Rational Rose 2000e Forward and Reverse Engineering with Ada 83

Part Number: 800-023221-000 Rev A
Revision 2.2, March 2000, for Rational Rose 2000e

This document is subject to change without notice.

Note the Reader's Comments form at the end of this book, which requests your evaluation to assist Rational in preparing future documentation.

Contents

Chapter 1 OOD and Ada 83 1

Mapping Classes 1
- Standard Classes 1
- Utilities 2
- Parameterized Classes 3
- Bound Classes 3

Mapping Relationships 3
- Dependency Relationships 3
- Has Relationships 3
- Generalization Relationships (Inheritance) 4
- Association Relationships 5

Achieving Polymorphism with Ada 5

Unmapped Elements for Ada 5

Chapter 2 Ada Code Generation 7

What is the Ada Generator? 7

Basic Steps for Iterative Code Development 8
- Overview 8
- The Generated Files 9
- The Basic Code Contents 9
- Evaluating the Generated Code 10
- Completing the Implementation of the Generated Code . 11
- Regenerating Code 11

Refining the Subsystem and View Structure 12

Determining the Directory for an Ada File 12
Mapping Classes and Modules to Ada Units 13
Specifying Filenames 14

Refining Class Definitions 14
Standard Operations 15
User-Defined Operations 15
Get and Set Operations 15
Inherited Operations 16
Record Fields and Object Declarations 16

Specifying Additional Ada Unit Contents 17
Adding Structured Comments 17
Adding With Clauses 17
Adding Global Declarations 18

Chapter 3 Reverse Engineering from Apex 19

Basic Operations 19
Creating the Model File 20
Displaying the Model 20

Dialog Box Options 21

How Ada 83 Is Represented In a Class Diagram 22
Mapping Package Specifications 22
Mapping Type Declarations 22
Details of a Has Relationship 23
Mapping Subprogram Declarations 23
Mapping Object Declarations 24
Mapping With Clauses 24
Ada Constructs not Mapped 24
Special Handling for Subsystems in the $APEX_BASE Directory 25

Chapter 4 Ada 83 Code Generation Properties 27

Design Properties 28
Spec File Extension 28
Spec File Backup Extension 28

Spec File Temporary Extension 29
Body File Extension 29
Body File Backup Extension 29
Body File Temporary Extension 29
Create Missing Directories 30
Generate Bodies 30
Generate Standard Operations 30
Implicit Parameter 31
Stop On Error 31
Error Limit 31
File Name Format 31
Directory 32

Class Properties 33
Code Name 34
Class Name 34
Class Access 34
Implementation Type 35
Is Subtype 35
Polymorphic Unit 35
Handle Name 35
Handle Access 36
Discriminant 36
Variant 36
Enumeration Literal Prefix 37
Record Field Prefix 37
Generate Standard Operations 37
Implicit Parameter 38
Class Parameter Name 38
Default Constructor Kind 38
Default Constructor Name 39
Inline Default Constructor 39
Copy Constructor Kind 39
Inline Copy Constructor 40
Destructor Name 40
Inline Destructor 40

Class Equality Operation 41
Handle Equality Operation 41
Inline Equality 41
Is Task 41

Operation Properties 42
Code Name 42
Subprogram Implementation 42
Class Parameter Mode 43
Inline 43
Entry Code 43
Exit Code 43

Has Properties 43
Code Name 44
Name If Unlabeled 44
Data Member Name 45
Get Name 45
Inline Get 45
Set Name 45
Inline Set 46
Is Constant 46
Initial Value 46
Variant 47
Container Generic 48
Container Type 48
Container Declarations 48

Attribute Properties 49
Code Name 49
Data Member Name 49
Get Name 49
Inline Get 50
Set Name 50
Inline Set 50

Association Role Properties 50
Code Name 51

Name If Unlabeled 51
Data Member Name 51
Get Name 52
Inline Get 52
Set Name 52
Inline Set 52
Initial Value 52
Container Generic 53
Container Type 53
Container Declarations 53

Association Properties 53
Name If Unlabeled 54
Get Name 54
Inline Get 55
Set Name 55
Inline Set 55
Generate Associate 55
Associate Name 55
Inline Associate 56
Generate Dissociate 56
Dissociate Name 56
Inline Dissociate 56

UML Package Properties 56
Directory 56

Module Spec Properties 56
Generate 57
Copyright Notice 57
Return Type 57
Generic Formal Parameters 58
Additional Withs 58

Module Body Properties 59
Generate 59
Copyright Notice 59
Return Type 60

Additional Withs 60

Index 61

Chapter 1

OOD and Ada 83

This document contains the following topics:

