Using Rose Ada for Forward and Reverse Engineering

Rational Rose®

VERSION: 2001A.04.00

PART NUMBER: 800-024465-000

support@rational.com
http://www.rational.com

Rational®
the e-development company™

COPYRIGHT NOTICE

U.S. GOVERNMENT RIGHTS NOTICE

TRADEMARK NOTICE

Printed in the U.S.A.

Contents

Preface

Rational Rose®, hereafter referred to as Rose, is a comprehensive, integrated programming environment that supports the development of complex software systems. This manual presents the concepts needed to use all of the Rose functionality to its fullest extent:

- Software system architecture
- Rational subsystems (high-level software partitions and their interfaces)
- Software development, integration, and release processes

Audience

This manual is intended for:

- Database developers and administrators
- Software system architects
- Software engineers and programmers
- Anyone who makes design, architecture, configuration management, and testing decisions

This manual assumes you are familiar with a high-level language and the life-cycle of a software development project.

Other Resources

- For more information on training opportunities, see the Rational University Web site at http://www.rational.com/university.

Contacting Rational Technical Publications

To send feedback about documentation for Rational products, please send e-mail to our Technical Documentation Department at techpubs@rational.com.

Contacting Rational Technical Support

If you have questions about installing, using, or maintaining this product, contact Rational Technical Support.

Your Location	Telephone	Fax	E-mail
North America	(800) 433-5444 (toll free) (408) 863-4000 Cupertino, CA	(781) 676-2460 Lexington, MA	support@rational.com
Europe, Middle East, Africa	+31 (0) 20-4546-200 Netherlands	+31 (0) 20-4545-201 Netherlands	support@europe.rational.com
Asia Pacific	+61-2-9419-0111 Australia	+61-2-9419-0123 Australia	support@apac.rational.com

Note: When you contact Rational Technical Support, please be prepared to supply the following information:

- Your name, telephone number, and company name
- Your computer's make and model
- Your operating system and version number
- Product release number and serial number
- Your case ID number (if you are following up on a previously-reported problem)

Introducing Rational Rose Ada

1

Contents

This chapter is organized as follows:

What is Code Generation?

Rose Ada provides the Ada Generator to generate Ada units from information in a Rose model. These units contain Ada code constructs that correspond to the notation items (classes, relationships, and adornments) you have defined in the model using diagrams and specifications. Rose Ada supports either Ada 95 or Ada 83 code generation with the Ada 95 or the Ada 83, respectively, add-in to Rational Rose. The commands for the Ada Generator are located in the **Ada 95** / **Ada 83** submenu of the Rose **Tools** menu.

The Ada Generator provides code-generation properties that control the kinds of Ada code constructs that are generated for the various kinds of notation items in the model. You can use the default values for these properties or you can specify different values to generate the code you want.

Using Code Generation

To generate Ada 95 code:

1. Activate the Ada 95 add-in using the Add-In Manager, which is accessible from the **Add-Ins** menu.
2. Set the default language for your model to be Ada 95. Do this by clicking **Tools > Options** and click the **Notation** tab.
3. In the **Default Language** list, select **Ada 95**.

To generate Ada 83 code:

1. Activate the Ada 83 add-in using the Add-In Manager, which is accessible from the **Add-Ins** menu.
2. Set the default language for your model to be Ada 83. Do this by clicking **Tools > Options** and click the **Notation** tab.
3. In the **Default Language** list, select **Ada 83**.

You may generate a different language for some classes by associating them with a component that has a different language.

What is Reverse Engineering?

Reverse Engineering generates a model from compiled code.

Rose can analyze Ada code compiled with Rational Apex and generate a Rose model containing class and component diagrams that present a high-level view of the code.

This capability is only available for Ada units that have been compiled with the Apex compiler and that are in the installed (analyzed) or coded states.

The reverse engineering tool can create both class diagrams and component diagrams. Class diagrams show the high-level relationships between Ada units and types, and the operations and data structures associated with each type. Component diagrams come in two forms:

- An Ada unit diagram, which displays the “with” structure of the Ada units in a program, independent of subsystem structure.
- A subsystem diagram, which displays the import structure of the views you specify.

Each view displays the “with” structure of the Ada units in that view.

Using Reverse Engineering

Select the Ada unit or view you wish to diagram, and click **Rose > Ada > Reverse Engineer**. The **Reverse Engineer** dialog box appears, allowing you to modify various options. Click **OK** or **Apply** to create the model file.

Once you have created the model file, you can load it into Rose. Select the file in the directory viewer (you may need to click **File > Redisplay** first). Then choose **Start Rose** from the **Rose > Ada** submenu. This will invoke Rose and display the model.

Note: For traversal to work, you must invoke Rose from the Apex menu. If Rose is already running before you started Apex, exit Rose and restart from the Apex menu command.

Mapping the UML Notation to Ada 95 — Code Generation

2

Contents

This chapter is organized as follows:

Introduction

This chapter details the forward-engineering mapping between the UML notation and the Ada 95 programming language.

Roughly speaking, classes are transformed into types declared in library packages, utilities are transformed into library packages, attributes and relationships are transformed into record components. The main source of information for the code generation are the class diagrams. Code generation properties may be used to gain finer control over the way that code is produced. If component diagrams are present, some of the information they contain is also used by the code generator.

Because UML and Ada use the word “package” to designate two different concepts, this document uses the phrase “UML package” for a package in the UML acceptation, and the word “package” without qualification for an Ada package. When necessary, the phrases “logical UML package” and “component UML package” are used to refer to UML packages in the logical view or in the component view, respectively.

Name Space

This section defines how the naming of entities in the UML notation corresponds to the naming of declarations in the generated Ada 95 code.

The following rules define the legal names for entities of a model that is used to generate Ada 95 code:

- The name of any entity in a model may have the form:

  ```
  identifier
  ```

  where **identifier** is a legal Ada 95 identifier. In other words, the name of any entity name may be an Ada simple name.

- The name of any class or module may also have the form (using the same BNF notation as in the *Ada 95 Reference Manual*):

  ```
  identifier{.identifier}
  ```

  where **identifier** is a legal Ada 95 identifier. In other words, the name of any class or module may be either an Ada simple name or an Ada expanded name.

- The name of any normal or parameterized class (but not an utility or a bound class) may also have the form:

  ```
  identifier{.identifier}:identifier
  ```

In other words, a class name must either be an Ada simple name, an Ada expanded name, or a pseudo-expanded name (an expanded name followed by a colon and an identifier: this is called the *colon notation* hereafter).

The code generator checks the legality of names, in particular in terms of consistency with the *Ada Reference Manual*.

From the name of a class the code generator derives the name of a library-level package (the package where the type and operations associated with the class are declared) and the name of a type (the type associated with the class) as follows:

- If the class is associated with a module, the package name is the name of the associated module. The type name is given by the code generation property TypeName, unless the class name uses the colon notation, in which case the type name is the segment following the colon in the class name.
- If the class is not associated with a module, and its name uses the colon notation, the package name is made of the name segments preceding the colon, and the type name is the name segment following the colon.
- If the class name does not use the colon notation, the package name is the name of the class, and the type name is given by the code generation property TypeName.

The code generation property TypeName defaults to “Object”.

These rules support two different approaches to naming the classes in the Rose model: either the class name reflects the hierarchy of units, or the class name is for design purposes only, and the hierarchical unit structure is defined using the mapping to modules. In the former case, the colon notation may be used to make the type names explicit in the class diagram. Alternatively, the type names may be specified using the property TypeName.

For utilities, similar rules are used, except that there is no type declaration, so the TypeName property is irrelevant, and the colon notation is not allowed.

Note that it is possible for several classes to map to types declared in the same Ada package, either by using the colon notation, or by using associations between classes and modules. However, such a mapping is only legal if all classes that map to a given module are part of the same UML package. In the case of associations between classes and modules, the correspondence between logical and component UML packages ensure that the mapping is always legal. In the case of the colon notation, the legality of the mapping is checked by the code generator.

Name Resolution

While a large part of the information in a model is entered graphically by means of relationships and adornments, there are a number of situations where the user enters textually in the model a piece of information which designates a class. Examples of such situations include the definition of the type of attributes or parameters.

The code generator performs name resolution to determine the Ada type to be generated in these circumstances. To explain how the name resolution works, consider the case of class A having an operation Op with a parameter (or result) type written as “B”. The code generator performs the following operations:

- It finds all the relationships originating at class A. Note that this includes in particular the dependency relationships, which are not otherwise used for code generation (except that they result in *with* clauses, as explained below). As a consequence, dependency relationships may be used to introduce visibility between classes for the sake of name resolution.
- It looks at the names of all classes which are the targets of these relationships.
- If any of these classes is named “B” (the comparison is case-insensitive, but must otherwise be exact), the type of the parameter in the generated code is the Ada type generated for class B. This ensures that the generated code is legal. Assuming that the default properties are used for class B, the generated code looks like:

  ```
  procedure Op (X : B.Object);
  ```

- If any of the target classes is named “B:T” (the comparison with the name segments preceding the colon is case insensitive, but must otherwise be exact; the name segment following the colon is ignored), the type of the parameter in the generated code is the Ada type generated for class B:T, i.e. B.T. The generated code looks like:

  ```
  procedure Op (X : B.T);
  ```

- If none of the target classes is named “B” or “B:T”, the type of the parameter in the generated code is simply copied from the model. In this case, the generated code looks like:

  ```
  procedure Op (X : B);
  ```

Note that this resolution mechanism applies regardless of whether the parameter type is a simple name (like “B”), an expanded name (like “B.C”) or a colon notation (like “B:T” or “B.C:T”). If the parameter type uses the colon notation, it will only match a class name that also uses the colon notation. In all cases, the generated code references the type name, not the class name.

It may be that there are ambiguities, for instance if the parameter type is given as “B” and the set of target classes includes classes named “B:T1” and “B:T2”. In this case, an error message is emitted, and the parameter type has to be made more explicit.

This name resolution mechanism makes it possible to use class names everywhere in the model, and defer the mapping of class names to Ada type names by setting the TypeName code generation property and / or the mapping of classes to modules. Changing the mapping of classes to types and modules doesn't require to change the attributes, parameters, etc., scattered throughout the model.

Of course, the user may always enter an Ada type name for the type field of an attribute or parameter, since such a name will not match any class name, and will thus be copied verbatim in the generated code. This may be useful for predefined types like Integer or Calendar.Time, for which it would be cumbersome to create a class in the model. However, it is strongly recommended that class names, not type names, be used wherever possible in order to ease maintenance of the model if the mapping of classes to types ever has to change.

Code Generation Properties and Consistency

Various entities in a model have associated code generation properties which may be used to control the way that the code is produced. Often, there exist consistency requirements between the values of the code generation properties of one or several entities.

These requirements come most of the time from language rules, and ensure that the generated code is correct. To take an example, in Ada 95, it is not possible to specify, when declaring a derived type, if it is limited: it just inherits its limited-ness from the root of the derivation tree. In Rose / Ada, the code generation property IsLimited may be used to control whether or not the type generated for a given class is limited. Clearly, it does not make sense for a root class A to have IsLimited set to False, and for a class B, subclass of A, to have IsLimited set to True.

In practice however, having to set code generation properties in a consistent manner over large models may become burdensome. To avoid this, some code generation properties are said to be *dominant* over others. A dominant property determines the code generated, and the dominated property is ignored altogether, even if it specifies a different code generation. For instance, if a root class has IsLimited set to True, the code generation property IsLimited of its subclasses is not even considered: these classes will all be limited.

In some circumstances, a property is dominant only when it has a specific value (or set of values). For instance, the property TypeImplementation dominates IsLimited only when it has the values Task or Protected (because task types and protected types are always limited).

One may however wish to be able to track and correct inconsistencies where, for instance, IsLimited is set to True on the root class but to False on some of its subclasses, Such inconsistencies may turn out to be a problem in organizations having strict quality assurance policies. To ease detection of inconsistencies, the code generator emits a warning message whenever it detects that a dominated property has a value which is inconsistent with the dominant property.

Classes

If a “normal” class is associated with a module, that module must be a non-generic package.

Normally, the type generated to represent objects of the class is a non-limited, private type. This can be controlled using the code generation properties IsLimited and TypeVisibility attached to the class:

- For a class which has no superclass, the boolean code generation property IsLimited may be set to True, in which case a limited type is generated. The property IsLimited of a root class dominates the same property for its subclasses.
- TypeVisibility can take two values: Public and Private. Setting this property to Public causes the full type declaration to be generated in the visible part of the associated library package. Setting it to Private causes a private type to be generated. TypeVisibility defaults to Private.

The scheme used to generate the code associated with a class is governed by the code generation properties TypeImplementation and TypeDefinition.

If TypeDefinition is not empty, it dominates TypeImplementation, and the type generated uses the contents of that property (technically, the contents of TypeDefinition must be an Ada type definition). If for instance TypeDefinition is set to “range -1 .. 3” then the generated type declaration is:

```
type Object is range -1 .. 3;
```

If TypeDefinition is empty (the default), TypeImplementation is used to control the code generation scheme. TypeImplementation can take one of five values: Tagged, Record, Mixin, Task or Protected. In the rest of this section, we consider each of these schemes in turn. In this discussion, unless otherwise specified we assume the default values for properties IsLimited and TypeVisibility.

Tagged Implementation

The class corresponds to a tagged type. If the class has no superclass, the declaration of the corresponding type is:

```
type Object is tagged private;
```

If the class has a superclass, the declaration of the corresponding type is:

```
type Object is new Superclass.Object with private;
```

If the class has more than one superclass, we are in a situation of multiple inheritance, which is covered later.

If the class is abstract, the associated type declaration includes the reserved word *abstract*:

```
type Object is abstract tagged private;
```

```
type Object is abstract new Superclass.Object
                                    with private;
```

Record Implementation

In this scheme, polymorphism (if any) is implemented using records with variants. This means that if the class has any subclass, an enumeration type is created to represent all possible variants, and the record type declaration associated with the class is a variant record gathering the attributes and relationships of all the subclasses.

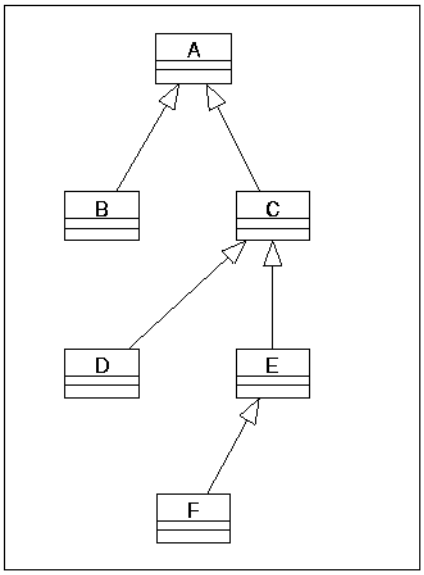
The properties TypeImplementation and IsLimited of the root class dominate those of the subclasses. Also, none of the classes may be marked abstract.

There are two ways that the record mapping can be implemented, so the Record scheme is further controlled by the code generation property RecordImplementation associated with the root class. This property can take the two values SingleType and MultipleTypes. The property RecordImplementation of the root class dominates the same property for its subclasses.

Regardless of the mapping chosen, for a class which has no superclass and no subclasses, the generated code is simply (assuming the default values for the properties TypeVisibility and IsLimited):

```
package A is
    type Object is private;
private
    type Object is
        record
            ...
        end record;
end A;
```

When discussing the two possible record implementations in more complex cases, we'll use the following generalization hierarchy as an example:



SingleType Record Implementation

In this scheme, a single record type is created for the complete generalization hierarchy. An enumeration type is created that lists all the variants, and the structure of the record corresponds to that of the generalization tree. For each subclass, a package is created that declares a subtype or derived type with a discriminant constraint (depending on the property IsSubtype). For leaf classes, the discriminant is omitted. The code generated is as follows:

```
package A is    -- The root package
    type A_Kinds is (Some_A, Some_B, Some_C,
                            Some_D, Some_E, Some_F);
    type Object (Kind : A_Kinds := Some_A) is private;
private
type Object (Kind : A_Kinds := Some_A) is
        record
            Ca : Integer;
            case Kind is
               when Some_B =>
                    Cb : Integer;
               when Some_C | Some_D | Some_E | Some_F =>
                    Cc : Integer;
                    case Kind is
                        when Some_D =>
                            Cd : Integer;
```

```
                        when Some_E | Some_F =>
                            Ce : Integer;
                            case Kind is
                                when Some_F =>
                                    Cf : Integer;
                                when others =>
                                    null;
                            end case;
                        when others =>
                            null;
                    end case;
                when others =>
                    null;
            end case;
        end record;
end A;
with A;
package B is                -- A leaf
    type Object is private;
private
    type Object is new A.Object (A.Some_B);
end B;
with A;
package C is        -- An intermediate node
    subtype C_Kinds is A.A_Kinds
                            range A.Some_C .. A.Some_F;
    type Object (Kind : C_Kinds := A.Some_C) is private;
private
    type Object (Kind : C_Kinds := A.Some_C) is
                            new A.Object (Kind);
end C;
```

The prefix used to generate the names of the enumeration literals is specified using the code generation property EnumerationLiteralPrefix of the class. This property defaults to “A_”. In the above examples, we have assumed for readability that it was set to “Some_”.

Note that the code generator orders the enumeration literals in a way that is suitable for the constraints on subtype Kinds in the intermediate nodes.

The property TypeVisibility of the root class dominates the same property for subclasses.

The SingleType mapping may result in name conflicts: if two components of two classes in a generalization hierarchy have the same name, they will clash when they are put together in the above record type declaration. It is the user’s responsibility to avoid such conflicts.

MultipleTypes Record Implementation

In this scheme, one record type is created for each class in the hierarchy, and these types are aggregated in a discriminated record at each level, according to the structure of the generalization hierarchy. For subclasses, a subtype or derived type with a discriminant constraint is created (depending on the property IsSubtype). For leaf classes, the discriminant is omitted.

```
package A_Record_Kind is
    type A_Kinds is (Some_A, Some_B, Some_C,
                            Some_D, Some_E, Some_F);
end A_Record_Kind;
with A_Record_Kind;
with B;
with C;
package A is
    use A_Record_Kind;
    subtype A_Kinds is A_Record_Kind.A_Kinds;
    type Object (Kind : A_Kinds := Some_A) is private;
private
type Object (Kind : A_Kinds := Some_A) is
        record
            Ca : Integer;
            case Kind is
               when Some_B =>
                    The_B : B.Object;
               when Some_C | Some_D | Some_E | Some_F =>
                    The_C : C.Object (Kind);
               when others =>
                    null;
            end case;
        end record;
end A;
package B is
    type Object is private;
private
    type Object is
        record
            Cb : Integer;
        end record;
end B;
with A_Record_Kind;
with D;
with E;
package C is
    use A_Record_Kind;
    subtype C_Kinds is A_Kinds range Some_C .. Some_F;
    type Object (Kind : C_Kinds := Some_C) is private;
private
    type Object (Kind : C_Kinds := Some_C) is
        record
            Cc : Integer;
            case Kind is
```

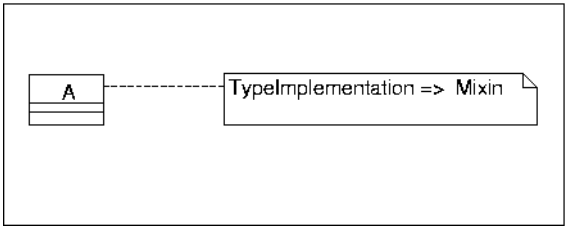
```
                when A_Record_Kind.Some_D =>
                     The_D : D.Object(Kind);
                when A_Record_Kind.Some_E |
                       A_Record_Kind.Some_F =>
                     The_E : E.Object(Kind);
                when others =>
                     null;
            end case;
        end record;
end C;
```

As before, the prefix used to generate the names of the enumeration literals is specified using the code generation property EnumerationLiteralPrefix of the root class, which was set to “Some_” in the above example. Also, the prefix used to generate the names of the intermediate record components is given by the code generation property RecordFieldPrefix of the root class (this property defaults to “The_”).

Finally, the name of the auxiliary package used to declare the enumeration type Kinds is given by the code generation property RecordKindPackageName of the root class. This property defaults to “${class}_Record_Kinds”.

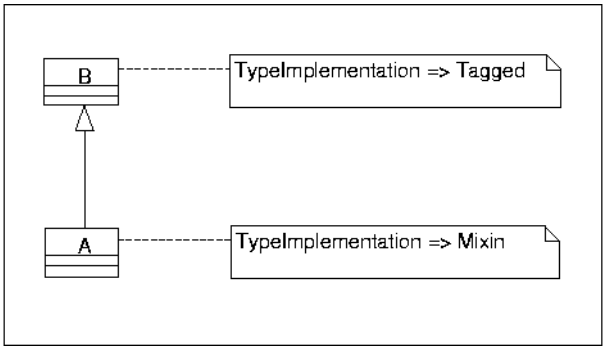
Mixin Implementation

A class whose TypeImplementation property is set to Mixin must be abstract. If that class has no superclass (see figure), the following code is generated:




```
generic
    type Superclass is abstract tagged private;
package A is
    type Object is abstract new Superclass with private;
    -- declaration of the operations
    -- of the class here.
private
    type Object is new Superclass with
        record
            -- declaration of the attributes
            -- and relationships
            -- of the class here.
        end record;
end A;
```

If the class has (exactly one) superclass, B, then B must have its TypeImplementation property set to Tagged (see figure), and the generic formal part above is changed as follows:




```
with B;
generic
    type Superclass is abstract new B.Object with private;
package A is ...
```

Classes implemented according to the Mixin scheme are used in multiple inheritance situations as explained later on.

