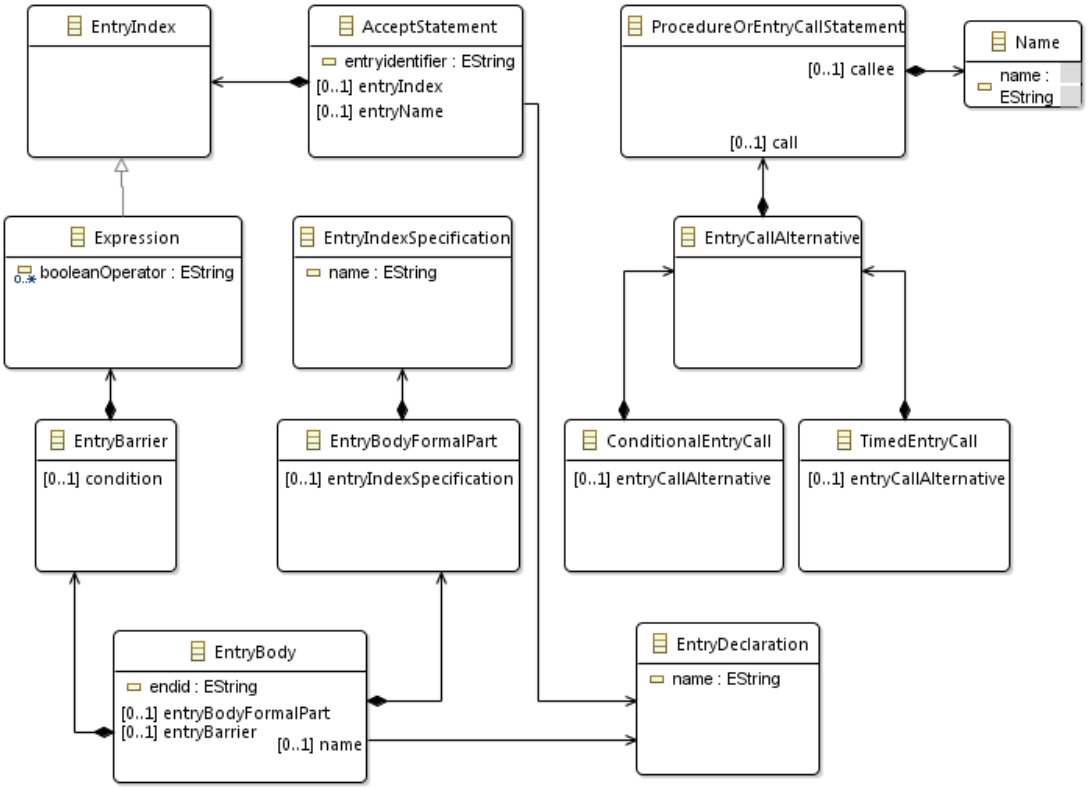

Figure 6.2: Entry call

Chapter 7

Linking with Xtext

7.1 Names and namespaces

Linking deals with referencing by name. To ensure names are unique, container's names are used as namespaces in order to disambiguate. For instance, the local variable of a method is preceded by the name of this method which is preceded by the name of the class and so on.

Concatenating these names is performed by descendants of ***IQualifiedNameProvider***. By default, ***DefaultDeclarativeQualifiedNameProvider*** is the implementator. This function of this class is based on attributes *name* of the ecore classes of the meta-model. Nevertheless, the xtext grammar of the user could not respect for good reasons the convention used by the default implementation. That's why it is possible to derive ***DefaultDeclarativeQualifiedNameProvider*** to adapt the function. More, do not forget to add in the descendant of ***DefaultRuntimeModule*** :

```
public Class<? extends org.eclipse.xtext.naming.IQualifiedNameProvider>
bindIQualifiedNameProvider()
{
    return YourQualifiedNameProvider.class;
}
```

A name preceded by its containers names is called a ***QualifiedName***.

7.2 Named objects container

All named objects are stored in a global container called the index.