Task Implementation

A class whose TypeImplementation property is set to Task must not be abstract, and its code generation property IsLimited is dominated. Also, its operations must all be procedures (as opposed to functions). A task type is generated for such a class.

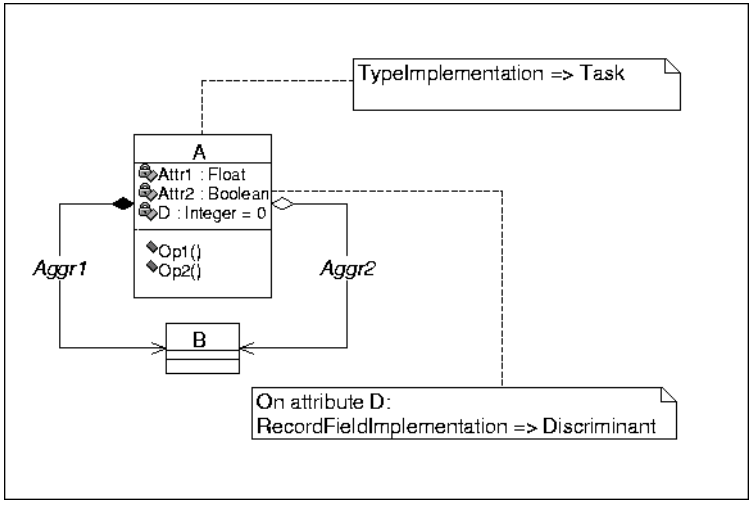
The operations are transformed into entries, and their SubprogramImplementation property is dominated. Depending on the visibility of each operation, the entry is declared either in the visible part or in the private part of the task type. No implicit parameter is ever generated for an operation in the Task mapping, because the implicit parameter is the task itself: TypeImplementation dominates ImplicitParameter.

For each visible operation of the class, a procedure is also generated in the visible part of the package that declares the task type. This procedure has the same profile as the corresponding entry of the task, except for an additional parameter that designates the object being operated upon. The name of this additional parameter is given by the code generation property ImplicitParameterName of the class. The body of each of these procedures simply calls the corresponding entry of the given task object.

The attributes and “has” relationships whose property RecordFieldImplementation is either Discriminant or AccessDiscriminant are transformed into discriminants, as for any composite type. The attributes and “has” relationships whose property RecordFieldImplementation is Component, and the associations, are transformed into variables declared in the task body.

Accessor operations (Get and Set) are never generated for attributes of a class whose TypeImplementation property is Task (in other words, GenerateGet and GenerateSet are dominated).

An example of code generated for the Task mapping is as follows:




```
package A is
    type Object (D : Integer := 0) is limited private;
    procedure Op1 (This : Object);
private
    task type Object (D : Integer := 0) is
        entry Op1;
    private
        entry Op2;
    end Object;
end A;

with B;
package body A is

    procedure Op1 (This : Object) is
    begin
        This.Op1;
    end Op1;

    task body Object is
```

```
        Attr1 : Float;
        Attr2 : Boolean := False;
        Aggr1 : B.Object;
        Aggr2 : B.Handle;
        ...
    end Object;

end A;
```

Classes implemented according to the Task mapping cannot be used in generalization relationships.

Protected Implementation

A class whose TypeImplementation property is set to Protected must not be abstract, and its code generation property IsLimited is dominated. A protected type is generated for such a class.

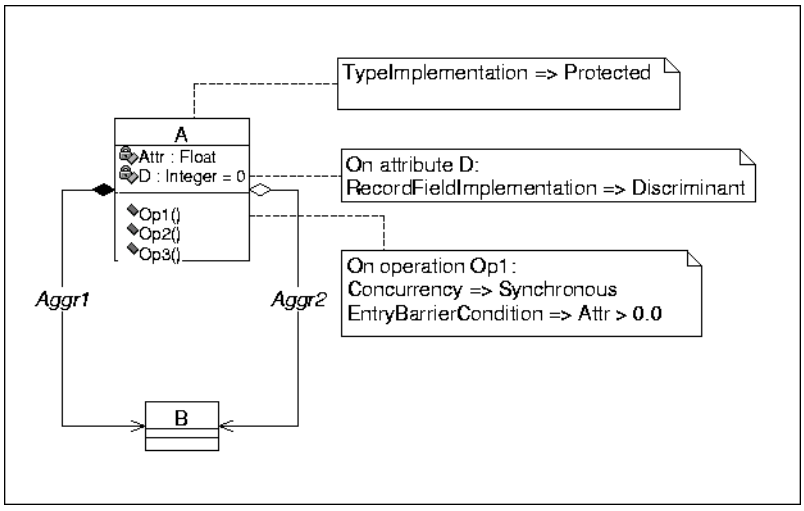
The operations are transformed into protected functions or protected procedures, except that an operation whose concurrent behavior is specified as synchronous is transformed into an entry. The code generation property EntryBarrierCondition of such an operation contains the boolean expression used for the barrier of the entry body. This property defaults to “True”.

Depending on the visibility of each operation, it is declared either in the visible part or in the private part of the protected type. No implicit parameter is ever generated for an operation in the Protected mapping, because the implicit parameter is the protected object itself: TypeImplementation dominates ImplicitParameter.

For each visible operation of the class, a subprogram is also generated in the visible part of the package that declares the task type. This subprogram has the same profile as the corresponding protected subprogram, except for an additional parameter that designates the object being operated upon. The name of this additional parameter is given by the code generation property ImplicitParameterName of the class. The body of each of these subprograms simply calls the corresponding protected subprogram of the given protected object.

The attributes and “has” relationships whose property RecordFieldImplementation is either Discriminant or AccessDiscriminant are transformed into discriminants, as for any composite type. The attributes and “has” relationships whose property RecordFieldImplementation is Component, and the associations, are transformed into components of the protected object (and are thus declared in the private part).

An example of code generated for the Protected mapping is as follows:




```
package A is
    type Object (D : Integer := 0) is limited private;
    procedure Op1 (This : Object);
    function Op2 (This : Object) return Integer;
private
    protected type Object (D : Integer := 0) is
        entry Op1;
        function Op2 return Integer;
    private
        procedure Op3;
        Attr : Float;
        Aggr1 : B.Object;
        Aggr2 : B.Handle;
    end Object;
end A;

with B;
package body A is

    procedure Op1 (This : Object) is
    begin
        This.Op1;
    end Op1;

    function Op2 (This : Object) return Integer is
    begin
        return This.Op2;
    end Op2;

    protected body Object is
        entry Op1 when Attr > 0.0 is
        begin
```

```
            ...
        end Op1;
        function Op2 return Integer is
        begin
            ...
        end Op2;
        procedure Op3 is
        begin
            ...
        end Op3;
    end Object;

end A;
```

Classes implemented according to the Protected mapping cannot be used in generalization relationships.

Parameterized Classes

There exist two mappings for parameterized classes: either as types declared in generic units, or as types with unconstrained discriminants. Correspondingly, there exist two mappings for bound classes: generic instantiations and constrained types. The mapping is selected by the code generation property ParameterizedImplementation: if this property is set to Generic (the default), the “generic” mapping is used, if it is set to Unconstrained the “unconstrained type” mapping is used.

In all cases, if a parameterized class is associated with a module, the code generation property ParameterizedImplementation must be consistent with the nature of the associated module: if ParameterizedImplementation is Generic, the associated module must be a generic package, if it is Unconstrained it must be a non-generic package.

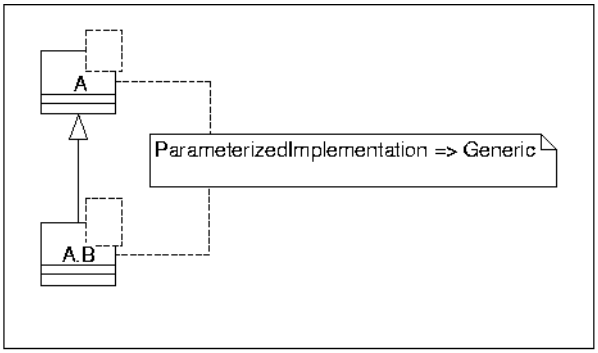
If a class is parameterized, all its subclasses must also be parameterized. The property ParameterizedImplementation of a root class dominates the same property for its subclasses.

Generic Implementation

The root class is transformed into a type declared in a generic library package. The exact nature of the type is controlled by the property TypeImplementation, as for normal classes. The formal part of the generic is extracted from the class specification.

Subclasses are transformed into a tagged type declared in a generic library package, but we have two cases to consider:

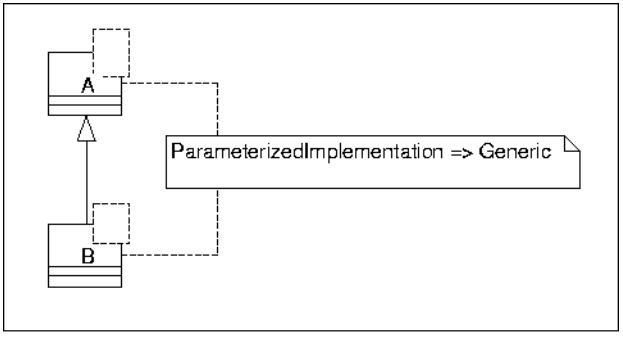
- If the generic library package is a child of the package that contains the superclass, then its formal part only includes the parameters extracted from the class specification of the subclass.




```
generic
    ... -- parameters of the superclass
package A is
    type Object is tagged private;
    ...
end A;
```

```
generic
    ... -- parameters of the subclass
package A.B is
    type Object is new A.Object with private;
    ...
end A.B;
```

- If, on the other hand, the generic library package is not a child of the package that contains the superclass, then it must import the superclass' package as a generic formal package, as shown on the following example:



```
generic
    ... -- parameters of the superclass
package A is
    type Object is tagged private;
    ...
end A;
with A;
generic
    with package Superclass is new A (<>);
    -- parameters of the subclass
package B is
    type Object is new Superclass.Object with private;
    ...
end B;
```

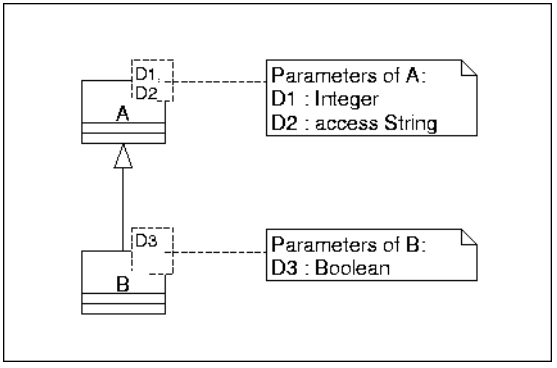
The name of the formal package parameter is given by the property ParentClassName of the subclass, and defaults to “Superclass”.

Unconstrained Type Implementation

The discriminant part of the type is derived from the class parameters. Each class is transformed into a type having unconstrained discriminants (without default values). For a subclass, type derivation is used to add discriminants without constraining the discriminants inherited from the parent type.

If one any of the parameters has a type of the form “access T” then the property IsLimited is dominated, and a limited type is generated for the class.

An example of code generated for the Unconstrained Type implementation is as follows (assuming the default values for other code generation properties):




```
package A is
    type Object (D1 : Integer; D2 : access String) is
                                                tagged limited private;
```

```
    ...
end A;
with A;
package B is
    type Object (D1 : Integer;
                 D2 : access String;
                 D3 : Boolean) is new A.Object (D1, D2)
                                                    with private;
    ...
end B;
```

Bound Classes

If a bound class is associated with a module, that module must be a non-generic package.

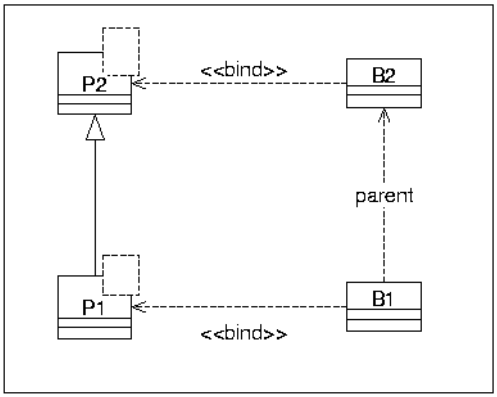
The value of ParameterizedImplementation for a parameterized class (Generic or Unconstrained) determines the mapping chosen for any bound class obtained by binding the parameters of that parameterized class. In other words, the property ParameterizedImplementation of a parameterized class dominates the same property for the bound classes.

Generic Implementation

The class is transformed into a library-level generic instantiation. The actual parameters are extracted from the class specification.

Consider a bound class B1 obtained by binding the parameters of a parameterized class P1. Say that P1 is not a root class, but has instead a superclass P2. Because the actual parameters of B1 only specify values for the parameters of P1, and not of P2, there must exist a bound class B2, obtained by binding the parameters of a parameterized class P2, from which B1 “inherits” the actual parameters for P1.

The UML notation does not allow inheritance relationships between bound classes, because bound classes are fully specified by their template. Therefore, the pseudo-inheritance between B1 and B2 is represented by a dependency relationship labelled “parent”, as shown on the diagram below:



Based on this information, the code is generated in two different ways depending on whether P1 had visibility over its ancestor by a parent-child relationship or by a formal package (see above):

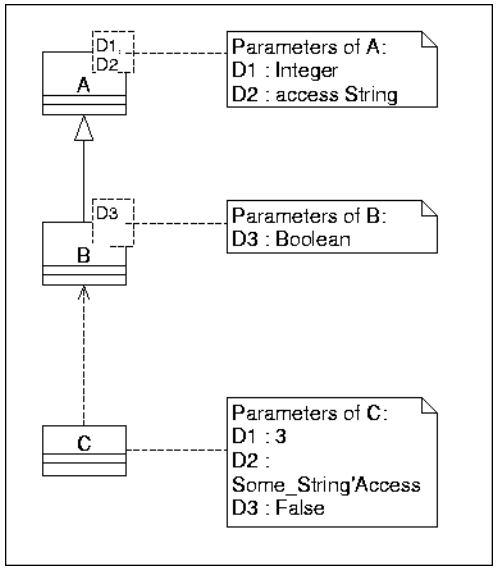
```
package B1 is new B2.P1 (...);
package B1 is new P1 (Superclass => B2, ...);
```

Unconstrained Type Implementation

The class is transformed into a type declaration that provides discriminant constraints. Alternatively, a subtype is generated if the boolean code generation property IsSubtype for the class is True (this property defaults to False).

Each bound class must provide values for all parameters (i.e., constraints for all discriminants), including those inherited from the generalization hierarchy.

An example of code generated for the Unconstrained Type implementation is as follows (assuming the default values for other code generation properties):




```
package C is
    subtype Constrained_Object is
            B.Object
                (D1 => 3,
                 D2 => Some_String'Access,
                 D3 => False);
    type Object is Constrained_Object with private;
    ...
private
    type Object is Constrained_Object with
        record
            ...
        end record;
end C;
```

Utilities

If an utility is associated with a module, that module must be a non-generic package or subprogram. If an utility is not associated with a module, it is transformed into a package. Similarly, parameterized utilities are transformed into generic units, and bound utilities are transformed into library-level instantiations.

If an utility is transformed into a package, no type declaration is produced. Instead, each operation of the utility is transformed into a subprogram in that package. Attributes of such an utility become package-level declarations, regardless of the setting of the “static” button.

If an utility is transformed in a subprogram, then the utility must declare exactly one operation. Note that a bound utility must map to the same kind of program unit as its template.

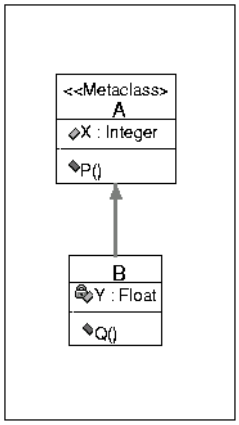
Metaclasses

A metaclass must not have any associated module. The attributes and operations it declares are instead used to generate code for classes that derive from that metaclass.

A metaclass attribute or relationship is transformed into a variable or constant. Depending on the visibility of the attribute or relationship, the variable is declared in the visible part (public), the private part (protected or private) or the body (implementation) of the package associated with each class that derives from the metaclass.

A metaclass operation is transformed into a subprogram, which is declared in the same package as each class which derives from the metaclass. Each parameter (or result) of such a subprogram which had a type name identical to that of the metaclass is transformed into a class-wide parameter. Depending on the visibility of the operation, the subprogram is declared in the visible part (public), the private part (protected or private) or the body (implementation) of the package associated with each class that derives from the metaclass.

An example of code generated for metaclasses is as follows. Note that no module is generated for the metaclass A. Also note the difference between class attributes and operations on one hand, and metaclass attributes and operations on the other hand:




```
package B is

    type Object is tagged private;
    procedure Q (This : Object);

    X : Integer;
    procedure P (This : Object'Class);

private
    type Object is tagged
        record
            Y : Float;
        end record;
end B;
```

Attributes

An attribute is generally transformed into a record component. There exists two special cases for the generation of attributes: the attributes of a metaclass are transformed into package-level declarations, as explained above. The attributes of a normal class which are marked as “static” are also transformed into package level declarations. In fact, in term of code generation, static attributes are handled exactly as attributes of metaclasses.

The record component corresponding to an attribute has a name which is given by the code generation property RecordFieldName.

The code generated for an attribute is controlled by the code generation property RecordFieldImplementation. This property can take the values Discriminant, AccessDiscriminant, and Component (the default). For a parameterized class whose ParameterizedImplementation is Unconstrained, the property RecordFieldImplementation is dominated, and all attributes are implemented as components. If a class has, in its generalization hierarchy, an attribute implemented as an AccessDiscriminant, then the property IsLimited is dominated, and a limited type is generated for that class.

The semantics of RecordFieldImplementation is as follows:

- If RecordFieldImplementation is set to Discriminant, a normal discriminant is generated, as in:

  ```
  type Object (D : Integer := 3) is private;
  ```

- If RecordFieldImplementation is set to AccessDiscriminant, an access discriminant is generated, as in:

  ```
  type Object (D : access Integer) is limited private;
  ```

- If RecordFieldImplementation is set to Component, a normal component is generated in the full type declaration, as in:

  ```
  type Object is
      record
          C : Integer;
      end record;
  ```

All attributes (and “has” relationships; see below) whose RecordFieldImplementation property is either Discriminant or AccessDiscriminant must agree on the existence of default values, and on the visibility: either all have defaults, or none have defaults, and they all have the same visibility. In addition, if the code generation property TypeImplementation of the class is Tagged, then it dominates the property InitialValue, and no default value is generated.

The discriminants always appear in the full type declaration. For private types, whether or not the discriminants appear in the private type declaration depends on their visibility and on the existence of defaults:

- If the discriminants have defaults, they appear in the private type declaration only if their visibility is public. Otherwise, the private type declaration does not include a discriminant part.

- If the discriminants don't have defaults, they appear in the private type declaration only if their visibility is public. Otherwise, the private type declaration includes an unknown discriminant part, as in:

```
package A is
    type Object (<>) is private;
private
    type Object (D : Integer) is
        record ... end record;
end A;
```

The case of a class inheriting discriminants from its superclass (and possibly adding new discriminants) is handled in a manner similar to the Unconstrained Type mapping of parameterized classes.

Has Relationships

"Has" relationships are not part of the UML notation. However, they can be created in Rose using the **View > As Booch** option. When viewed using the UML or OMT notation, they are displayed as unidirectional aggregation relationships. However, they have slightly different code generation properties than true aggregations, because they gather together the properties borne by associations and the properties borne by roles.

An "has" relationship is generally transformed into a record component. There exists two special cases for the generation of "has" relationships: the relationships of a metaclass are transformed into package-level declarations, as explained above. The relationships of a normal class which are marked as "static" are also transformed into package level declarations. In fact, in term of code generation, static "has" relationships are handled exactly as "has" relationships of metaclasses.

In the rest of this discussion, we consider the case of class A having a "has" relationship to class B.

The mapping of an "has" relationship depends on whether it is by-value or by-reference:

- A by-value relationship is represented using the type associated with B (either directly or through some container, depending on the multiplicity of the relationship; see below).
- A by-reference relationship is represented using an access type that designates the type associated with B (either directly or through some container, depending on the multiplicity of the relationship; see below). This access type is only created for

those classes that are the target of some by-reference “has” relationship. There is only one such access type, even if class B is the target of several “has” relationships.

The access type used to represent by-reference relationships targeting B is declared in the package associated with class B. Its name is given by the code generation property AccessTypeName of class B (this property defaults to “Handle”). It is generated either in the public part or in the private part, based on the code generation property AccessTypeVisibility, which can take the values Public (the default) and Private.

If the code generation property AccessTypeDefinition of B is not empty, it dominates, and the declaration of the access type uses this property. Technically, AccessTypeDefinition must contain an Ada type definition. For instance, if AccessTypeDefinition is set to “access constant B.Object” the access type is declared as follows:

```
type Handle is access constant B.Object;
```

If the code generation property AccessTypeDefinition of B is empty (the default), an access type is generated as follows:

- If B is associated with a tagged type, the access type is a class-wide type:

  ```
  type Handle is access B.Object; -- B not tagged
  ```

  ```
  type Handle is access B.Object'Class; -- B tagged
  ```

- If the code generation property MaybeAliased for B is set to True (it defaults to False), the access type is a general access-to-variable type:

  ```
  type Handle is access B.Object'Class;
          -- B tagged, not aliased
  ```

  ```
  type Handle is access all B.Object'Class;
          -- B tagged, may be aliased
  ```

There may be circumstances where it is useful to have an access type declaration generated for class B, even though B is not (or not yet) the target of any by-reference “has” relationship. The code generation property GenerateAccessType controls the generation of an access type. It can take the values Auto and Always. The default is Auto, and corresponds to the case where the generation of the access type depends on the existence of a by-reference “has” relationship. The value Always force the generation of an access type declaration, regardless of the existence of by-reference “has” relationships.

If the maximum allowable cardinality of the relationship is 1, the type of the record component representing the relationship is directly the object or access type associated to B, as explained above.

If, on the other hand, the maximum allowable cardinality of the relationship is larger than 1, an intermediate container type is required to support the one-to-many relationship. The scheme used to generate the code associated with a one-to-many relationship is governed by the code generation properties ContainerImplementation and ContainerType.

If ContainerType is not empty, it dominates ContainerImplementation, and specifies the container type used to represent the one-to-many relationship. The code generation property ContainerDeclarations may be used to specify auxiliary declarations that may be necessary to build the container type.

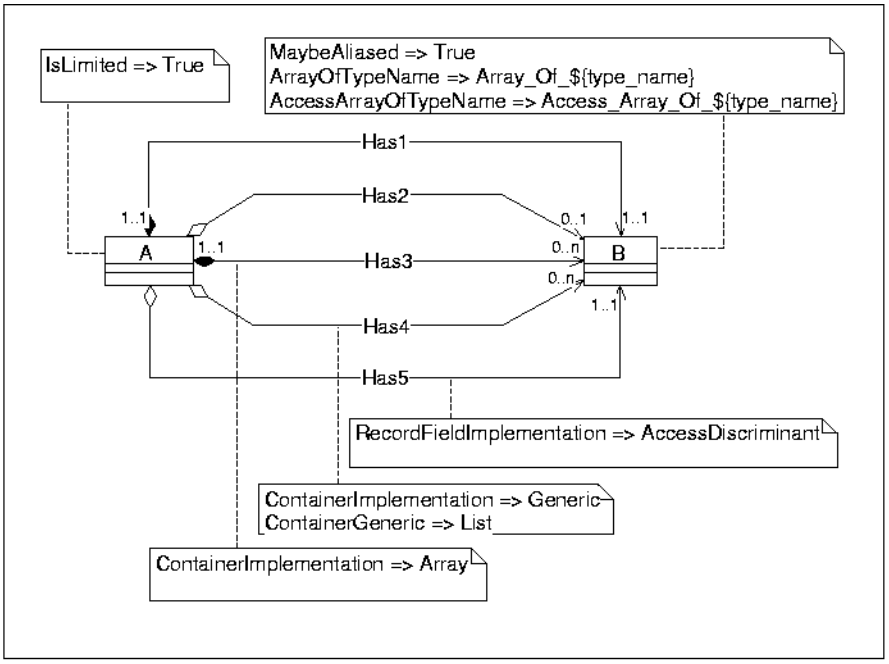
If ContainerType is empty (the default), ContainerImplementation is used to control the code generation scheme. ContainerImplementation can take the two values Generic and Array, and defaults to Array. The semantics of this property is as follows:

- If ContainerImplementation is set to Generic, the generic unit given by the property ContainerGeneric is instantiated, with a single parameter which is the type corresponding to class B, or the access type associated to B, depending on whether the relationship is by-reference or by-value.
- If ContainerImplementation is set to Array, an unconstrained array type, and an access to that array type, are declared to represent the one-to-many relationship. The array element type is either the type associated to B (if the “has” relationship is by-value) or the access type associated with B (if the relationship is by-reference). The name of the array type and access type are given by the code generation properties ArrayOfTypeName (or ArrayOfAccessTypeName) and AccessArrayOfTypeName (or AccessArrayOfAccessTypeName) of class B. These properties default to Array_Of_${type}, Array_Of_${access_type}, Access_Array_Of_${type} and Access_Array_Of_${access_type}, respectively. The index specification for the array types is given by the code generation property ArrayIndexDefinition, which defaults to “Positive range <>”.

The code generation property RecordFieldImplementation which was discussed above in the context of attributes can also be applied to “has” relationships, with the same semantics, except that AccessDiscriminant is not allowed for a by-value relationship.

Note that the target of a “has” relationship must not be a class whose TypeImplementation property is Mixin.

As an illustration of the implementation of “has” relationship, consider the following class diagram:

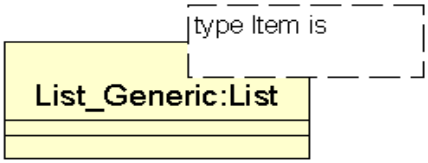


It results into the following code (note that only the “get” accessors are shown; the “set” accessors have similar parameter types):

```
with B;
 with List_Generic;
 package A is
     type Object (Has5 : access B.Object) is tagged limited private;

     package B_List is new List_Generic (B.Object);

     function Get_Has1 (This : in Object) return B.Object;
     function Get_Has2 (This : in Object) return B.Handle;
     function Get_Has3 (This : in Object) return B.Array_Of_Object;
     function Get_Has4 (This : in Object) return B_List.List;

     -- "set" accessors go here

 private
     type Object (Has5 : access B.Object) is tagged limited
         record
             Has1 : B.Object;
             Has2 : B.Handle;
             Has3 : B.Access_Array_Of_Object;
             Has4 : B_List.List;
```

```
        end record;
end A;

package B is
    type Object is tagged private;
    type Handle is access all Object'Class;
    type Array_Of_Object is
       array (Positive range <>) of Object;
    type Access_Array_Of_Object is
       access Array_Of_Object;
private
       ...
end B;
```

The following defines the generic container used by Has4:




```
generic
    type Item is private;
package List_Generic is
    type List is tagged private;
private
    ...
end List_Generic;
```

Associations

Associations fall into two categories:

- Simple Associations
- Association Classes

The generated code for both categories follows a number of common principles.

Code is only generated for the roles which are marked as *navigable* in the Rose model. If an association has no navigable role, no code is generated for that association.

Code is only generated if the two classes that participate in the association have their Type Implementation property set to Record or Tagged. An error is emitted if an association involves classes with a non-record, non-tagged implementation.

There exist many similarities between the mapping of associations and that of “has” relationships:

- A role always becomes a component in a record or tagged type.
- The name of a role determines the name of the various declarations generated for that role (record component, accessor subprograms, etc.). If a role is unnamed, the name of the class at the other end of the association is used to determine the name of the declarations generated for that role.
- If a class is the target of a navigable by-reference role, an access type is generated for that class. The characteristics of that access type depend on the code generation properties AccessTypeName, AccessTypeVisibility, AccessTypeDefinition and MaybeAliased of the class.
- The mapping of a role depends on its multiplicity. If the maximum allowable cardinality is larger than 1, a container type is declared, as specified by the code generation properties ContainerImplementation, ContainerType, ContainerGeneric and ContainerDeclarations for the role.
- The code generation properties NameIfUnlabeled, RecordFieldName, GenerateGet, GetName and InlineGet may also be applied to a role, with a semantic similar to the semantics they have for “has” relationships. The code generation properties GenerateSet, SetName and InlineSet are only used when the role belongs to a unidirectional association, i.e., an association with only one navigable role. They are not used when the role belongs to a bidirectional association, i.e., an association with two navigable roles.

Simple Associations

If a simple association has only one navigable role, the code generated for that association is *exactly identical* to the code that would be generated for an “has” relationship similar to that role. Such an association may be marked “static”, in which case package-level declarations are generated instead of record components (again, this is identical to the case of an “has” relationship).

A warning is emitted by the code generator when it encounters a unidirectional association, because an association normally has two navigable roles (and thus the presence of only one navigable role may indicate a mistake).

The rest of this section pertains only to the case of a simple association with two navigable roles.

The two classes which participate in the association must map to the same package, either because their names use the colon notation and have the same prefix, or because they are associated with the same module (a package specification).

In both classes the TypeVisibility property must be set to Private.

An association may have *keys* which are used to unambiguously identify an object. Keys are handled by Rose / Ada exactly as attributes of classes: they are normally generated as record components, possibly with “get” and “set” accessors. If several associations originating from the same class declare keys with the same name, the record component is only generated once. An error is detected in this case if the various keys don't have the same type.

A bidirectional association may not be marked “static”.

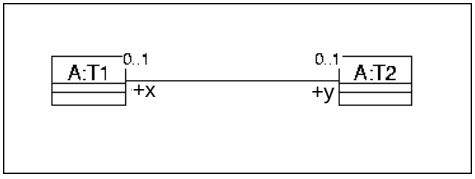
Data Structures

If any role of a bidirectional association is by-value, an error is detected.

If both roles of a bidirectional association are by-reference, the data structures (record, components, discriminants, etc.) generated for the association are exactly identical to the data structure that would be generated for two by-reference “has” relationships. These data structures depend on the multiplicity of the association. They are shown below, assuming that both classes use the Tagged implementation, and that arrays are used to represent relationships with maximum allowable cardinality larger than 1.

In the following examples, the AccessTypeName class property must be given a unique name since both classes map to the same package.

- For a one-to-one association, the generated data structures are as follows:




```
package A is

    type T1 is tagged private;
    type H1 is access T1'Class;

    type T2 is tagged private;
    type H2 is access T2'Class;

    -- Operations go here

private

    type T1 is tagged
```

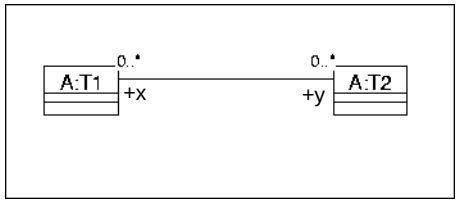
```
        record
            -- Keys and attributes go here
            Y : H2;
        end record;

    type T2 is tagged
        record
            -- Keys and attributes go here
            X : H1;
        end record;

end A;
```

- For a one-to-many association, the generated data structures are as follows:


```
package A is

    type T1 is tagged private;
    type H1 is access T1'Class;

    type Array_Of_H1 is
                  array (Positive range <>) of H1;
    type Access_Array_Of_H1 is access Array_Of_H1;

    type T2 is tagged private;
    type H2 is access T2'Class;

    -- Operations go here

private

    type T1 is tagged
        record
            -- Keys and attributes go here
            Y : H2;
        end record;

    type T2 is tagged
        record
            -- Keys and attributes go here
```

```
            X : Access_Array_Of_H1;
        end record;

end A;
```

- For a many-to-many association, the generated data structures are as follows:




```
package A is

    type T1 is tagged private;
    type H1 is access T1'Class;

    type Array_Of_H1 is
                 array (Positive range <>) of H1;
    type Access_Array_Of_H1 is access Array_Of_H1;

    type T2 is tagged private;
    type H2 is access T2'Class;

    type Array_Of_H2 is
                 array (Positive range <>) of H2;
    type Access_Array_Of_H2 is access Array_Of_H2;

    -- Operations go here

private

    type T1 is tagged
        record
            -- Keys and attributes go here
            Y : Access_Array_Of_H2;
        end record;

    type T2 is tagged
        record
            -- Keys and attributes go here
            X : Access_Array_Of_H1;
```

```
         end record;

end A;
```

Subprograms

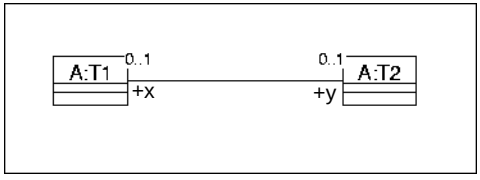
A “get” accessor may be generated for each role in the association, based on the code generation properties GenerateGet, GetName and InlineGet of the role.

Bidirectional associations must be created and deleted using the subprograms Associate and Dissociate as explained below. This is for integrity reasons: if two objects are linked by a bidirectional association, it is important that each of them has a pointer to the other. If “set” accessors were generated in that case, they could be used to create a situation where object A has a pointer to object B, but object B doesn't have a pointer to object A. Such a situation doesn't correspond to an association, but to two aggregation relationships. By generating Associate and Dissociate subprograms instead of “set” accessors for bidirectional associations, Rose / Ada prevents such violations of the association model.

Two families of subprograms, named Associate and Dissociate by default, may be generated for each role, under the control of the code generation properties GenerateAssociate and GenerateDissociate of the association. These subprograms are used to establish or break an association by establishing or breaking linkages between objects. The profiles of these subprograms depend on the multiplicities of both roles, and on the nature of the construct used to implement relationships with maximum allowable cardinality larger than 1. The code shown below corresponds to the case where the ContainerImplementation property of the roles is Array. If the ContainerImplementation is Generic, or if a container type is provided, the name of the container type is substituted to the name of the array type in the subprogram declarations.

Alternate names may be provided for the Associate and Dissociate subprograms using the code generation properties AssociateName and DissociateName. The code generation properties InlineAssociate and InlineDissociate control whether or not a pragma Inline is emitted for these subprograms.

- For a one-to-one association, the generated subprograms are as follows:



```
procedure Associate
              (This_H2 : in H2; This_H1 : in H1);

procedure Dissociate (This_H2 : in H2);
procedure Dissociate (This_H1 : in H1);
```

The semantics of Associate is that it establishes a two-way linkage between the given objects. If the given objects are already part of an association, this association is not broken, but instead Associate raises the exception System.Assertion_Error.

The semantics of Dissociate is that it breaks the linkage between the given object and its correspondent (if any). Dissociate may be used for either extremity of the association: that's why there are two overloaded declarations, one taking an H1, the other taking an H2.

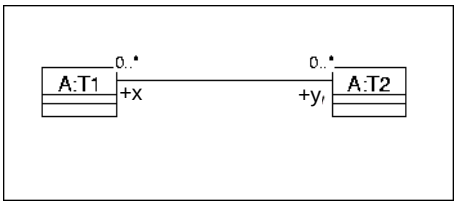
- For a one-to-many association, the following Associate and Dissociate procedures are generated *in addition* to the ones described above for one-to-one associations:


```
procedure Associate (This_H2 : in H2;
               This_Array_Of_H1 : in Array_Of_H1);

procedure Dissociate (This : in Array_Of_H1);
```

The semantics of Associate is that it establishes two-way linkages between the object designated by This_H2 and each of the objects designated by the pointers in This_Array_Of_H1. These linkages are *added* to those that might already exist between the object designated by This_H2 and other objects of type T1. If some of the objects designated by the pointers in This_Array_Of_H1 are already part of an association, the exception System.Assertion_Error is raised.

The semantics of Dissociate is that it breaks the linkages between each object designated by the pointers in This_Array_Of_H1 and the associated object of type T2.

- For a many-to-many association, the following Associate and Dissociate procedures are generated *in addition* to the ones described above for one-to-one and one-to-many associations:




```
procedure Associate
                (This_Array_Of_H2 : in Array_Of_H2;
                     This_H1 : in H1);
procedure Associate
                (This_Array_Of_H2 : in Array_Of_H2;
                This_Array_Of_H1 : in Array_Of_H1);

procedure Dissociate (This : in Array_Of_H2);
```

  The semantics of Associate is that it establishes two-way linkages between the object designated by This_H1 (or by the pointers in This_Array_Of_H1) and each of the objects designated by the pointers in This_Array_Of_H2. These linkages are *added* to those that might already exist between the designated objects designated by This_H2 and other objects of type T1. Note that the exception System.Assertion_Error is never raised by Associate for a many-to-many association (notwithstanding what was said above for one-to-one and one-to-many associations).

  The semantics of Dissociate is that it breaks the linkages between each object designated by the pointers in This_Array_Of_H2 and the associated objects of type T1.

- For an association having a finite multiplicity (e.g. 1..4), the subprograms profiles and semantics are similar to those corresponding to the unlimited case (e.g. one-to-many), except that the Associate subprogram check the multiplicity constraint (e.g. it is not possible to associate more than 4 objects of type T1 to an object of type T2). The exception System.Assertion_Error is raised if this check fails.

Note that for associations having a role whose maximum allowable cardinality is 1, Associate never *replaces* the current association, if it turns out that the object on that role is already part of some association. Instead, the exception System.Assertion_Error is raised. On the other hand, for a role whose maximum cardinality is unlimited, it is always possible to *augment* the current association, so no exception is ever raised.

If replacement is needed for an association, it may be implemented by successively calling Dissociate and Associate.

If the Association and Dissociate subprograms are passed null pointers, they raise System.Assertion_Error. However, for the versions of these subprograms which take arrays of access values, it is acceptable for the arrays to contain null pointers: these null pointers are simply skipped. Still, the entire array must contain at least one non-null pointer.

For one-to-one associations, and for one-to-many or many-to-many associations with ContainerImplementation properties set to Array, the bodies of the Associate and Dissociate procedures are entirely generated by Rose/Ada, with the semantics explained above. They perform storage management by reusing empty slots in the arrays, allocating longer arrays if needed, and reclaiming storage when appropriate. They also preserve the integrity of the association by detecting the case where two of the access passed to Associate denote the same object. Because the generated code is part of a protected region, it can be modified by the user to meet special needs. It is however recommended that the above semantics be adhered to.

For one-to-many or many-to-many associations with a specific ContainerType, or with a ContainerImplementation set to Generic, the bodies of the Associate and Dissociate procedures are left empty.

Association Classes

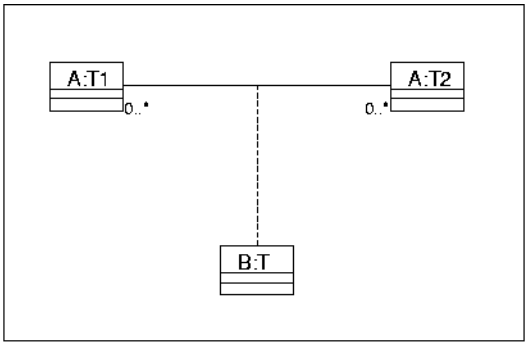
For an association class, independent objects must be created to hold the attributes of the association. Therefore, a type is generated which corresponds to the association class. This type may be generated in any package: it doesn't have to be located in the same package which contains the two principal classes involved in the association.

Data Structures

The generated data structures are similar to what would be generated if the association class had a one-to-many association with each of the two principal classes. However, these data structures are essentially hidden, and the clients are only given operations to query, create or delete the association, and operations to read or modify the attributes of the association. This ensures that the integrity of the association is preserved.

The data structures are such that, from each end of the association, it is possible to find a list of auxiliary records. Each of these auxiliary records contains a value of the association class, and two pointers to both ends of the association. So it is possible to traverse from one end of the association to the other through the auxiliary record. The auxiliary record and the associated type declaration are not exported, to preserve the integrity of the association.

The generated data structures for a many-to-many association class are as follows:




```
package B is
    type T is tagged private;

    -- Operations go here

private
    type T is tagged
        record
            -- Attributes go here
        end record;
end B;

with B;
package A is

    type T1 is tagged private;
    type H1 is access T1'Class;
    type Array_Of_H1 is array (Positive range <>) of H1;
    type Access_Array_Of_H1 is access Array_Of_H1;

    type T2 is tagged private;
    type H2 is access T2'Class;
    type Array_Of_H2 is array (Positive range <>) of H2;
    type Access_Array_Of_H2 is access Array_Of_H2;


-- Operations go here

private

    type Attribute_B is
        record
            Attribute : B.T;
            The_T1 : H1;
            The_T2 : H2;
        end record;
```

```
    type Access_Attribute_B is access Attribute_B;

    type Array_Of_Access_Attribute_B is
            array (Positive range <>) of Access_Attribute_B;
    type Access_Array_Of_Access_Attribute_B is
            access Array_Of_Access_Attribute_B;

    type T1 is tagged
        record
            -- Keys and attributes go here
            The_B : Access_Array_Of_Access_Attribute_B;
        end record;

    type T2 is tagged
        record
            -- Keys and attributes go here
            The_B : Access_Array_Of_Access_Attribute_B;
        end record;

end A;
```

Similar code would be generated in the one-to-one and one-to-many cases.

Subprograms

Associate and Dissociate procedures are generated for the entire association. These procedures are similar to those corresponding to a simple association, except for that only one Associate procedure is generated, regardless of the multiplicity. That's because it is mandatory to specify, when establishing an association, the value of the association class. The variants of the Associate subprogram that would take array of accesses for the principal classes would also have to take array of values for the association class. This interface would be complex and difficult to use, so it is not supported by Rose / Ada.

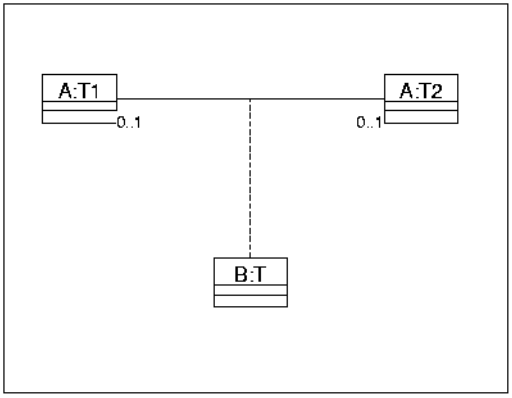
Two accessor subprograms are generated to read and modify the value of the attributes of the association class. In order to determine the association to modify, these subprograms take:

- One access value designating an object on the cardinality 1 role of the association, for one-to-one and one-to-many associations.
- Two access values, designating objects of the two principal classes, for many-to-many associations.

That information makes it possible to unambiguously locate the association whose attributes must be read or modified. The generation of the “get” accessor is controlled by the properties GenerateGet, GetName and InlineGet of the association. Similarly the generation of the “set” accessor is controlled by the properties GenerateSet, SetName and InlineSet of the association.

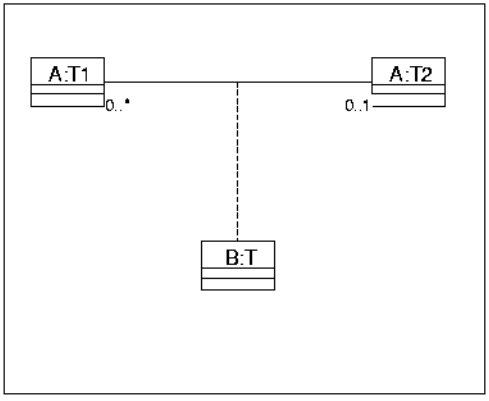
The generated subprograms for an association class are shown below (we omit the Dissociate procedures which are exactly identical to those generated for simple associations):

- For one-to-one association classes, the generated subprograms are as follows:



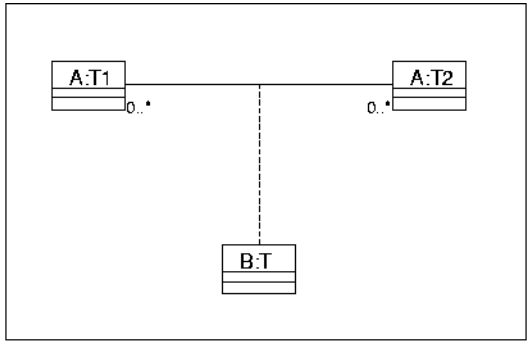

```
procedure Associate (This_H1 : in H1;
                             This_H2 : in H;
                             This_T : in B.T);

function Get_T (This_H1 : in H1) return B.T;
function Get_T (This_H2 : in H2) return B.T;

procedure Set_T (This_H1 : in H1; This_T : in B.T);
procedure Set_T (This_H2 : in H2; This_T : in B.T);
```

- For one-to-many association classes, the generated subprograms are as follows:



```
procedure Associate (This_H1 : in H1;
                     This_H2 : in H;
                     This_T : in B.T);

function Get_T (This_H1 : in H1) return B.T;

procedure Set_T (This_H1 : in H1; This_T : in B.T);
```

- For many-to-many association classes, the generated subprograms are as follows:




```
procedure Associate (This_H1 : in H1;
                     This_H2 : in H;
                     This_T : in B.T);

function Get_T (This_H1 : in H1;
                     This_H2 : in H2)
                     return B.T;

function Get_T (This_H2 : in H2) return B.T;

procedure Set_T (This_H1 : in H1;
                     This_H2 : in H2;
                     This_T : in B.T);
```

As in the case of simple associations, Rose/Ada generates a full implementation for these subprograms if the roles with maximum allowable cardinality larger than 1 are represented by arrays. It generates a [statement] prompt otherwise. This implementation checks the consistency of the operations, and raises System.Assertions_Error if inconsistencies are detected. It also performs storage management, allocating and reclaiming the arrays and auxiliary records as appropriate.

Dependency Relationships

A dependency relationship between two classes is transformed in a *with* clause between the corresponding library units, unless of course both classes happen to map to types in the same library unit. Note that in addition to dependency relationships, *with* clauses are also generated from the module dependencies appearing in the component diagrams.

Generalization Relationships (Inheritance)

To some extend, the generalization relationship has already been discussed in the section about classes above.

The visibility of a generalization relationship is used to determine how the type derivation is declared. If the relationship is public, the derivation occurs in the visible part, with a private extension:

```
package Subclass is
    type Object is new Superclass.Object with private;
private
    type Object is new Superclass.Object with
        record ... end record;
end Subclass;
```

If the relationship is not public, the derivation occurs in the private part:

```
package Subclass is
    type Object is tagged private;
private
    type Object is new Superclass.Object with
        record ... end record;
end Subclass;
```

If the class Subclass has its code generation property TypeVisibility set to Public, then regardless of the visibility of the relationship, the code is simply:

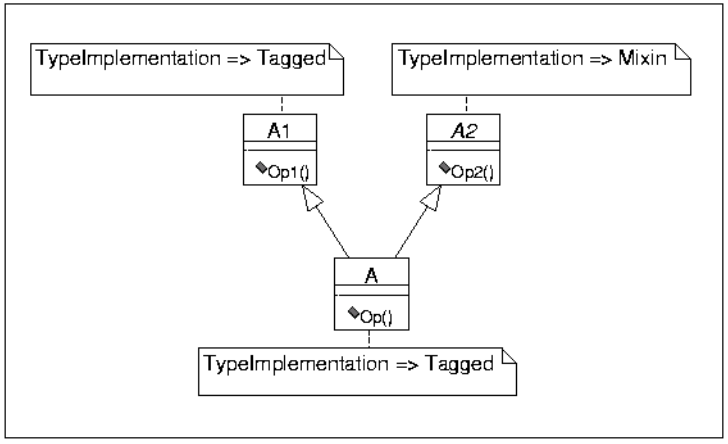
```
package Subclass is
    type Object is new Superclass.Object with
        record ... end record;
end Subclass;
```

The case of multiple inheritance is more complex. If a class A has more than one superclass, there are two ways that this relationship can be represented in Ada 95: “mixin” inheritance or “multiple views” inheritance. The code generation properties TypeImplementation of the superclasses of A determine what mapping is used.

Mixin Inheritance

In mixin inheritance, exactly one of the superclasses of A must have its code generation property TypeImplementation set to Tagged. This superclass defines the "main" line of inheritance (or generalization). All other superclasses must have their code generation property TypeImplementation set to Mixin.

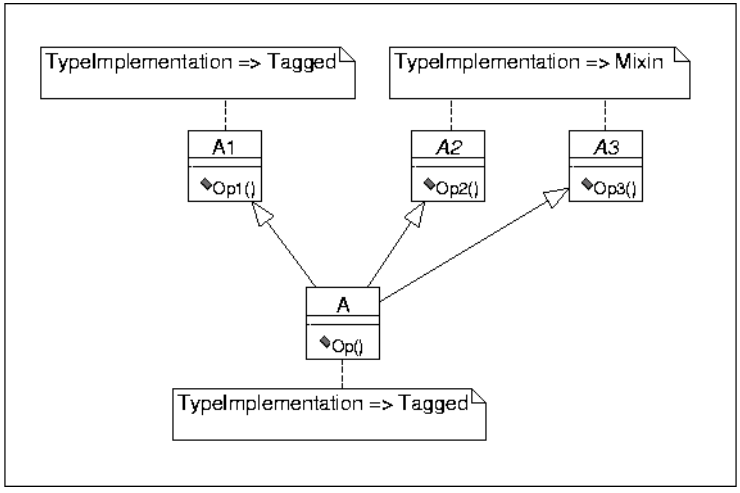
The type representing A is declared by deriving from its main superclass, and instantiating the generic packages associated with the mixin superclasses to add more primitive operations to the resulting type. Assume that the main superclass is called A1 and the mixin superclass A2. The generated code is as follows, assuming that A1 and A2 each declare an operation (we use the defaults for those code generation properties that have no direct bearing on multiple inheritance):




```
package A1 is
    type Object is tagged private;
    procedure Op1 (This : Object);
private
    type Object is tagged
        record ... end record;
end A1;

generic
    type Superclass is abstract tagged private;
package A2 is
    type Object is abstract new Superclass with private;
    procedure Op2 (This : Object);
private
    type Object is abstract new Superclass with
        record ... end record;
end A2;

with A1;
with A2;
```

```
package A is
    package A2_Instantiation is
        new A2 (Superclass => A1.Object);
    type Object is new A2_Instantiation.Object with
                                                private;
    procedure Op (This : Object);
private
    type Object is new A2_Instantiation.Object with
        record ... end record;
end A;
```

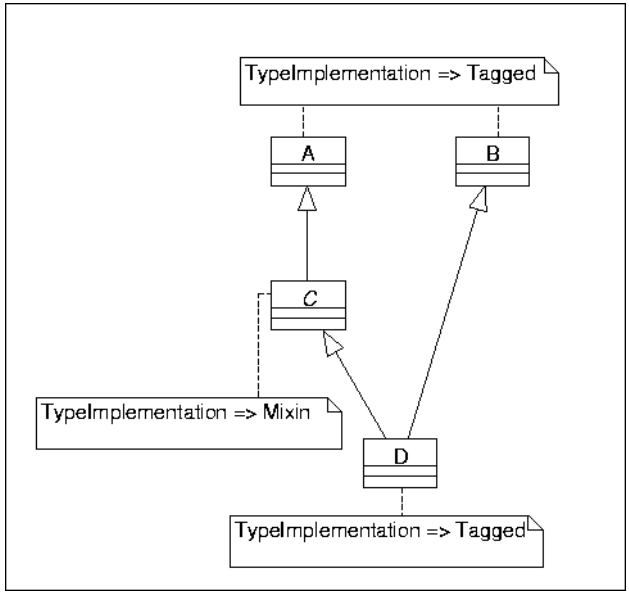
The case of triple inheritance and beyond is handled similarly, with more instantiations adding more primitive operations. Assuming that we add a mixin superclass, A3, to the above example, we obtain the following code (A1 and A2 are unchanged):




```
generic
    type Superclass is abstract tagged private;
package A3 is
    type Object is abstract new Superclass with private;
    procedure Op3 (This : Object);
private
    type Object is abstract new Superclass with
        record ... end record;
end A3;

with A1;
with A2;
with A3;
package A is
    package A2_Instantiation is
        new A2 (Superclass => A1.Object);
    package A3_Instantiation is
        new A3 (Superclass => A2_Instantiation.Object);
```

```
    type Object is
        new A3_Instantiation.Object with private;
    procedure Op (This : Object);
private
    type Object is new A3_Instantiation.Object
        with record ... end record;
end A;
```

Note a constraint on mixin inheritance: if any of the mixins has a superclass, it is necessary for the “main” superclass to be a specialization of the same class (otherwise the instantiation would be illegal). This means that the following diagram is illegal because B is not identical to A and is not a subclass of A:



In the case of triple inheritance and beyond, this rule becomes slightly more complicated: all the mixins must either have no superclass, or have the same superclass, and the main class must be identical to this common superclass, or inherit from it.

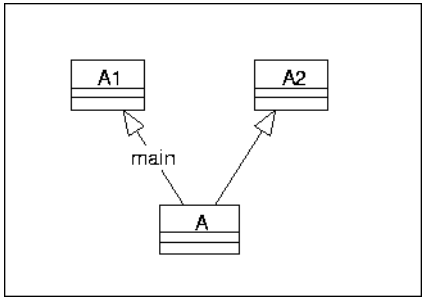
Multiple Views Inheritance

In multiple views inheritance, all the superclasses of A must have their code generation property TypeImplementation set to Tagged. In addition, one of the inheritance (or generalization) relationships must be identified as the main line of descent by giving it the name “main”.

There are a number of restrictions on multiple views inheritance. First, all superclasses must be limited, by setting their code generation property IsLimited to True (or because IsLimited is dominated by another property which forces limited-ness). Second, the main inheritance relationship cannot be “less visible” than the auxiliary relationships. For instance, it is not possible to have a private main inheritance and a public auxiliary inheritance. On the other hand, it is possible to have only private inheritance, or to have a public main inheritance, a public auxiliary inheritance, and another, private, auxiliary inheritance.

All the operations of the superclasses are inherited, and default bodies are generated if necessary. If two operations coming from different superclasses would result in homograph declarations for the class A, the operation coming from the main line of inheritance has precedence.

Assuming that the main superclass is called A1 and the auxiliary superclass is called A2, the following code is generated (again, we use the defaults for those code generation properties that have no direct bearing on multiple inheritance):



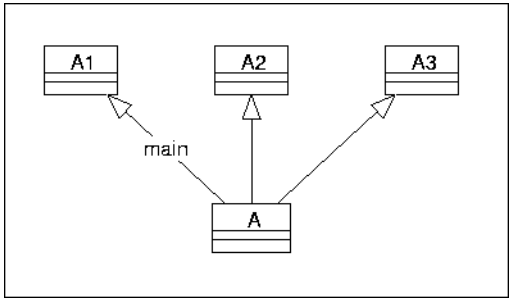

```
package A1 is
    type Object is tagged limited private;
    procedure Op1 (This : Object);
private
    type Object is tagged limited
        record ... end record;
end A1;

package A2 is
    type Object is tagged limited private;
    procedure Op2 (This : Object);
private
    type Object is tagged limited
        record ... end record;
end A2;
with A1;
with A2;
package A is
    type Views;
```

```
    type A2_With_Back_Pointer
        (Back : access Views'Class) is
        new A2.Object with null record;

    type Views is abstract new A1.Object with
        record
            A2_View : A2_With_Back_Pointer (Views'Access);
        end record;

    type Object is new Views with private;
    procedure Op (This : Object);
    procedure Op2 (This : Object);
private
    type Object is new Views with
        record ... end record;
end A;
```

The body of subprogram Op2 is generated as follows, in order to call the corresponding subprogram of the superclass:

```
procedure Op2 (This : Object) is
begin
    A2.Op2 (A2.Object (This.A2_View));
end Op2;
```

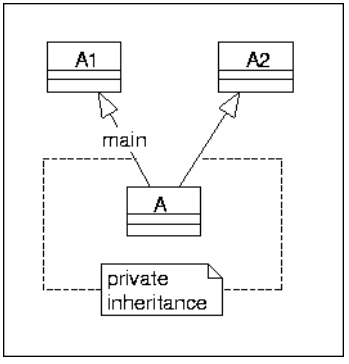
The same scheme extends to triple inheritance and beyond. If we add superclass A3, we obtain:




```
package A3 is
    type Object is tagged limited private;
    procedure Op3 (This : Object);
private
    type Object is tagged limited
        record ... end record;
end A3;

with A1;
with A2;
with A3;
package A is
    type Views;
```

```
    type A2_With_Back_Pointer
        (Back : access Views'Class) is
        new A2.Object with null record;
    type A3_With_Back_Pointer
        (Back : access Views'Class) is
        new A3.Object with null record;

    type Views is abstract new A1.Object with
        record
            A2_View : A2_With_Back_Pointer (Views'Access);
            A3_View : A3_With_Back_Pointer (Views'Access);
        end record;

    type Object is new Views with private;
    procedure Op (This : Object);
    procedure Op2 (This : Object);
    procedure Op3 (This : Object);
private
    type Object is new Views
        with record ... end record;
end A;
```

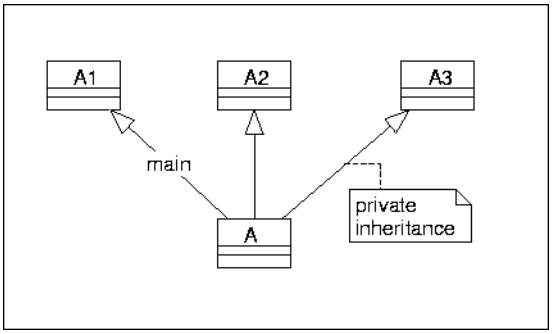
The interaction with the visibility of inheritance relationships is worth expressing in detail. In the first case, if the inheritances from A1 and A2 are changed to be private (or protected), we don't need the intermediate type Views anymore, and the code generated for A becomes:




```
with A1;
with A2;
package A is
    type Object is tagged limited private;
    procedure Op (This : Object);
private
    type A2_With_Back_Pointer
        (Back : access Object'Class) is
        new A2.Object with null record;

    type Object is new A1.Object with
```

```
        record
            A2_View : A2_With_Back_Pointer (Object'Access);
            ...
        end record;
    procedure Op2 (This : Object);
end A;
```

In the case of triple inheritance, if the visibility of the inheritance from A3 is changed to private (or protected) the generated code for A becomes:




```
with A1; with A2; with A3;
package A is
    type Views;

    type A2_With_Back_Pointer
        (Back : access Views'Class) is
        new A2.Object with null record;

    type Views is abstract new A1.Object with
        record
            A2_View : A2_With_Back_Pointer (Views'Access);
        end record;

    type Object is new Views with private;
    procedure Op (This : Object);
    procedure Op2 (This : Object);
private
    type A3_With_Back_Pointer
        (Back : access Object'Class) is
        new A3.Object with null record;
    type Object is new Views with
        record
            A3_View : A3_With_Back_Pointer (Object'Access);
            ...
        end record;
    procedure Op3 (This : Object);
end A;
```

Operations

The operations given in a class specification are simply copied in the generated code.

If the code generation properties ImplicitParameter of the project and of the class are both True, a first parameter may be added to the profile of each operation. The type of this parameter is the type associated with the given class, its mode is given by the code generation property ImplicitParameterMode of the operation, and its name is given by the code generation property ImplicitParameterName of the class. These properties default to In and “This”, respectively.

The code generation property ImplicitParameter at the project level defaults to False. The code generation property ImplicitParameter of the class defaults to True. By having two code generation properties, one at the project level and one at the class level, Rose / Ada supports the following usage patterns:

- The default is to never add this first parameter.
- By setting the code generation property ImplicitParameter to True at the project level, a user may decide to add the first parameter for all classes in the project.
- If some classes must be handled specially, and no first parameter is required for them, the code generation property ImplicitParameter of these classes may be set to False.

The code generation property ImplicitParameterMode can take the values In, InOut and Out. There are also circumstances in which it is useful to generate a subprogram taking an access parameter in addition (or instead of) the subprogram taking an object parameter. The code generation property GenerateAccessOperation controls whether a subprogram taking an access parameter is generated. This property is only used if ImplicitParameter is True.

Accessor Operations

Each attribute, “has” relationship, and association role has two code generation properties, GenerateGet and GenerateSet, which control generation of accessor operations for this attribute or relationship. These properties default to False.

- The “get” accessor is used to read the corresponding attribute or relationship. It is a function taking an object of the class and returning the type of the attribute.
- The “set” accessor is used to update the corresponding attribute or relationship. It is a procedure taking as *in out* parameter an object of the class, and a value of the type of the attribute.

For attributes and “has” relationships which are translated into discriminants, the “set” accessor doesn't make sense, and is therefore not generated (in other words, GenerateSet is dominated by RecordFieldImplementation). The “get” accessor is not generated either, because a discriminant is directly visible to clients, even for a private type: GenerateGet is also dominated by RecordFieldImplementation in this case.

In addition to (or instead of) the “get” and “set” accessors which take object parameters, Rose / Ada can also generate accessors which take access parameters. This is controlled by the code generation properties GenerateAccessGet and GenerateAccessSet.

The boolean code generation properties InlineGet and InlineSet of the attribute, relationship or role control whether a pragma Inline is generated for the accessor operations. These properties default to True.

Standard Operations

Standard operations, not explicitly present in the model, may be generated if the code generation property GenerateStandardOperations of the project is set to True (it defaults to False):

- A constructor is generated if the code generation property GenerateDefaultConstructor is not DoNotCreate (this property may take the values Function, Procedure and DoNotCreate; the default is Function). The name of the constructor is given by DefaultConstructorName (this property defaults to “Create”).
- A copy constructor is generated if the code generation property GenerateCopyConstructor is not DoNotCreate (this property may take the values Function, Procedure and DoNotCreate; the default is Function). The name of the constructor is given by CopyConstructorName (this property defaults to “Copy”).
- A destructor is generated if the code generation property GenerateDestructor is not DoNotCreate (this property may take the values Procedure and DoNotCreate; the default is Procedure). The name of the destructor is given by DestructorName (this property defaults to “Free”).
- An equality operator is generated if the code generation property GenerateTypeEquality is not DoNotCreate (this property may take the values Function and DoNotCreate; the default is DoNotCreate). The name of the operator is given by TypeEqualityName (this property defaults to “${quote}=${quote}”).

If an access type is generated for the class (in addition to the true object type), and the class is not abstract, then the above properties also control generation of the subprograms pertaining to this access type. For instance, if GenerateCopyConstructor is set to Function, and CopyConstructorName is set to “Copy”, two Copy functions

are generated: one for the object type, and one for the associated access type. This rule only applies to the subprograms described in this section: it doesn't apply to “get” and “set” accessors, or to user-defined subprograms.

On an abstract class, the above subprograms, if generated, are made abstract.

Note that making the constructors functions (as opposed to procedures) on classes which map to limited types may lead to difficulties, and is not recommended (although it may make sense in some circumstances).

The boolean code generation properties InlineDefaultConstructor, InlineDestructor, InlineCopyConstructor and InlineEquality of the class control whether a pragma Inline is generated for the above operations. All these properties default to False.

Subprogram Implementation

The code generation property SubprogramImplementation is used to control the code generated for a subprogram body. This property can take the values Body, Renaming, Separate, Abstract and Spec. The default is Body. The semantics of these choices are as follows:

- If SubprogramImplementation is set to Body, a normal body is generated.
- If SubprogramImplementation is set to Renaming, a renaming-as-body is generated for the subprogram body. The name of the renamed subprogram is obtained from the property Renames of the operation.
- If SubprogramImplementation is set to Separate, a stub is generated instead of a normal body.
- If SubprogramImplementation is set to Abstract, no body is generated, instead the specification of the subprogram includes the reserved words “is abstract” (making it an abstract subprogram). It is an error to set SubprogramImplementation to Abstract on an operation of a non-abstract class.
- If SubprogramImplementation is set to Spec, no body is generated, but the subprogram is not made abstract. This option (which doesn't result in legal code) is intended to be complemented by the insertion, in some protected region of the generated code, of a pragma (like Import or Interface) which specifies the implementation of the subprogram without providing an explicit body.

In addition, the code generation property Inline is used to control whether or not a pragma Inline is generated for the operation. This property defaults to False.

Visibility

The visibility of each operation determines where it is declared. A public operation is declared in the visible part of the associated package, a protected or private operation is declared in the private part of the package, and an operation with only implementation visibility is declared in the package body (note that such an operation is not inherited).

Overriding

The code generator takes care to generate the proper overriding subprogram declarations whenever the language requires it:

- If an abstract operation is inherited by a concrete class. This includes the case where the concrete class has several superclasses, either because of mixin inheritance, or because of multiple views inheritance.
- If a function returning a value of the superclass is inherited by a concrete class. The language rules state that such a function becomes abstract by derivation.
- If one of the “back pointer” types generated for multiple views inheritance inherits an abstract operation. That's because the “back pointer” types are always concrete.

In addition to these cases where overriding is required by the language, the code generator also generates an overriding declaration if the inherited operation has it code generation property GenerateOverriding set to True. This property defaults to True.

Each overriding subprogram declaration has the same parameter names, modes and default values as that of the original subprogram. The proper type name is substituted for each controlling operand. The types of other operands are left unchanged.

Rose / Ada generates a body for each overriding subprogram declaration. This body does a view conversion of its controlling parameters, and calls the corresponding operation of the parent type (or superclass). While this implementation in itself is not extremely useful, it turns out that most overridden subprograms first call the operation of their parent type, and then perform additional processing specific to the added record components. By generating the call to the superclass’ operation, Rose / Ada makes it easy to adhere to this model. (This is similar to sending a message to *super* in languages like Smalltalk or Java.)

Note that there is not property GenerateOverriding for the “get” and “set” accessor. That's because most of the time the inherited implementation is appropriate. Therefore, no overriding declaration is ever generated for these accessors.

Bodies

Except for the accessor operations, the body generated for an operation contains only a [statement] prompt. This ensures that the code can be compiled under Rational Apex, but that any attempt to execute an operation whose body is incomplete raises Program_Error. Note that, if using another compiler, the prompt is likely to result in syntax errors: legal code must be written to replace these dummy bodies before the code can be compiled.

The code generation properties EntryCode and ExitCode associated with an operation contain Ada statements which are copied verbatim at the beginning and at the end, respectively, of the statement part of the generated body. These properties are empty by default.

User-Defined Initialization, Assignment and Finalization

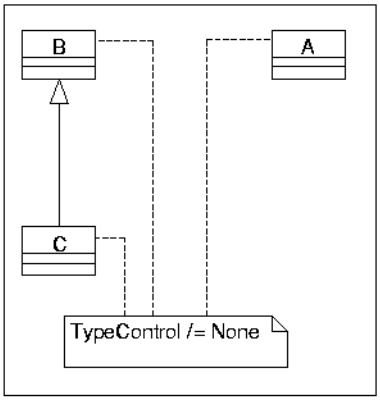
Controlled types may be produced for any type whose TypeImplementation is Tagged. In addition to producing the proper type structure, Rose/Ada is also capable of generating overriding declarations for the procedures Initialize, Adjust and Finalize, and for the operator "=".

The code generation property TypeControl of a class may take the following values:

- None: the type is not a controlled type
- InitializationOnly: the type is a controlled type, with only user-defined initialization.
- AssignmentFinalizationOnly: the type is a controlled type, with only user-defined assignment and finalization.
- All: the type is a controlled type with both user-defined initialization and user-defined assignment and finalization.

TypeControl defaults to None. For a class whose TypeImplementation is not Tagged, TypeControl is dominated, and the generated type is not a controlled type. A class whose TypeControl property is not None must not be involved in a multiple inheritance relationship.

When discussing the effect of TypeControl, we'll use the following class hierarchies as examples:



If TypeControl is not None, the declaration of the type associated with a class is changed as follows:

- If the class has no superclass, the type is derived from Ada.Finalization.Controlled or Ada.Finalization.Limited_Controlled, depending on the value of the property IsLimited. This derivation occurs on the full type declaration:

```
package A is
   type Object is tagged private;
private
   type Object is new Ada.Finalization.Controlled with
        record
            ...        -- Attributes go here
        end record;
end A;
```

- If the class has a superclass, an auxiliary type is introduced, which contains the attributes of the class, and is used to build the actual type associated with the class. Again, this type is derived from Ada.Finalization.Controlled or Ada.Finalization.Limited_Controlled, depending on the value of the property IsLimited:

```
with B;
package C is
    type Object is new B.Object with private;
private
    type Controlled_Object is new
            Ada.Finalization.Controlled with
        record
            ...        -- Attributes go here
```

```
        end record;
    type Object is new B.Object with
        record
            Contents_Of_C : Controlled_Object;
        end record;
end C;
```

The name of the auxiliary controlled type is given by the code generation property TypeControlName, which defaults to Controlled_${type}. The name of the intermediate record component is always Contents.

If the code generation property TypeControl is set to InitializationOnly or to All, an overriding declaration for Initialize is inserted in the private part of the package (even if the controlled type is declared in the visible part):

```
package A is
    type Object is tagged private;
private
    type Object is ...;
    procedure Initialize (What : in out Object);
end A;
package C is
    type Object is new B.Object with private;
private
    type Controlled_Object is ...
    procedure Initialize
                        (What: in out Controlled_Object);
    type Object is new B.Object with ...
end C;
```

If the code generation property TypeControl is set to AssignmentFinalizationOnly or to All, overriding declarations are inserted for Adjust and Finalize in the private part of the package, and a declaration for the operator "=" is inserted in the visible part. Adjust is only declared if IsLimited is False:

```
package A is
    type Object is tagged private;
    function "=" (Left, Right : in Object) return Boolean;
private
    type Object is ...
    procedure Adjust (What : in out Object);
    procedure Finalize (What : in out Object);
end A;

with B;
package C is
    type Object is new B.Object with private;
    function "=" (Left, Right : in Object) return Boolean;
private
```

```
    type Controlled_Object is ...
    procedure Adjust (What : in out Controlled_Object);
    procedure Finalize (What : in out Controlled_Object);
    type Object is new B.Object with ...
end C;
```

In the declaration of procedures Initialize, Adjust and Finalize, the name of the only parameter is given by the code generation property ImplicitParameterName for the class. In the declaration of operator “=”, the parameters are named Left and Right.

The code generation property TypeControl, when it is not None, dominates the properties GenerateDefaultConstructor, DefaultConstructorName, GenerateCopyConstructor, CopyConstructorName, GenerateDestructor, DestructorName, GenerateTypeEquality and TypeEqualityName: no standard operation is generated, and the name of the equality operator, when it is generated, is always “=”. This is because standard operations and controlled types are two different mechanisms to achieve similar effects, and they are not intended to coexist in a single class.

GenerateGet and GenerateSet may be used in conjunction with controlled types: the accessor operations which are generated correctly take into account the internal structure of the type (possibly with an auxiliary controlled type) to access the various components.

A class whose code generation property TypeControl is not None may be abstract. However, the auxiliary controlled type (if generated) is never made abstract, and the Initialize, Adjust and Finalize procedures (if generated) are not made abstract either.

OOD and Ada 83

3

Contents

This chapter is organized as follows:

Note: Because UML and Ada use the word “package” to designate two different concepts, this document uses the phrase “UML package” for a package in the UML acceptation, and the word “package” without qualification for an Ada package.

Mapping Classes

The following kinds of classes in the UML notation have a mapping to Ada:

- Standard Classes
- Utilities
- Parameterized Classes
- Bound Classes

Standard Classes

A class, as defined by UML, is a set of objects that share a common structure and a common behavior. This concept is best represented as an Ada package with a private type and a set of visible subprograms.

The structure of a class is a private or limited private type, implemented as a record type. The name of the type defaults to Object. Each “has” relationship, generalization relationship, and attribute becomes a field in the record. Optionally, there may be an additional access type, called Handle, that points to the class type.

Using this representation of a class in Ada, an object is simply an instance (i.e., variable declaration) of the class type and is accessed, manipulated, and controlled by the subprograms in the class package.

Class Operations

The behavior of the class is captured by the subprograms in the visible part of the package. Each operation defined in the class is mapped to either an Ada procedure or function. The formal parameter list begins with the class type, whose name defaults to this.

Usually, several standard operations are needed for every class. Constructors (default name: Create), are responsible for creation and initialization of class objects. A copy constructor adds additional logic required when copying the contents of one object to another. The destructor (default name: Free) may deallocate memory or call other destructors. Finally, an equality operation can be added when "=" does not make sense.

Export control adornments can be attached to operations. If the export control is public, the subprograms will be part of the visible part of the package. Otherwise, the subprogram will be hidden in the body.

Utilities

Generally, a utility is used to collect a set of free subprograms that are cohesive by some measure. For instance, consider a collection of subprograms (String_Compare, Upper_Case, …) that manipulate a string, yet do not need any direct access to the structure of a string. These can be gathered together into a utility.

In Ada, a utility is represented as a package containing a collection of subprograms. These packages typically have names ending with suffixes like _Utilities, _Services, etc. A utility package has no class type.

Parameterized Classes

A parameterized class in the UML notation corresponds to a generic package in Ada. Class parameters become generic formal parameters.

Bound Classes

A bound class maps to a generic instantiation in Ada.

Mapping Relationships

The following relationships defined in the UML notation have a defined mapping to Ada:

- Dependency Relationships
- Has Relationships

- Generalization Relationships (Inheritance)
- Association Relationships

Dependency Relationships

The dependency relationship means that a client class is dependent on the interfaces of a supplier class. A dependency relationship maps to an Ada with clause. Note that a “has” association or generalization relationship also implies a with clause.

Export control adornments on a dependency relationship define the location of the with clause. If the relationship is public, the clause will be in the package specification. Otherwise, it will be in the body.

Has Relationships

“Has” relationships are not part of the UML notation. However, they can be created in Rose using the **View > As Booch** option. When viewed using the Booch or OMT notation, they are displayed as unidirectional aggregation relationships.

The “has” (aggregation) relationship denotes a whole / part association. There are two distinct types of “has” relationships: by-value and by-reference. A by-value “has” relationship, also known as physical containment, generally indicates that the part does not exist independently of the whole, and / or the whole is responsible for construction and destruction of the part. A by-reference relationship, also referred to as logical containment, indicates that the part is not physically contained within the whole and is potentially shared with other objects.

A “has” relationship becomes a component in the client's class record type. The type of the record component depends on the by-value or by-reference nature of the relationship. If the relationship is by-value, the type of the component is the class type of the part class (i.e., Object). If the relationship is by-reference, the component type must use the access type of the part class (i.e., Handle).

When the static adornment is added to a “has” relationship, the relationship is interpreted as being a class relationship rather than an object relationship. In Ada, this means that the relationship will be represented as a variable declaration in the private part of the client's package.

Generalization Relationships (Inheritance)

Ada 83 has no direct language support for inheritance. With the help of automation, however, inheritance can be achieved. There are actually several ways to support inheritance; the one chosen for the Ada 83 Generator is the best balance of understandability, extensibility, and simplicity.

Inheritance can best be achieved by using type extension, which builds on an existing class by inheriting, modifying, and/or adding to both the structure and behavior of the existing type. In Ada, type extension is accomplished by creating a new class package that re-declares all of the subprograms of the superclass, and declares a new class type that includes an instance of the superclass as a component. The implementation of the re-declared subprograms simply call back to the subprograms in the superclass' package. The subclass' package can then be extended by adding additional attributes, relationships, and operations, and/or overriding the implementation of the re-declared subprograms.

Association Relationships

Associations are similar to “has” relationships.

- For unidirectional associations, the generated code is identical to that which would be generated for a “has” relationship.
- For bidirectional associations the data structures are identical to that which would be generated for two symmetrical “has” relationships. An association provides a set of operations that preserve the integrity of the linkage between the objects.
- Association classes provide an additional mechanism to store and retrieve the information held by the association class.

Achieving Polymorphism with Ada

Because Ada 83 has no built-in polymorphism, the Ada 83 Generator produces the subprograms and data structures needed to emulate polymorphism.

This technique consists of creating a union package over the root class and its direct subclasses. This package consists of a variant record type that uses an enumeration type listing the possible variants. The enumerated type includes the root class and all subclasses. This package also re-declares all of the subprograms exported by the superclass. The body of each of these subprograms uses the discriminant of the variant record to dispatch a call to the appropriate subprogram.

Unmapped Elements for Ada

The following elements are part of the UML notation, and can be described in Rational Rose, but have no mapping to the Ada language. They are ignored or flagged by the code generator:

- Metaclasses
- Abstract classes
- Friendship
- Multiple inheritance

Ada Code Generation 4

Contents

This chapter is organized as follows:

What is the Ada Generator?

The Ada Generator is the code generation capability that is provided by the Ada 95 / Ada 83 add-in to Rational Rose. The commands for the Ada Generator are located in the **Ada 95/Ada 83** submenu of the Rose **Tools** menu.

You use the Ada Generator to generate Ada units from information in a Rose model. These units contain Ada code constructs that correspond to the notation items (classes, relationships, and adornments) you have defined in the model via diagrams and specifications.

The Ada Generator provides code-generation properties that control the kinds of Ada code constructs that are generated for the various kinds of notation items in the model. You can use the default values for these properties or you can specify different values to generate the code you want.

The Ada Generator inserts specially-marked code regions into the generated files where you can add further code (for example, to fill in extra private declarations in a package specification). By default, such regions are preserved, so you can regenerate the file without losing the code you added.

The Ada Generator may generate code in a directory hierarchy or, if Rational Apex is available, in subsystems and views. In order to generate code in subsystems and views, the Apex add-in must be activated, and the property CreateApexSubsystemAndView of the Apex add-in must be set to “yes”. The Ada

Generator, when generating code for Apex, makes use of some properties defined by the Apex add-in. These properties have a name which starts with “Apex” and are described in the documentation for the Apex add-in.

Basic Steps for Iterative Code Development

The basic strategy for generating code is to use the default values for code-generation properties initially, and later introduce non-default values as needed. This section describes these steps for generating Ada units from a Rose model:

- Overview
- The Generated Files
- The Basic Code Contents
- Entering Parameters for Parameterized Classes
- Entering Static Attributes and Metaclass Attributes
- Evaluating the Generated Code
- Completing the Implementation of the Generated Code
- Regenerating Code

Overview

In order to generate Ada 95 or Ada 83 code, you must first activate the Ada 95 / Ada 83 add-in using the Add-In Manager, which is accessible from the **Add-Ins** menu.

Then, you must set the default language for your model to be Ada 95 or Ada 83: choose the **Tools > Options** menu item, and in the **Options** dialog box click the **Notation** tab; use the **Default Language** list to select Ada 95 or Ada 83.

You may generate a different language for some classes by associating them with a component that has a different language.

By default, code is generated in the current directory or working view (determined initially when you start Rose and changed each time you open a model in a different view). If this is unacceptable, you can specify a default view before generating code.

1. Start Rose, if necessary.
2. Create or open the Rose model from which you want to generate code and display an appropriate class diagram.
3. Select one or more class items (classes, utilities, parameterized classes and bound classes) or UML packages.
4. Choose the **Code Generation** command from the **Tools > Ada 95** submenu. If code generation fails, inspect the log.

5 Evaluate the generated code. Based on your evaluation, you can change the model and/or code-generation properties, and then regenerate the code.

The Generated Files

The generated files are placed in a directory based on the properties of the model and the component UML packages. By default, each logical or component UML package in Rose is associated with an Apex view within a subsystem (if Apex is available) or with a hierarchy of directories (if Apex is not available).

In general one specification file (.1.ada) is generated for each class you selected in the diagram. The name of each file is derived from the name of the corresponding class. If you selected a UML package, a file is generated for each class in the UML package.

Note that the generated file structure realizes the physical portion of your Rose model. If you have developed only a logical model (class diagrams), the Ada Generator assumes an implicit physical model in which each class is effectively assigned to an implicit module specification, and therefore an Ada package specification.

The Basic Code Contents

The content of the generated code is based on the notation items in the logical portion of your model. In general:

- Each selected class generates a private record declaration and visible operations in a package specification. In addition, an optional access type, known as a handle, can be generated.
- Each of a class's "has" relationships generates a component. The relationship's containment and multiplicity partly determine the type of the component, and may create additional supporting type declarations.
- Each of a class's navigable association roles generates a component. The role's containment and multiplicity partly determine the type of the component, and may create additional supporting type declarations.
- Each operation in a class specification generates a subprogram declaration in the package specification.
- (Ada 95) Generalization relationships generate type derivation.
- (Ada 83) Generalization relationships generate components in the record declaration. In addition, all non-standard operations in the superclass are duplicated in the subclass package specification.
- Each selected utility generates a package specification with subprogram and object declarations only.

- “Has”, generalization, association and dependency relationships result in appropriate with clauses.
- If desired, a body is generated for each specification, with stubbed code for the user-defined operations.

The Ada Generator takes into account all model information that pertains to the selected class items, even information that does not appear in the diagram. For example, a component is generated for every “has” relationship that is defined for a class, including “has” relationships defined on other diagrams or in the class specification.

Entering Parameters for Parameterized Classes

The parameters for parameterized classes are entered in Rose using a dialog box which has two fields: Name and Type. Because there is such a large variety of formal parameters in Ada generics, and of discriminants in unconstrained types, users must follow a convention that specifies the nature of the parameters. Roughly speaking, the Name field contains the name of the parameter, and may start with an Ada keyword that indicates its nature. The type field contains any additional information that may be needed to complete the formal parameter or discriminant declaration. The Ada Generator adds the syntactic glue required by the language, such as the reserved words **with**, **is**, **new**, and the colons and semicolons.

Here is a detailed list of the possible formal parameters, and how they may be entered in the Type and Name fields. Note that an anonymous access type is only allowed if the Unconstrained Type implementation is used. Conversely, formal types, procedures, functions, packages, and formal object with an explicit mode are only legal if the Generic implementation is used.

- Generic formal object: the Name field contains the name of the object; the Type field contains its type, possibly followed by a default value, and possibly preceded by a mode. For example:

```
Name:    Foo
Type:    in out Bar
```

  or:

```
Name:    Foo
Type:    Bar := 3
```

  In the case of the Unconstrained Type implementation, the above notation may be used to represent an access discriminant:

```
Name: Foo
Type: access Bar
```

- Generic formal type: the Name field contains the reserved word **type**, followed by the name of the type, and by discriminants, if any; the Type field contains the type definition. For example:

```
Name:    type Foo (D : Integer := 3)
Type:    tagged private
```

- Generic formal procedure: the Name field contains the reserved word **procedure**, followed by the name and parameters of the procedure; the Type field contains the default name for the formal procedure, if any. For example:

```
Name:    procedure Foo (X : in out Integer)
Type:    Bar.Func
```

- Generic formal function: the Name field contains the reserved word **function**, followed by the name, parameters and result type of the function; the Type field contains the default name for the formal function, if any. For example:

```
Name:    function Foo (X : Float) return Boolean
Type:    <>
```

- Generic formal package (Ada 95): the Name field contains the reserved word **package**, followed by the name of the formal package. The Type field contains the name of the corresponding generic package, followed by instantiation parameters. For example:

```
Name:    package Foo
Type:    List_Generic (<>)
```

For actual parameters (appearing in bound classes) the convention is the following: the Name field contains the value of the actual parameter, and the Type field contains the name of the formal parameter. For example, if a parameterized class has the following parameters:

```
Name:    Foo
Type:    Float
```

it may be instantiated using the following parameters:

```
Name:    3.14
Type:    Foo
```

Entering Static Attributes and Metaclass Attributes

Static attributes and metaclass attributes can result in a wide variety of (package-level) declarations. They are entered in Rose using a dialog box which has two fields: Name and Type. In order to control the nature of the declaration that is generated, users must follow the following conventions:

- The Name field must contain the simple name of the entity being declared, i.e., the part that appears before the colon in the Ada declaration. No colon or semicolon may appear in the Name field.
- The Type field must contain anything that appears after the colon in the Ada declaration. However, no initial value must be specified. Instead, the code generation property InitialValue must be used if an initial value is to be generated for the declaration. No colon or semicolon may appear in the Type field.

The following examples demonstrate how to use these fields, and what is the corresponding Ada declaration:

- Variable:

  ```
  Name: Foo
  Type: Integer
  ```

  Generated declaration:

  ```
  Foo : Integer;
  ```

- Constant:

  ```
  Name: Foo
  Type: constant Boolean
  ```

  Code generation property InitialValue: “False”

  Generated declaration:

  ```
  Foo : constant Boolean := False
  ```

- Named Number:

  ```
  Name: Foo
  Type: constant
  ```

  Code generation property InitialValue: “3.14”

  Generated declaration:

  ```
  Foo : constant := 3.14;
  ```

- Exception:

  ```
  Name: Foo
  Type: exception
  ```

  Generated declaration:

  ```
  Foo : exception;
  ```

- Renaming:

```
Name: Foo
Type: Integer renames Bar
```

Generated declaration:

```
Foo : Integer := Bar;
```

Evaluating the Generated Code

After you have located the generated files, you evaluate them to determine whether to use them as generated. Based on your evaluation, you may decide to regenerate the code after refining the model, adjusting the values of code-generation properties, or both.

Use the information provided in the rest of this chapter to guide your evaluation. Each section lists some of the things you can change about a particular aspect of code generation.

Completing the Implementation of the Generated Code

When you are satisfied with the way code is generated from your model, you complete the code by implementing the package bodies. If you did not use the Ada Generator to create stubbed bodies, you can select the specifications in Apex, and choose the **Build Body** command from the **Compile** menu. Rational recommends, however, that you let Rose generate code for the bodies, since it will produce the appropriate code regions.

To complete the implementation of your code, you may insert additional statements and/or declarations in the preserved code regions. A preserved code region is a special block of comments starting with **--##** and containing the clause preserve=yes. Preserved code regions are preserved by the code generator the next time the code is regenerated. This makes sure that you may continue evolving your model in Rose after you have started refining the implementation of the code. Note that some of the code regions that Rose generate have preserve=no, so if you want them preserved, you must change this clause to preserve=yes.

You cannot add your own code regions: if you try to do this, they will be considered orphaned by the code generator (see below). You must use the code regions produced by the Ada Generator. Here is a list of the code regions that the Ada Generator produces:

- The module.cp region, which appear at the beginning of the unit, contains the text found in the property CopyrightNotice of a Module Spec/Body. This region may be preserved if the region is modified manually.

- The module.withs region, which follows the *with* clauses and precedes the compilation unit, may be used to insert additional *with* clauses, *use* clauses, or pragmas.
- The module.declarations region, which occurs at the beginning of the package visible part and at the beginning of the package body, may be used to insert additional declarations.
- The module.additionalDeclarations region, which occurs at the end of the package visible part and at the end of the package body, may be used to insert additional declarations.
- The module.privateDeclarations region, which occurs at the beginning of the private part, may be used to insert additional declarations.
- The module.additionalPrivateDeclarations region, which occurs at the end of the private part, may be used to insert additional declarations.
- The module.statements region, which covers the statement part of the package body, may be used to insert statements which are executed at elaboration time. By default, the statement part of any package body contains a single **null** statement.
- The class_name.operation_name%context.id.declarations region, which covers the declarative part of each generated subprogram, and may be used to insert declarations in the subprogram body. The name of this section is generated by Rose from the class name, the operation name, and various other pieces of information that help disambiguate the identity of the subprogram.
- The class_name.operation_name%context.id.statements region, which covers the statement part of each generated subprogram, and may be used to insert statements in the subprogram body. The name of this section is generated by Rose from the class name, the operation name, and various other pieces of information that help disambiguate the identity of the subprogram.

Regenerating Code

You can regenerate code for a given set of class items by following the same steps you used to generate the original code. When you regenerate code into existing files, the current contents of these files are saved in backup files before the new contents are written. By default, each backup file has the extension .1.ad~ or .2.ad~, as appropriate. The same backup files are overwritten each time you regenerate code to the same source-code files. The regenerated files:

- Reflect any changes you made to the model or to properties.
- Contain any code regions you edited in the previously generated version of the files, provided that the preserve keyword for each region was set to yes.

Note that if you delete or rename a notation item for which a code region was preserved, that region is “orphaned” when you regenerate code. This means that the Ada Generator places the code region in a special section at the end of the regenerated file so that you can decide whether to reuse any of the edits you made in that region. The Ada Generator automatically changes the preserve keyword to no in orphaned regions, so that they are discarded the next time you regenerate the file.

Refining the Subsystem and View Structure

Determining the Directory for an Ada File

There are several properties which the Ada Generator uses when determining the directory for an Ada file, if Apex is available:

- The project properties Directory and ApexView
- The UML package properties ApexSubsystem and ApexView

The directory for a module is based on the concatenation of the project Directory property, and the UML package’s ApexSubsystem and ApexView properties. Modules must be contained within component UML packages.

The directory for a class which has been assigned to a module is determined by applying these rules to its assigned module. The directory for a class which has not been assigned to a module is based on the UML package to which it is assigned: if it is enclosed in a logical UML package which is assigned to a component UML package, its directory is created from the ApexSubsystem and ApexView properties for the component UML package. If ApexSubsystem is blank, the subsystem name is set to the name of the component UML package.

If it is enclosed in a logical UML package which is not assigned to a component UML package, its directory is created from the default values of ApexSubsystem and ApexView properties, plus the project Directory property. If the default ApexSubsystem property is blank, the subsystem name is set to the name of the logical UML package.

If Apex is not available, a hierarchy of directories is created using the name of the component UML packages (if they exist) or of the logical UML packages (in the absence of component UML packages).

Mapping Classes and Modules to Ada Units

By default, each class is assigned to an implicit module specification. From these implicit modules, the Ada Generator produces a package specification containing the class definition. The units are generated according to the values in the default module-spec property set.

To change the default mapping from classes to units, you may either change the class name, or assign two or more classes to the same module, as follows:

1. Introduce component diagrams into your model.
2. Create a module specification for each Ada specification you want to generate.
3. Assign each class to the appropriate module via the class's specification: to generate a package specification, you assign the class to a module specification. To generate the code for multiple classes in a single package, you assign each class to the same module.

Specifying Filenames

The name of a generated file has two parts: a name and an extension, separated by a period (for example, foo.1.ada). The name is generated automatically, and the extension is controlled by different code-generation properties. If you are using Rational Apex, you should not change these values.

When a file is generated from a module, the filename is determined by the name of the module: it is the same as the module name, except in lowercase.

In the default case where classes are mapped to implicit modules, each implicit module assumes the name of the corresponding class. Consequently, each generated filename is based on the implicit module name (and, indirectly, on the class name).

To specify a non-default file name for a generated class, introduce a component diagram, if necessary, and assign the class to a module specification with the desired name.

Refining Class Definitions (Ada 83)

The Ada Generator creates a type declaration for each selected class. The format of the type depends on the following property values:

- Class Name
- Discriminant
- Implementation Type
- Is Subtype

- Is Task
- Variant

See *Code Generation Properties* on page 89 for more information on each property.

Standard Operations

Standard operations are subprogram declarations that are commonly found in Ada classes. They include:

- Default constructor
- Copy constructor
- Destructor
- Equality operation

By default, each class is generated with a default constructor, copy constructor, and destructor. Class properties permit you to specify the kind (procedure or function) and name for some of these standard operations.

Note that you can overload a standard operation by setting the relevant class property to cause it to be generated, and then specifying one or more additional operations with the same name, but different parameters in the class specification.

User-Defined Operations

User-defined operations are subprogram declarations that are generated from the operations you define in a class specification. Note that you do not need to define standard operations in a class specification, unless you want to overload them (see above).

If you want additional subprogram declarations for a class, or if you want different arguments or return types, you must edit the class specification.

One operation property, ClassParameterMode, permits you to specify the parameter mode of the class parameter, which is included automatically.

Get and Set Operations

Get and set operations are subprogram declarations that provide access to components. By default, a pair of get and set operations are generated from each “has” relationship, providing the relationship is public.

You can suppress the generation of a get and set operations by blanking-out the GetName and SetName properties in the property set that is attached to the has relationship. To define your own get and set functions, you define them as you would any other user-defined operation in the class specification.

Inherited Operations

When one class (called a subclass) inherits another class, all of the visible user-defined, get, and set operations defined in the superclass get replicated in the package specification of the subclass. This is how Ada 83 can achieve inheritance: the data is inherited by adding a field to the record, and the operations are inherited by replicating them in the subclass definition.

When you implement the body of an inherited operation, you typically do nothing except call the operation of the inherited class with record field that matches that class. If you do anything else, you are overriding that operation.

Record Fields and Object Declarations

Record fields are generated from “has”, association and generalization relationships and attributes defined in diagrams or in specifications. (If you have set the static adornment on the “has” relationship, an object declaration in the private part of the package specification is generated.

The component type is determined by a number of factors. By default, the type is determined by a combination of the supplier class and the multiplicity and containment of the “has” relationship.

In the simplest cases, the component type is:

- The class name of the supplier class for a one-to-one by-value relationship.
- The handle name of the supplier class for a one-to-one by-reference relationship.

In more complex cases (maximum allowable cardinalities larger than 1), the Ada Generator inserts a container class for the component type, which you can either use as generated or replace with the name of a container class of your own.

For bounded containers, the Ada Generator creates an array declaration in the private part of the class package specification.

For unbounded containers, the Ada Generator instantiates a container generic package in the private part of the package specification.

You replace these default container classes by setting the various Container class properties.

Specifying Additional Ada Unit Contents

You can tailor aspects of the structured comments and context clauses that appear at the beginning of the generated Ada units. You can also cause the Ada Generator to generate visible declarations at the beginning of one or more units.

Adding Structured Comments

The Ada Generator inserts a block of structured comments at the beginning of each generated file. You can set properties to generate a copyright notice string in these comments.

In the default case where classes are mapped to implicit modules, you edit properties in the default module-spec property set, which is attached to the implicit modules. If you have explicitly assigned classes to modules, you must edit each property set that is attached to a module.

Adding With Clauses

By default, the Ada Generator produces with clauses in units based on class relationships and module dependencies in your model. If you want additional with clauses to appear in one or more generated files, use one of the following methods, as appropriate.

If you want more generated units to reference each other in with clauses, you can inspect the relationships among existing items in the model to determine whether you have represented them adequately.

For example, you may find that you need to add a uses relationship from one class to another, which will cause a with clause to be generated in the first class's Ada unit. (A with clause is generated only if the classes are generated in different units.)

Similarly, you can introduce dependencies among modules in a module diagram, which result in generated with clauses.

If you want any of the generated units to reference units that are not among the generated units, you can use the AdditionalWiths property to insert additional with clauses to reference those units.

If you want to put a special with clause in just one or two generated units, you can do so by editing these units directly. To do this, you insert the desired with clauses between these source markers at the beginning of the unit:

```
--##begin module.withs preserve=yes
--##end   module.withs
```

Adding Global Declarations

You can cause the Ada Generator to generate global declarations before the first class definition in a unit. To do this, you:

1. Introduce a module diagram, if necessary, and assign one or more classes to a module specification (or body, as appropriate).
2. Double-click on the module specification to bring up its specification.
3. Enter the desired declaration(s) in the Declarations box. The text you enter here will be inserted at the beginning of the generated unit.

Reverse Engineering from Apex

5

Contents

This chapter is organized as follows:

Rose can analyze Ada 83 / Ada 95 code compiled with Rational Apex and generate a Rose model containing class and component diagrams that present a high-level view of the code.

Note that this capability is only available for Ada units that have been compiled with the Apex compiler, and that all units must be in the installed (analyzed) or coded states.

Basic Operations

The reverse engineering tool can create both class diagrams and component diagrams. Class diagrams show the high-level relationships between Ada units and types, and the operations and data structures associated with each type. Component diagrams come in two forms:

- An Ada unit diagram, which displays the "with" structure of the Ada units in a program, independent of subsystem structure.
- A subsystem diagram, which displays the import structure of the views you specify.

Within each view is a display of the "with" structure of the Ada units in that view.

Creating the Model File

No matter which type of diagram you want, the reverse engineering tool always generates a model file, called rose_ada.mdl by default. This file can be opened within Rose for layout and display.

Select the Ada unit or view you wish to diagram, and choose **Reverse Engineer...** from the **Rose > Ada** Apex submenu. You will see the Reverse Engineer dialog box, where you can modify various options. Choose **OK** or **Apply** to create the model file. See *Dialog Box Options* on page 84.

Displaying the Model

Once you have created the model file, you can load it into Rose. Select the file in the directory viewer (you may need to do **File > Redisplay** first). Then choose **Start Rose** from the **Rose > Ada** submenu. This will invoke Rose and display the model.

Note: For traversal to work, you must invoke Rose from the Apex menu. If Rose is already running before you started Apex, exit Rose and restart from the Apex menu command.

Once Rose is invoked, your next action depends on whether you created a class diagram or a component diagram. If you created a class diagram, choose **Format > Layout Diagram** to format the diagram. If you created a component diagram, choose **Browse > Component Diagram**. Select the <Top Level> / Main component diagram and choose **OK**. When the module is displayed, you will see the UML packages or units displayed in a straight diagonal line. Layout the diagram by choosing **Format > Layout Diagram**.

If you created a component diagram, you can double-click on a UML package box to see the units within that view. You will need to run **Format > Layout Diagram** on each UML package individually.

If you created a class diagram based on Apex views, you will see UML packages in the top-level class diagram. Double-click on the UML package to see the classes and utilities in that view. You will need to run **Format > Layout Diagram** on each UML package individually.

Use **File > Save** to save the model with the diagrams laid out.

To traverse from an unit in a Rose diagram to the actual Ada source code, select the unit and choose **Browse > Browse Spec**. This will invoke the Apex editor for that unit.

Dialog Box Options

Here is a brief description of each option in the **Reverse Engineer** dialog box:

Include Closure of Views/Units

With this button selected, reverse engineering processes all selected views or units, plus the import closure or Ada closure. This option is the default.

Exclude Views/Units with Prefix

Use this option to exclude views or units starting with a given prefix. For instance, you might want to exclude the rational_dir/base/ada area.

Include Views/Units with Prefix

Use this option to include *only* views or units starting with the given prefix. This option would let you limit your diagram to a particular project, for example.

Include only Views/Units Selected

When this option is selected, only the views or units on the right side of the Objects or Views area are included in the petal file.

Petal File Name

By default, reverse engineering creates a file called rose_ada.mdl. Use this box to have it create a different file.

Include Classes

If you select this button, reverse engineer creates a class diagram of the units or views selected.

Include Modules

If you select this button, reverse engineering creates a component diagram of the units or views selected.

If neither Include Classes nor Include Modules is selected, a component diagram showing just the import structure of the subsystems is created.

How Ada Is Represented in a Class Diagram

The reverse engineering tool uses various algorithms to map Ada constructs to the UML notation, based primarily on the mapping described in Chapter 1.

Mapping Package Specifications (Ada 95)

An Ada package becomes a utility if contains subprograms which are not operations of some class-like type declared in the same package (see below). Each of these subprograms becomes an operation of the utility.

Packages that contain only subprograms associated with some class-like type do not correspond directly to an entity of the class diagram (although the class-like types that they contain do). Their name can still be used to generate the prefix of entity names that use the colon notation.

All package specifications result in the creation of a package specification module in the proper component diagram. The “with” relationships between packages result in the creation of dependency relationships between the corresponding modules.

Mapping Package Specifications (Ada 83)

An Ada package will become either a utility or a class. To become a class, the package must meet the following criteria:

- It must define at least one private record type
- All visible subprograms must include a parameter with a private record type

Mapping Type Declarations (Ada 95)

Only those types which are *class-like* result in the creation of a class in the model. The distinction between class-like types and other types is important, because it avoids cluttering the model with classes that would correspond to minor type declarations, introduced for low-level implementation reasons.

The definition of class-like types is as follows:

- A private or limited type is class-like (regardless of the nature of its full type declaration.
- A record or tagged type is class-like (even if it is not private).
- A task or protected type is class-like (note that the existence of a task or protected *object* doesn’t cause the creation of a class).

All other types are not class-like. Such types do not cause the creation of a class, although they may be used to fill some other information of the model (e.g., code generation properties). Note in particular that (non-private) access and array types, which are produced by the code generator to implement by-reference relationships and multiplicities larger than 1, are not class-like

For record and tagged types, the components become either attributes or “has” relationships. A “has” relationship is created if the type of the component is a class-like type, or an access type designating a class-like type, or an array type whose component is class-like, or access to class-like. The containment and multiplicity of the relationship is set accordingly, as well as the code generation properties that describe the container and access types. In all other cases an attribute is created.

The subprograms that include a class-like type as a parameter become operations of the class.

Mapping Type Declarations (Ada 83)

All types declared in the specification of a package become classes in the class diagram.

Most types become “implementation types,” where the ImplementationType property is set to the definition of the type, and where no operations or attributes are assigned to the class. These classes are not visible in the initial class diagram displayed by Rose, but can be made visible using **Query > Add Classes**.

If a type is a record type, defined in the private part of a package specification, it becomes a class. The components of the record become attributes, or “has” relationships, of the class. The subprograms that include the record type as a parameter become operations of the class.

If a type is an access type to a private record type, no class is created, but the Handle Name property of the referenced class is set based on the name of type.

Normally every class has an associated utility, which is the parent package where the type is declared. If, however, all subprogram declarations map to a class, then the first class that is not an implementation type becomes the representation of the entire package.

To associate each type with its enclosing package, Reverse Engineer creates a dependency relationship with the type as the supplier and the enclosing utility, or class if the utility is not needed, as the client. The relationship is named **decl,** a keyword that the code generator uses to determine whether a class is declared within the context of a utility or other class.

Details of a Has Relationship (Ada 83)

The multiplicity and access of a “has” relationship are determined by the type of each component of the record. If the type is a simple type, the multiplicity is set to **1**. If it’s an array, the multiplicity is set to the size of the array, or to ***** if the array is unbounded. If the type is defined by a generic, and the generic is declared in the same package, the multiplicity is set to *****.

If the component of the record is an access type, the access is set to “by-reference,” and otherwise is set to “by-value.”

Mapping Subprogram Declarations

All subprograms declared in an package specification, visible or private, become operations. If there is a class-like type declaration, and the subprogram includes a parameter of that type, or is a function that returns that type, then the operation is assigned to that class. Otherwise, the operation is assigned to the utility that corresponds to the package.

Mapping Object Declarations

An object declaration is a variable, constant, or named number declared in a package specification. Each object declaration becomes a static attribute or “has” relationship. If the object is a constant, the IsConstant property is set to True.

If the package where the object is declared contains at least one class-like type, and all subprograms are associated to classes, then the objects become static attributes of the first class found in the package. Otherwise, the objects become static attributes of the utility.

An exception declaration, while not technically an object, maps to an attribute using the same algorithms described above for variables and constants.

Mapping “With” Clauses

Reverse engineering tracks the With clauses that would be generated by the “has” relationships between the various classes in the package specifications. The remaining “with” clauses, those that are used for parameter types and return types, become dependency relationships in the model.

Special Handling for Subsystems in the $APEX_BASE Directory

Since the subsystems in the $APEX_BASE directory are defined by Apex, doing a complete analysis only wastes space in the model. However, some analysis of the types defined in these subsystems is required to guarantee that “has” relationships in other subsystems have classes as their suppliers. Thus, reverse engineering examines only the type declarations in these subsystems, and does not evaluate attributes or operations.

Code Generation Properties

6

Contents

This chapter is organized as follows:

Model Properties

The model properties are described on the following pages:

Spec File Extension

The Spec File Extension property specifies the file name extension that the Ada Generator uses when creating Ada specification files. For Rational Apex the extension should be .1.ada.

Spec File Backup Extension

If the Ada Generator produces an Ada specification file that already exists, the previous version of the file is renamed to a backup file. The Spec File Backup Extension property specifies the file name extension that the Ada Generator uses when creating backup files for Ada specifications.

Spec File Temporary Extension

When the Ada Generator writes a specification file, it actually writes the code to a temporary file. Once the code is completely written, the following steps are taken:

1. The backup file (see the Spec File Backup Extension property) is deleted, if there is one.
2. The existing specification file is renamed to the backup file, assuming an existing specification file is present.
3. The temporary file is renamed to be the new specification file.
4. The Spec File Temporary Extension property specifies the filename extension that the Ada Generator uses when creating temporary specification files.

Body File Extension

The Body File Extension property specifies the file name extension that the Ada Generator uses when creating Ada body files. For Rational Apex, the extension should be 2.ada.

Body File Backup Extension

If the Ada Generator produces an Ada body file that already exists, the previous version of the file is copied to a backup file. The Body File Backup Extension property specifies the file name extension that the Ada Generator uses when creating backup files for Ada bodies.

Body File Temporary Extension

When the Ada Generator writes a body file, it actually writes the code to a temporary file. Once the code is completely written, the following steps are taken:

1. The backup file (see the Body File Backup Extension property) is deleted, if there is one.
2. The existing body file is renamed to the backup file, assuming an existing body file is present.
3. The temporary file is renamed to be the new body file.
4. The Body File Temporary Extension property specifies the filename extension that the Ada Generator uses when creating temporary body files.

Create Missing Directories

The Create Missing Directories property indicates whether or not the Ada Generator should create directories needed to mirror the model's UML package hierarchy, or stop and report an error if such directories are missing.

The default setting is `True`.

Generate Bodies

The Generate Bodies property indicates whether or not the Ada Generator should create Ada body files for the classes or modules that are selected for code generation.

When `True`, the Ada Generator will automatically create Ada bodies for selected classes and for module specs which have corresponding module bodies defined for them. Ada bodies will not be created for module specs which have no corresponding module body.

When `False`, the Ada Generator will not automatically create Ada bodies for selected classes or module specs. Ada bodies will still be created for module bodies that are explicitly selected.

The default setting is `True`.

Generate Standard Operations

The Generate Standard Operations property indicates whether or not the Ada Generator should create the standard operations for the classes selected for code generation. The property is used in conjunction with the class property of similar name. When set to `True`, the class property is then taken into consideration. When set to `False`, no standard operations are generated.

The default setting is `True`.

Implicit Parameter

The Implicit Parameter property indicates whether or not the Ada Generator should provide an implicit class parameter object for all the user-defined operations of a class. The property is used in conjunction with the class property of similar name. When set to `True`, the class property is then taken into consideration. When set to `False`, no implicit parameter is generated.

The default setting is `True`.

Stop On Error

The Stop On Error property indicates whether or not the Ada Generator stops generating code when the error count threshold is exceeded (see Error Limit property). This threshold does not apply to warnings (for which there is no limit) or fatal errors (which cause the Ada Generator to terminate immediately).

The default setting is `True`.

Error Limit

The Error Limit property specifies the error count threshold used in conjunction with the Stop On Error property.

The default setting is `30`.

File Name Format

The File Name Format property controls the automatic generation of directory and file names when the value of the model Directory property, or a UML package Directory property is `AUTO GENERATE`.

The value is expected to be an integer followed by zero or more flag characters. The integer is the maximum number of characters in a file or directory name. The flags are:

_	retain underscores
v	retain vowels
u	convert all letters to upper case
l	convert all letters to lower case
x	retain case

The default, if the property is blank, is to compress the filename to 8 characters on Windows, or 32 on UNIX, eliminate vowels first, eliminate white-space, and eliminate underscores. When a blank or underscore is eliminated, the next character is capitalized.

Directory

The Directory property specifies the project directory, which is the directory in which all subsystems for a project are generated. This property defaults to `AUTO GENERATE`, which tells the Ada Generator to use the current working directory.

Class Properties

The class properties are described on the following pages:

Representation

The Representation property is used to specify one or more representation items, including pragmas. The constructs must be fully defined, including a terminating semicolon. The following predefined names yield the name of the entity which can be used within the property definition:

- type
- access_type
- component_clauses -- yields a list of component_clause
- array_type -- Ada95 only
- access_array_type -- Ada95 only
- array_access_type -- Ada95 only
- access_array_access_type -- Ada95 only

The predefined "component_clauses" is used to construct the record representation clause in the class. For example,

```
for $type use
   record
```

```
      $component_clauses
   end record;
```

The placement of the representations, in a class, comes after the full definition of the type plus any other auxiliary type definitions. For a task or protected type, the representation comes after the "is" in the specification.

Generate Accessor Operations

The Generate Accessor Operations property indicates whether or not the Ada Generator should create the accessor operations for this class. Both the model and class property must be set to True for this to take effect.

The default setting is `True`.

Access Class Wide (Ada 95)

The Access Class Wide property specifies that the access type is class-wide. For example:

```
type AccessTypeName is access TypeName'Class;
```

The default setting is `True`.

Code Name

The Code Name property specifies the name for the class in the generated code. You need to set this property only if you want the class to be named differently than it is in the Rose model. This is especially useful when the Rose model and code are expressed in different natural languages. The value of this property should be a valid Ada identifier.

Type Name (Ada 95) / Class Name (Ada 83)

The Type Name property determines the Ada type name used by the Ada Generator to represent a Rose class. For example, if Type Name (Ada 95) / Class Name (Ada 83) is set to `File_Type`, the Ada Generator will output:

```
type File_Type is ...;
```

If Type Name (Ada 95) / Class Name (Ada 83) is set to `Object`, the Ada Generator will output:

```
type Object is ...;
```

You have the option of setting the Type Name (Ada 95) / Class Name (Ada 83) property to `${class}`, where the Ada Generator will use the name of the Rose class for the name of the type.

Note, that this property is ignored if the class name uses the colon notation, *ClassName:TypeName*.

The default setting is `Object`.

Type Visibility (Ada 95) / Class Access (Ada 83)

The Type Visibility (Ada 95) / Class Access (Ada 83) property controls the definition of the Ada type used by the Ada Generator to represent a Rose class.

Public	The type will be a public type.
Private	The type will be a private type. The corresponding complete type declaration will appear in the private part of the Ada specification.
Limited Private (Ada 83)	The type will be a limited private type. The corresponding complete type declaration will appear in the private part of the Ada specification.
Do Not Create (Ada 83)	No type declaration will be output by the Ada Generator.

The default setting is `Private`.

Type Implementation (Ada 95)

The Type Implementation property controls the implementation of the Ada type used by the Ada Generator to represent a Rose class.

Tagged	The class corresponds to a tagged type.
Record	The class is implemented using records and variants.
Mixin	The class is represented as a generic used in multiple inheritance.
Protected	The class corresponds to a protected type.
Task	The class corresponds to a task type.

The default setting is `Tagged`.

Type Control (Ada 95)

The Type Control property specifies whether a controlled type implementation should be generated for the Ada type. The Type Implementation property must be set to `Tagged`.

None	The type is not a controlled type.
Initialization Only	The type is a controlled type, with only user-defined initialization.
Assignment Finalization Only	The type is a controlled type, with only user-defined assignment and finalization.
All	The type is a controlled type with both user-defined initialization and user-defined assignment and finalization.

The default setting is `None`.

Type Control Name (Ada 95)

The Type Control Name property controls the name of the auxiliary controlled type.

The default setting is `Controlled_${type}`.

Type Definition (Ada 95) / Implementation Type (Ada 83)

The Type Definition (Ada 95) / Implementation Type (Ada 83) property allows a Rose class to be defined as something other than one of the types available in Type Implementation.For example, if Type Definition is set to `range 1 .. 500`, the Ada Generator will output:

```
type TypeName is range 1 .. 500;
```

If Type Definition is set to `new String (1 .. 4)`, the Ada Generator will output:

```
type TypeName is new String (1 .. 4);
```

For Ada 95, when the Type Definition property is set, it dominates the Type Implementation property.

Record Implementation (Ada 95)

The Record Implementation property controls the implementation of the Ada record type. It is used in conjunction with the property Type Implementation when set to Record.

Single Type	A single type is created for the complete generalization hierarchy.
Multiple Types	One record type is created for each class in the generalization hierarchy.

The default setting is `Single Type`.

Record Kind Package Name (Ada 95)

The Record Kind Package Name property controls the name of the auxiliary package used to declare the enumeration type Kinds of the root class.

The default setting is `${class}_Record_Kinds`.

Is Limited (Ada 95)

The Is Limited property controls whether the type is limited. This applies to tagged types and record types with private visibility.

The default setting is `False`.

Is Subtype

For Ada 95: The Is Subtype property is used in conjunction with the Type Definition property and a Single Type Record Implementation to define a subtype declaration.

The default setting is `False`.

For Ada 83: The Is Subtype property is used in conjunction with the Implementation Type property to define an subtype declaration. It is ignored when the Implementation Type property is blank.

Polymorphic Unit (Ada 83)

The Polymorphic Unit property tells the Ada Generator to treat the class as a polymorphic class instead of as a normal class. A polymorphic class must have a single dependency relationship, the supplier of which is the root of the generalization hierarchy for which a polymorphic package is to be created.

Handle Name (Ada 83)

The Handle Name property determines the name of the type created by the Ada Generator for “By Reference” instances of the class. For example, if Handle Name is set to `Handle` (and all other properties have their default values), the Ada Generator will output:

```
type Object is private;
type Handle is access Object;
```

If Handle Name is set to `Object_Name`, the Ada Generator will output:

```
type Object is private;
type Object_Name is access Object;
```

Handle Access (Ada 83)

The Handle Access property controls the definition of the Ada type used by the Ada Generator for “By Reference” instances of the class.

Public	(Default) The type will be defined as “access <Class Name>”
Private	The type will be defined as private
Limited Private	The type will be defined as limited private
Do Not Create	No type will be declared.

Discriminant (Ada 83)

The Discriminant property specifies the discriminant of the Ada type used by the Ada Generator to represent a Rose class. For example, if Discriminant is set to `Size : Positive := 100` (and all other properties have their default values), the Ada Generator will output:

```
type Object (Size : Positive := 100) is private;
```

The class property Variant and the “has” properties Container Type and Variant are also used when defining discriminated records.

Variant (Ada 83)

The Variant property is used in conjunction with the Discriminant property to define a single variant part for a discriminated record. The Variant property should be set to the simple name of a discriminant defined in the Discriminant property. For example, if Discriminant contains `Unit : Device := Disk` (and all other properties have their default values), the Ada Generator will output:

```
type Object (Unit : Device := Disk) is record
  ...
end record;
```

If Variant is set to `Unit`, the Ada Generator will output:

```
type Object (Unit : Device := Disk) is record
    ...
  case Unit is
    ...
  end case;
end record;
```

The Variant property is only used in the complete type declaration in the private part of the Ada specification. It has no effect on the visible type declaration. The Variant property is ignored when the Discriminant property is blank.

Generate Access Type (Ada 95)

The Generate Access Type property controls the generation of the Ada type used by the Ada Generator for By-Reference instances of the class.

Always	The type will always be generated.
Auto	The type will be defined as needed.

The default setting is `Auto`.

Access Type Name (Ada 95)

The Access Type Name property determines the name of the type created by the Ada Generator for By-Reference instances of the class. For example, if Access Type Name is set to `Handle`, the Ada Generator will output:

```
type TypeName is private;
type Handle is access TypeName;
```

If Access Type Name is set to `Object_Name`, the Ada Generator will output:

```
type TypeName is private;
type Object_Name is access TypeName;
```

The default setting is `Handle`.

Access Type Visibility (Ada 95)

The Access Type Visibility property controls the definition of the Ada type used by the Ada Generator for By-Reference instances of the class.

Public	The type will be defined as "access *TypeName*".
Private	The type will be defined as private.

The default setting is `Public`.

Access Type Definition (Ada 95)

The Access Type Definition property allows the access to a Rose class to be defined as something other than:

`type` *`AccessTypeName`* `is access` *`TypeName`*`;`

If Access Type Definition is set to `array (Positive range 1 .. 10) of Object`, the Ada Generator will output:

```
type AccessTypeName is
           array (Positive range 1..10) of Object;
```

Maybe Aliased (Ada 95)

The Maybe Aliased property specifies that the access type is a general access-to-variable type. For example,

`type` *`AccessTypeName`* `is access all` *`TypeName`*`'Class;`

The default setting is `False`.

Parameterized Implementation (Ada 95)

The Parameterized Implementation property controls the mapping of parameterized and bound classes.

Generic	The type will be declared in generic units.
Unconstrained	The discriminant part of the type is derived from the class parameters.

The default setting is Generic.

Parent Class Name (Ada 95)

The Parent Class Name property specifies the name used to reference the superclass, for a parameterized class whose Parameterized Implementation property has been set to Generic.

The default setting is `Superclass`.

Enumeration Literal Prefix

The Enumeration Literal Prefix property specifies the prefix that is prefixed to enumeration literal values, that the Ada Generator automatically generates.

The default setting is `A_`.

Record Field Prefix

The Record Field Prefix property specifies the prefix that is prefixed to component and discriminant identifiers, that the Ada Generator automatically generates.

The default setting is `The_`.

Array Of Type Name (Ada 95)

The property Array Of Type Name specifies the name of the array type of a one-to-many by-value “has” relationship. The string can include the variable `${type}`, which expands to the type name of the class.

The default setting is `Array_Of_${type}`.

Access Array Of Type Name (Ada 95)

The property Access Array Of Type Name specifies the name of the access type whose designated type is given by the property Array Of Type Name.

The default setting is `Access_Array_Of_${type}`.

Array Of Access Type Name (Ada 95)

The property Array Of Access Type Name specifies the name of the array type of a one-to-many by-reference “has” relationship. The string can include the variable `${access_type}`, which expands to the access type name of the class.

The default setting is `Array_Of_${access_type}`.

Access Array Of Access Type Name (Ada 95)

The property Access Array Of Access Type Name specifies the name of the access type whose designated type is given by the property Array Of Access Type Name.

The default setting is `Access_Array_Of_${access_type}`.

Array Index Definition (Ada 95)

The property Array Index Definition supplies the index subtype definition for the array type definitions given by the properties Array Of Type Name and Array Of Access Type Name.

The default setting is `Positive range <>`.

Generate Standard Operations

The Generate Standard Operations property indicates whether or not the Ada Generator should create the standard operations for this class. Both the model and class property must be set to True for this to take effect.

The default setting is `True`.

Note: To auto-generate set operation (Project / Class property Generate Standard Operations set to `True`) you must have the attributes set to public.

Implicit Parameter

The Implicit Parameter property indicates whether or not the Ada Generator should provide an implicit class parameter object for all the user-defined operations of this class. Both the model and class property must be set to True for this to take effect.

The default setting is `True`.

Implicit Parameter Name (Ada 95) / Class Parameter Name (Ada 83)

All operations of a class can have as an implicit parameter a class object. The Implicit Parameter Name (Ada 95) / Class Parameter Name (Ada 83) property specifies the formal parameter name used by the Ada Generator for this class object. For example, if the Implicit Parameter Name (Ada 95) / Class Parameter Name (Ada 83) is set to `This`, (the property Generate Standard Operations must be active for Ada 95; all other properties have their default values for Ada 83), the class destructor will be declared as:

```
procedure Free (This : in out TypeName);
```

If Implicit Parameter Name (Ada 95) / Class Parameter Name (Ada 83) is changed to `The_Object`, the class destructor would be:

```
procedure Free (The_Object : in out TypeName);
```

The Implicit Parameter Name (Ada 95) / Class Parameter Name (Ada 83) property also controls the declaration of the class parameter to the constructor subprogram, get / set subprograms, inherited subprograms and subprograms for user-defined operations. It does not affect the names of the class parameters to the copy and equality subprograms.

The default setting is `This`.

Generate Default Constructor (Ada 95)/Default Constructor Kind (Ada 83)

The Generate Default Constructor (Ada 95) / Default Constructor Kind (Ada 83) property determines the kind of subprogram declared as the class constructor by the Ada Generator. The declaration of a class constructor can also be suppressed. If Generate Default Constructor (Ada 95) / Default Constructor Kind (Ada 83) is set to `Function`, the declaration output by the Ada Generator will be of the form:

```
function Create return TypeName;
```

If Generate Default Constructor (Ada 95) / Default Constructor Kind (Ada 83) is set to `Procedure`, the declaration output by the Ada Generator will be of the form:

```
procedure Create
        (ImplicitParameterName : in outTypeName);
```

The properties Generate Standard Operations, Type Name (Ada 95) / Class Name (Ada 83), Implicit Parameter Name (Ada 95) / Class Parameter Name (Ada 83), and Default Constructor Name also affect the declaration of the class constructor.

Function	The class constructor will be declared as a function.
Procedure	The class constructor will be declared as a procedure.
Do Not Create	No class constructor will be declared.

The default setting is `Function`.

Default Constructor Name

The Default Constructor Name property controls the simple name of the class constructor subprogram. For example, if the Default Constructor Name property is set to `Create`, the Ada Generator will output:

```
function Create return TypeName;
```

If the Default Constructor Name property is set to `New_Item`, the Ada Generator will output:

```
function New_Item return TypeName;
```

The default setting is `Create`.

Inline Default Constructor

The Inline Default Constructor property specifies whether an inline pragma should be generated for the Default Constructor.

The default setting is `False`.

Generate Copy Constructor (Ada 95) / Copy Constructor Kind (Ada 83)

The Generate Copy Constructor (Ada 95) / Copy Constructor Kind (Ada 83) property determines the kind of subprogram declared as the class constructor by the Ada Generator. The declaration of a class constructor can also be suppressed. If Generate Copy Constructor (Ada 95) / Copy Constructor Kind (Ada 83) is set to `Function`, the declaration output by the Ada Generator will be of the form:

```
function Copy (From : in TypeName) return TypeName;
```

If Generate Copy Constructor (Ada 95) / Copy Constructor Kind (Ada 83) is set to `Procedure`, the declaration output by the Ada Generator will be of the form:

```
procedure Copy (From : in TypeName;
                To: in out TypeName);
```

Function	The copy constructor will be declared as a function.
Procedure	The copy constructor will be declared as a procedure.
Do Not Create	No copy constructor will be declared.

The default setting is `Function`.

Copy Constructor Name (Ada 95)

The Copy Constructor Name property controls the simple name of the class constructor subprogram. For example, if the Copy Constructor Name property is set to Copy, the Ada Generator will output:

```
function Copy return TypeName;
```

If the Copy Constructor Name property is set to `Clone_Item`, the Ada Generator will output:

```
function Clone_Item return TypeName;
```

The default setting is `Copy`.

Inline Copy Constructor

The Inline Copy Constructor property specifies whether an inline pragma should be generated for the Copy Constructor.

The default setting is `False`.

Generate Destructor (Ada 95)

The Generate Destructor property specifies whether a destructor is declared by the Ada Generator.

If Generate Destructor is set to `Procedure`, the declaration output by the Ada Generator will be of the form:

```
procedure Free (ImplicitParameterName : in outTypeName);
```

The properties Generate Standard Operations, Type Name, Implicit Parameter Name, and Destructor Name also affect the declaration of the destructor.

Procedure	The class destructor will be declared as a procedure.
Do Not Create	No class destructor will be declared.

The default setting is `Procedure`.

Destructor Name

The Destructor Name property controls the simple name of the class destructor subprogram by the Ada Generator. For example, if the Destructor Name property is set to `Free`, the Ada Generator will output:

```
procedure Free (
    ImplicitParameterName :
         in outTypeName);
```

If the Destructor Name property is set to `Deallocate_Item`, the Ada Generator will output:

```
procedure Deallocate_Item (
    ImplicitParameterName :
         in outTypeName);
```

The default setting is `Free`.

If the Destructor Name is blank, no destructor will be generated.

Inline Destructor

The Inline Destructor property specifies whether an inline pragma should be generated for the Destructor.

The default setting is `False`.

Generate Type Equality (Ada 95)

The Generate Type Equality property determines whether the function is declared or suppressed.

Function	The type equality function will be declared.
Do Not Create	No type equality function will be declared.

The default setting is `Do Not Create`.

Type Equality Name (Ada 95) / Class Equality Operation (Ada 83)

The Type Equality Name (Ada 95) / Class Equality Operation (Ada 83) property controls the designator of the equality function declared by the Ada Generator to compare class objects. For example, if the Type Equality Name (Ada 95) / Class Equality Operation (Ada 83) property is set to `${quote}=${quote}`:

```
function "=" (L, R : in TypeName)
    return Boolean;
```

If the Type Equality Name (Ada 95) / Class Equality Operation (Ada 83) property is set to `Is_Equal`, the Ada Generator will output:

```
function Is_Equal (L, R : in TypeName)
    return Boolean;
```

The default setting is `${quote}=${quote}`.

Handle Equality Operation (Ada 83)

The Handle Equality Operation property controls the designator of the equality function declared by the Ada Generator to compare class handles. For example, if the Handle Equality Operation property is set to `${quote}=${quote}` (and all other properties have their default values), the Ada Generator will output:

```
function "=" (L, R : in Handle) return Boolean;
```

If the Handle Equality Operation property is set to `Is_Equal`, the Ada Generator will output:

```
function Is_Equal (L, R : in Handle) return Boolean;
```

If the property is blank, no handle equality function is output by the Ada Generator.

Inline Equality

The Inline Equality property specifies whether an inline pragma should be generated for the Equality operations.

The default setting is `False`.

Is Task (Ada 83)

The Is Task property is used to define a class as a task type. Operations become entries, and attributes are ignored.

Operation Properties

The operation properties are described on the following pages:

Implicit Parameter Class Wide (Ada 95)

The Implicit Parameter Class Wide property specifies whether the class parameter is specified with the `'Class` attribute.

The default setting is `False`.

Representation

The Representation property is used to specify one or more representation items, including pragmas. The constructs must be fully defined, including a terminating semicolon. The predefined name “operation” yields the name of the entity which can be used within the property definition.

The placement of the representations for an operation comes after the subprogram specification.

Use Colon Notation

The Use Colon Notation property is used to control whether colon notation is permitted to be used. Turning this off will cause errors to be generated for classes using colon notation.

The default setting is `True`.

Generate Accessor Operations

The Generate Accessor Operations property indicates whether or not the Ada Generator should create the accessor operations for the classes selected for code generation. The property is used in conjunction with the class property of similar name. When set to True, the class property is then taken into consideration. When set to False, no accessor operations are generated.

The default setting is `False`.

Use File Name

The Use File Name property is used to control the Module Spec/Body Property File Name.

The default setting is `False`.

Code Name

The Code Name property specifies the name for the operation in the generated code. You need to set this property only if you want the operation to be named differently than it is in the Rose model. This is especially useful when the Rose model and code are expressed in different natural languages. The value of this property should be a valid Ada identifier.

Subprogram Implementation

The Subprogram Implementation property is used to control the code generated for a subprogram body. This property can take on the following values.

Abstract (Ada 95)	An abstract specification is generated.
Body	A specification and body is generated.
Renaming (Ada 95)	A specification and renaming-as-body is generated.
Separate	A specification and stub is generated.
Spec	A specification is generated.

In addition, the code generation property Inline is used to control whether or not a pragma `Inline` is generated for the operation.

The default setting is `Body`.

Renames (Ada 95)

The Renames property is used in conjunction with the Subprogram Implementation property when set to Renaming. It specifies the name of the renamed subprogram.

Generate Overriding (Ada 95)

The Generate Overriding property specifies whether an overriding declaration should be generated.

The default setting is `True`.

Implicit Parameter Mode (Ada 95) / Class Parameter Mode (Ada 83)

The Implicit Parameter Mode (Ada 95) / Class Parameter Mode (Ada 83) property determines the mode of the class parameter for standard and user-defined operations.

Access (Ada 95)	The mode of the class parameter is “access”
In	The mode of the class parameter is “in”
In Out	The mode of the class parameter is “in out”
Out	The mode of the class parameter is “out”
Function Return (Ada 83)	The operation will be declared as a function with the Ada type of Class (See property Class Name) as its return value.

Do Not Create	No implicit parameter will be declared

The default setting is `In Out`.

Generate Access Operation (Ada 95)

The Generate Access Operation property specifies whether an access operation should be generated.

The default setting is `False`.

Inline

The Inline property specifies whether an inline pragma should be generated for the operation.

The default setting is `False`.

Entry Code

The Entry Code property provides the capability to insert code or comments at the beginning of the subprogram. This property is useful for inserting instrumentation, or adhering to documentation standards.

Exit Code

The Exit Code property provides the capability to insert code or comments at the end of the subprogram. This property is useful for inserting instrumentation, or adhering to documentation standards.

Entry Barrier Condition (Ada 95)

The Entry Barrier Condition property specifies the boolean expression used for the barrier of the entry body.

The default setting is `True`.

Has Properties

The has properties are described on the following pages:

Is Constant

If the “has” relationship is static and the Is Constant property is set to True, the Ada Generator will create a constant declaration rather than a variable declaration.

The default setting is `False`.

Is Aliased (Ada 95)

The Is Aliased property specified that the object or component is to be defined as *aliased*.

The default setting is `False`.

Code Name

The Code Name property specifies the name for the “has” relationship in the generated code. You need to set this property only if you want the “has” relationship to be named differently than it is in the Rose model. This is especially useful when the Rose model and code are expressed in different natural languages. The value of this property should be a valid Ada identifier.

Name If Unlabeled

The Name If Unlabeled property specifies the name which the Ada Generator will use for an unlabeled “has” relationship. The string can include the variable `${supplier}`, which expands to the name of the supplier class of the “has”

relationship. For example, if class Message and class Priority are the client and the supplier, respectively, of an unlabeled “has” relationship, the string `The_${supplier}` resolves to `The_Priority`.

The default setting is `The_${supplier}`.

Record Field Implementation (Ada 95)

The Record Field Implementation property controls the definition of the field within the record type definition for the “has” relationship.

Component	The relationship will be defined as a component.
Discriminant	The relationship will be defined as a discriminant.
Access Discriminant	The relationship will be defined as an access discriminant.

The default setting is `Component`.

Record Field Name (Ada 95) / Data Member Name (Ada 83)

The Record Field Name (Ada 95) / Data Member Name (Ada 83) property specifies the name the Ada Generator outputs for the record field of a “has” relationship. The string can include the variable `${supplier}`, which expands to the name of the supplier class of the “has” relationship, and the variable `${relationship}`, which expands to the name of the “has” relationship itself.

If the variable `${relationship}` is used, and the “has” relationship is unlabeled, then the value of `${relationship}` will be the value of the property Name If Unlabeled.

The default setting is `${relationship}`.

Generate Get (Ada 95)

The Generate Get property determines whether the function is declared or suppressed by the Ada Generator.

Function	The Get operation will be declared as a function.
Do Not Create	No Get operation will be declared.

The default setting is `Function`.

Generate Access Get (Ada 95)

The Generate Access Get property determines whether the function is declared or suppressed by the Ada Generator.

Function	The Access Get operation will be declared.
Do Not Create	No Access Get operation will be declared.

The default setting is `Do Not Create`.

Get Name

The Get Name property specifies the name the Ada Generator outputs for the get accessor of a “has” relationship. The string can include the variable `${supplier}`, which expands to the name of the supplier class of the “has” relationship, and the variable `${relationship}`, which expands to the name of the “has” relationship itself.

If the variable `${relationship}` is used, and the “has” relationship is unlabeled, then the value of `${relationship}` will be the value of the property Name If Unlabeled.

The default setting is `Get_${relationship}`.

Inline Get

The Inline Get property specifies whether an inline pragma should be generated for the Get operation.

The default setting is `True`.

Generate Set (Ada 95)

The Generate Set property determines whether the procedure is declared or suppressed by the Ada Generator.

Procedure	The Set operation will be declared.
Do Not Create	No Set operation will be declared.

The default setting is `Procedure`.

Generate Access Set (Ada 95)

The Generate Set property determines whether the procedure is declared or suppressed by the Ada Generator.

Procedure	The Set operation will be declared.
Do Not Create	No Set operation will be declared.

The default setting is `Do Not Create`.

Set Name

The Set Name property specifies the name the Ada Generator outputs for the set accessor of a “has” relationship. The string can include the variable `${supplier}`, which expands to the name of the supplier class of the “has” relationship, and the variable `${relationship}`, which expands to the name of the “has” relationship itself.

If the variable `${relationship}` is used, and the “has” relationship is unlabeled, then the value of `${relationship}` will be the value of the property Name If Unlabeled.

The default setting is `Set_${relationship}`.

Inline Set

The Inline Set property specifies whether an inline pragma should be generated for the Set operation.

The default setting is `True`.

Is Constant (Ada 83)

If a “has” relationship is static, and the Is Constant property is set to `True`, the Ada Generator will create a constant declaration rather than a variable declaration.

To create a named number declaration, do not set Is Constant to `True`; rather, set the type of the attribute to constant.

To define the value of the constant or named number, use the Initial Value property.

Initial Value

The Initial Value property attaches an initial value to a field declaration, variable declaration, or constant declaration.

Variant (Ada 83)

The Variant property is used in conjunction with the Class properties Discriminant and Variant to define a class as an Ada variant record. This Variant property assigns the component for the "has" relationship to a particular variant of the variant part of the record. For example, assume that class Peripheral has the following Ada declaration:

```
type Object is record
  Unit : Device;
  Status : State;
  Line_Count : Integer;
  Cylinder : Cylinder_Index;
  Track : Track_Number;
end record;
```

Assume that type Device has the enumerated values (Printer, Disk, Drum). This declaration can be changed to a discriminated record through the following steps:

Remove the Unit "has" relationship and set the Class property Discriminant to `Unit : Device`:

```
type Object (Unit : Device) is record
  Status : State;
  Line_Count : Integer;
  Cylinder : Cylinder_Index;
  Track : Track_Number;
end record;
```

Set the Class property Variant to Unit:

```
type Object (Unit : Device) is record
  Status : State;
  Line_Count : Integer;
  Cylinder : Cylinder_Index;
  Track : Track_Number;
  case Unit is
  end case;
end record;
```

Set the Variant property for the Line_Count "has" relationship to Printer, and set the Variant property for the Track and Cylinder "has" relationships to others:

```
type Object (Unit : Device) is record
  Status : State;
  case Unit is
    when Printer =>
      Line_Count : Integer;
```

```
      when others =>
        Cylinder : Cylinder_Index;
        Track : Track_Number;
    end case;
  end record;
```

The Ada Generator will always put the *others* variant last in the variant part.

Container Implementation (Ada 95)

The Container Implementation property controls the implementation scheme for a container type by the Ada Generator.

Array	Create an unconstrained array type and access to that array type.
Generic	Use the generic unit given by the property Container Generic Name.

The default setting is `Array`.

Container Generic

The Container Generic property provides some control over the generic package instantiated to handle one-to-many “has” relationships. For example, if Container Generic is set to `List`, then the package `List_Generic` will be instantiated (if the maximum allowable cardinality of the “has” relationship is larger than 1). If Container Generic is changed to `Queue`, the package `Queue_Generic` will be instantiated.

The default setting is `List`.

Container Type

The Container Type property specifies a data type for the record field generated for a “has” relationship. The Container Type property can be set to refer to an existing container class, and the Ada Generator will use that container class instead of generating its own container class.

Container Declarations

The Container Declarations property lets you create any declarations, such as array type declarations or generic instantiations, to support the Container Type property.

Attribute Properties

The attribute properties are described on the following pages:

Initial Value

The Initial Value property attaches an initial value to an object or component.

Representation

The Representation property is used to specify one or more representation items, including pragmas. The constructs must be fully defined, including a terminating semicolon. The predefined name “attribute” yields the name of the entity, which can be used within the property definition.

For an attribute, the representation depends on whether it is static, i.e. treated as an object declaration, or non-static, treated as a component declaration. For the latter, use the syntax for a component_clause.

The placement of the representations for an attribute comes after the object declaration.

Is Constant

If the attribute is static and the Is Constant property is set to `True`, the Ada Generator will create a constant declaration rather than a variable declaration.

The default setting is `False`.

Is Aliased (Ada 95)

The Is Aliased property specified that the object or component is to be defined as *aliased*.

The default setting is `False`.

Code Name

The Code Name property specifies the name for the attribute in the generated code. You need to set this property only if you want the attribute to be named differently than it is in the Rose model. This is especially useful when the Rose model and code are expressed in different natural languages. The value of this property should be a valid Ada identifier.

Record Field Implementation (Ada 95)

The Record Field Implementation property controls the definition of the field within the record type definition for attributes of the class.

Component	The attribute will be defined as a component.
Discriminant	The attribute will be defined as a discriminant.
Access Discriminant	The attribute will be defined as an access discriminant.

The default setting is `Component`.

Record Field Name (Ada 95) / Data Member Name (Ada 83)

The Record Field Name (Ada 95) / Data Member Name (Ada 83) property specifies the name the Ada Generator outputs for the record field of an attribute. The string can include the variable `${attribute}`, which expands to the name of the label of the class attribute in the model or the name specified in the attribute's Code Name property.

The default setting is `${attribute}`.

Generate Get (Ada 95)

The Generate Get property determines whether the function is declared or suppressed by the Ada Generator.

Function	The Get operation will be declared.
Do Not Create	No Get operation will be declared.

The default setting is `Function`.

Generate Access Get (Ada 95)

The Generate Access Get property determines whether the function is declared or suppressed by the Ada Generator.

Function	The Access Get operation will be declared.
Do Not Create	No Access Get operation will be declared.

The default setting is `Do Not Create.`

Get Name

The Get Name property specifies the name the Ada Generator outputs for the get accessor of an attribute. The string can include the variable `${attribute}`, which expands to the name of the label of the class attribute in the model or the name specified in the attribute's Code Name property

The default setting is `Get_${attribute}`.

Inline Get

The Inline Get property specifies whether an inline pragma should be generated for the Get operation.

The default setting is `True`.

Generate Set (Ada 95)

The Generate Set property determines whether the procedure is declared or suppressed by the Ada Generator.

Procedure	The Set operation will be declared.
Do Not Create	No Set operation will be declared.

The default setting is `Do Not Create`.

Generate Access Set (Ada 95)

The Generate Access Set property determines whether the procedure is declared or suppressed by the Ada Generator.

Procedure	The Access Set operation will be declared.
Do Not Create	No Access Set operation will be declared.

The default setting is `Do Not Create`.

Set Name

The Set Name property specifies the name the Ada Generator outputs for the set accessor of an attribute. The string can include the variable `${attribute}`, which expands to the name of the label of the class attribute in the model or the name specified in the attribute's Code Name property.

The default setting is `Set_${attribute}`.

Inline Set

The Inline Set property specifies whether an inline pragma should be generated for the Set operation.

The default setting is `True`.

Association Role Properties

The association role properties are described on the following pages:

- *Initial Value* on page 125
- *Container Implementation (Ada 95)* on page 125
- *Container Generic* on page 125
- *Container Type* on page 125
- *Container Declarations* on page 125

Record Field Implementation

The Record Field Implementation property controls the definition of the field within the record type definition for roles of the class.

If you select:	**The action is:**
Component	(Default) The role will be defined as a component.
Discriminant	The role will be defined as a discriminant.
Access Discriminant	The role will be defined as an access discriminant.

Is Constant

If the role is static and the Is Constant property is set to `True`, the Ada Generator will create a constant declaration rather than a variable declaration.

The default setting is `False`.

Is Aliased (Ada 95)

The Is Aliased property specified that the object or component is to be defined as *aliased*.

The default setting is `False`.

Code Name

The Code Name property specifies the name for the association role in the generated code. You need to set this property only if you want the association role to be named differently than it is in the Rose model. This is especially useful when the Rose model and code are expressed in different natural languages. The value of this property should be a valid Ada identifier.

Name If Unlabeled

The Name If Unlabeled property specifies the name to be used for an unlabeled role. The Ada Generator uses the name of the role to construct names for the corresponding component and get and set operations. If the role is not named, the Ada Generator uses this property to determine the name of the role.

When the Ada Generator needs the name of the role to generate a name for a component or a get or set operations, `${targetClass}` expands to the name of the association class or the association if there is one. Otherwise it expands to the name of the supplier class. If `${association}` is used in the Name If Unlabeled property, it expands to the name of the association.

The default setting is `The_${targetClass}`.

Record Field Name (Ada 95) / Data Member Name (Ada 83)

The Record Field Name (Ada 95) / Data Member Name (Ada 83) property specifies the name the Ada Generator outputs for the record field for an association role. The string can include the variable `${target}`, which expands to the name of the target of the component. If there is an association (class), this is the name of the association (class). If there is not an association (class), this is the name of the supplier role.

The default setting is `${target}`.

Generate Get (Ada 95)

The Generate Get property determines whether the function is declared or suppressed by the Ada Generator.

Function	The Get operation will be declared.
Do Not Create	No Get operation will be declared.

The default setting is `Function`.

Generate Access Get (Ada 95)

The Generate Access Get property determines whether the function is declared or suppressed by the Ada Generator.

Function	The Access Get operation will be declared.
Do Not Create	No Access Get operation will be declared.

The default setting is `Do Not Create`.

Get Name

The Get Name property specifies the name the Ada Generator outputs for the get accessor of an association role. The string can include the variable `${target}`, which expands to the name of the target of the component. If there is an association class, this is the name of the association class. If there is not an association class, this is the name of the supplier role.

The default setting is `Get_${target}`.

Inline Get

The Inline Get property specifies whether an inline pragma should be generated for the Get operation.

The default setting is `True`.

Generate Set (Ada 95)

The Generate Set property determines whether the procedure is declared or suppressed by the Ada Generator.

Procedure	The Set operation will be declared.
Do Not Create	No Set operation will be declared.

The default setting is `Procedure`.

Set Name

The Set Name property specifies the name the Ada Generator outputs for the set accessor of an association role. The string can include the variable `${target}`, which expands to the name of the target of the component. If there is an association class, this is the name of the association class. If there is not an association class, this is the name of the supplier role.

The default setting is `Set_${target}`.

Inline Set

The Inline Set property specifies whether an inline pragma should be generated for the Set operation.

The default setting is `True`.

Initial Value

The Initial Value property attaches an initial value to a field declaration.

Container Implementation (Ada 95)

The Container Implementation property controls the implementation scheme for a container type by the Ada Generator.

Array	Create an unconstrained array type and access to that array type.
Generic	Use the generic unit given by the property Container Generic Name.

The default setting is `Array`.

Container Generic

The Container Generic property provides some control over the generic package instantiated to handle one-to-many association roles. For example, if Container Generic is set to `List`, then the package `List_Generic` will be instantiated (if the maximum allowed cardinality of the “has” relationship is larger than 1). If Container Generic is changed to `Queue`, the package `Queue_Generic` will be instantiated.

The default setting is `List`.

Container Type

The Container Type property specifies a data type for the record field generated for an association role. The Container Type property can be set to refer to an existing container class, and the Ada Generator will use that container class instead of generating its own container class.

Container Declarations

The Container Declarations property lets you create any declarations, such as array type declarations or generic instantiations, to support the Container Type property.

Association Properties

The association properties are described on the following pages:

- *Name If Unlabeled* on page 126
- *Generate Get (Ada 95)* on page 126
- *Get Name* on page 126

- *Inline Get* on page 127
- *Generate Set (Ada 95)* on page 127
- *Set Name* on page 127
- *Inline Set* on page 127
- *Generate Associate* on page 127
- *Associate Name* on page 128
- *Inline Associate* on page 128
- *Generate Dissociate* on page 128
- *Dissociate Name* on page 128
- *Inline Dissociate* on page 128

Name If Unlabeled

The Name If Unlabeled property specifies the name to be used for an unlabeled association. The Ada Generator uses the name of the association to construct names for the corresponding component and get and set operations. If the association is not named, the Ada Generator uses this property to determine the name of the association.

When the Ada Generator needs the name of the association to generate a name for a component or a get or set operations, `${targetClass}` expands to the name of the association class or the association if there is one. Otherwise it expands to the name of the supplier class.

The default setting is `The_${targetClass}`.

Generate Get (Ada 95)

The Generate Get property determines whether the function is declared or suppressed by the Ada Generator.

Function	The Get operation will be declared as a function.
Do Not Create	No Get operation will be declared.

The default setting is `Function`.

Get Name

The Get Name property specifies the name the Ada Generator outputs for the get accessor of an association class. The string can include the variable `${association}`, which expands to the name of the association. If the association is unnamed, then the name of the association class is used.

The default setting is `Get_${association}`.

Inline Get

The Inline Get property specifies whether an inline pragma should be generated for the Get operation.

The default setting is `False`.

Generate Set (Ada 95)

The Generate Set property determines whether the procedure is declared or suppressed by the Ada Generator.

Procedure	The Set operation will be declared.
Do Not Create	No Set operation will be declared.

The default setting is `Procedure`.

Set Name

The Set Name property specifies the name the Ada Generator outputs for the Set accessor of an association class. The string can include the variable `${association}`, which expands to the name of the association. If the association is unnamed, then the name of the association class is used.

The default setting is `Set_${association}`.

Inline Set

The Inline Set property specifies whether an inline pragma should be generated for the Set operation.

The default setting is `False`.

Generate Associate

The Generate Association property determines whether the procedure is declared or suppressed by the Ada Generator.

Procedure	The Associate operation will be declared.
Do Not Create	No Associate operation will be declared.

The default setting is `Procedure`.

Associate Name

The Associate Name property specifies the name the Ada Generator outputs for the Associate operation of an association.

The default setting is `Associate`.

Inline Associate

The Inline Associate property specifies whether an inline pragma should be generated for the Associate operation.

The default setting is `False`.

Generate Dissociate

The Generate Dissociate property determines whether the procedure is declared or suppressed by the Ada Generator.

Procedure	The Dissociate operation will be declared.
Do Not Create	No Dissociate operation will be declared.

The default setting is `Procedure`.

Dissociate Name

The Dissociate Name property specifies the name the Ada Generator outputs for the Dissociate operation of an association.

The default setting is `Dissociate`.

Inline Dissociate

The Inline Dissociate property specifies whether an inline pragma should be generated for the Dissociate operation.

The default setting is `False`.

UML Package Properties

Directory

The Directory property specifies the UML package. This property defaults to `AUTO GENERATE`.

Module Spec Properties

The model spec properties are described on the following pages:

Generate

The Generate property specifies whether or not the Ada Generator will generate a code file for the module spec.

This property allows you to prevent code from ever being generated for a module, such as modules in third party libraries, even if it is selected when the Ada Generator is invoked.

The default value is `True`.

Copyright Notice

The Copyright Notice property contains text that is placed in a comment block at the beginning of the Ada specification file created by the Ada Generator for the module spec. This property can be used to include copyright notices or project identification information at the beginning of a module. The text in the Copyright Notice property is preceded by comment delimiters ("--"), so they do not need to be included in the text of the property itself.

Return Type

The Return Type property specifies the subtype indication for the return value of a subprogram module. For example, if the Return Type property is set to Calendar.Time for a subprogram specification module named Current_Time, the Ada Generator will output:

```
function Current_Time return Calendar.Time;
```

If Return Type is set to blank, the Ada Generator will output:

```
procedure Current_Time;
```

The Return Type property is ignored when the module spec is not a subprogram specification.

Generic Formal Parameters

The Generic Formal Parameters property is used to specify the generic formal part of a generic module spec. For example, if the Generic Formal Parameters property is set to `type Item is private` for a generic package specification module named Stack, the Ada Generator will output:

```
generic
    type Item is private;
package Stack is
    ...
end Stack;
```

If `Size : in Positive` is added to Generic Formal Parameters, the Ada Generator will output:

```
generic
    type Item is private;
    Size : in Positive;
package Stack is
    ...
end Stack;
```

The Generic Formal Parameters property is ignored when the module spec is not a generic. Additional generic formal parameters may be added to the generic formal part if a generic class is assigned to the module, because the generic formal parameters of the generic class will be merged with those of the module.

Additional Withs

The Additional Withs property specifies additional with clauses to be included in the context clause of the module spec. For example, if the Additional Withs property is set to `Text_Io` for a subprogram specification module named `Quadratic_Equation`, the Ada Generator will output:

```
-- Additional Withs:
with Text_Io;
procedure Quadratic_Equation;
```

If `Real_Operations` is added to Additional Withs, the Ada Generator will output:

```
-- Additional Withs:
with Text_Io;
with Real_Operations;
procedure Quadratic_Equation;
```

Only the simple names of the library units should be listed in the Additional Withs property, with one library unit per line.

Module Body Properties

The module body properties are described on the following pages:

Is Subunit

The Is Subunit property specifies whether the component is a subunit. For a component to be considered a subunit, the component name must be an expanded name, composed of the parent unit name and the name of the subunit. A dependency must be established between the subprogram body component and the package body component. The subunit name must correspond to a subprogram within an associated class of the parent component, where the property Subprogram Implementation is set to "Separate".

The default setting is `False`.

Is Private (Ada 95)

The Is Private property controls whether the package is private or not.

The default setting is `False`.

Generate

The Generate property specifies whether or not the Ada Generator will generate a code file for the module body.

This property allows you to prevent code from ever being generated for a module, such as modules in third party libraries, even if it is selected when the Ada Generator is invoked.

The default value is `True`.

Copyright Notice

The Copyright Notice property contains text that is placed in a comment block at the beginning of the Ada body file created by the Ada Generator for the module body. This property can be used to include copyright notices or project identification

information at the beginning of a module. The text in the Copyright Notice property is preceded by comment delimiters (“--”), so they do not need to be included in the text of the property itself.

Return Type

The Return Type property specifies the subtype indication for the return value of a subprogram module. For example, if the Return Type property is set to `Calendar.Time` for a subprogram body module named `Current_Time`, the Ada Generator will output:

```
function Current_Time
    return Calendar.Time is ...
```

If Return Type is set to blank, the Ada Generator will output:

```
procedure Current_Time is ...
```

The Return Type property is ignored when the module body is not a subprogram body.

Additional Withs

The Additional Withs property specifies additional with clauses to be included in the context clause of the module body. For example, if the Additional Withs property is set to `Text_Io` for a subprogram body module named Quadratic_Equation, the Ada Generator will output:

```
-- Additional Withs:
with Text_Io;
procedure Quadratic_Equation is …
```

If `Real_Operations` is added to Additional Withs, the Ada Generator will output:

```
-- Additional Withs:
with Text_Io;
with Real_Operations;
procedure Quadratic_Equation is …
```

Only the simple names of the library units should be listed in the Additional Withs property, with one library unit per line.

Index

A

B

C

D

E

F

G

H

I

K

M

N

O

P

Q

R

S

T

U

V

W