- "Mapping Classes" on page 1
- "Mapping Relationships" on page 3
- "Achieving Polymorphism with Ada" on page 5
- "Unmapped Elements for Ada" on page 5

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" on page 1
- "Utilities" on page 2
- "Parameterized Classes" on page 3
- "Bound Classes" on page 3

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" on page 3
- "Has Relationships" on page 3
- "Generalization Relationships (Inheritance)" on page 4
- "Association Relationships" on page 5

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

Chapter 2

Ada Code Generation

This chapter contains the following topics:

- "What is the Ada Generator?" on page 7
- "Basic Steps for Iterative Code Development" on page 8
- "Refining the Subsystem and View Structure" on page 12
- "Refining Class Definitions" on page 14
- "Specifying Additional Ada Unit Contents" on page 17

What is the Ada Generator?

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

This section contains the following topics:

- “Overview” on page 8
- “The Generated Files” on page 9
- “The Basic Code Contents” on page 9
- “Evaluating the Generated Code” on page 10
- “Completing the Implementation of the Generated Code” on page 11
- “Regenerating Code” on page 11

Overview

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 the steps for generating Ada units from a Rose model.

By default, code is generated in the current working view (determined initially when you start Rose/Ada 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.

In order to generate Ada 83 code, you must first activate the 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 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 83.

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

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 83** 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.
- 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.

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/Ada 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/Ada after you have started refining the implementation of the code. Note that some of the code regions that Rose/Ada 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. You must use the code regions produced by the Ada Generator.

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

This section contains the following topics:

- “Determining the Directory for an Ada File” on page 12
- “Mapping Classes and Modules to Ada Units” on page 13
- “Specifying Filenames” on page 14

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 Apex View
- The UML package properties Apex Subsystem and Apex View

The directory for a module is based on the concatenation of the project Directory property, and the UML package’s Apex Subsystem and Apex View 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 Apex Subsystem and Apex View properties for the component UML package. If Apex Subsystem is blank, the Apex 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 Apex Subsystem and Apex View properties, plus the project Directory property. If the default Apex Subsystem 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 have two options. The first option involves only the class diagram:

1. Create a uses relationship where the client class will become the Ada unit, and the supplier class will be declared within the Ada unit.
2. Change the name of the relationship to the keyword **decl**.

Your second option is to assign two or more classes to the same module:

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

This section contains the following topics:

- "Standard Operations" on page 15
- "User-Defined Operations" on page 15
- "Get and Set Operations" on page 15
- "Inherited Operations" on page 16
- "Record Fields and Object Declarations" on page 16

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 “Ada 83 Code Generation Properties” on page 27 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" on page 17
- "Adding With Clauses" on page 17
- "Adding Global Declarations" on page 18

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.

Chapter 3

Reverse Engineering from Apex

Rose can analyze Ada 83 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: *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.*

This chapter contains the following topics:

- "Basic Operations" on page 19
- "Dialog Box Options" on page 21
- "How Ada 83 Is Represented In a Class Diagram" on page 22

Basic Operations

Reverse Engineer can create both class diagrams and component diagrams. Class diagrams will show the relationships among Ada specifications, types and objects. Component diagrams come in two forms: 1) An Ada unit diagram, which displays the "with" structure of the Ada units in a program, independent of subsystem structure; and 2) A subsystem diagram, which displays the import structure of the subsystems you specify. Within each subsystem is a display of the "with" structure of the Ada units in that view.

This section contains the following topics:

- "Creating the Model File" on page 20
- "Displaying the Model" on page 20

Creating the Model File

No matter which type of diagram you want, Reverse Engineer will always generate 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 21.

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 **Tools: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 **Tools: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 **Tools: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 **Tools: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 Engineer will include 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 will be included in the petal file.

Petal File Name

By default, Reverse Engineer will create 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 will create a class diagram of the units or views selected.

Include Modules

If you select this button, Reverse Engineer will create a component diagram of the units or views selected. If neither Include Classes or Include Modules is selected, a component diagram showing just the import structure of the subsystems will be created.

How Ada 83 Is Represented In a Class Diagram

Reverse Engineer uses various algorithms to map Ada constructs to the UML notation, based primarily on the mapping described in Chapter 1.

This section contains the following topics:

- "Mapping Package Specifications" on page 22
- "Mapping Type Declarations" on page 22
- "Details of a Has Relationship" on page 23
- "Mapping Subprogram Declarations" on page 23
- "Mapping Object Declarations" on page 24
- "Mapping With Clauses" on page 24
- "Ada Constructs not Mapped" on page 24
- "Special Handling for Subsystems in the $APEX_BASE Directory" on page 25

Mapping Package Specifications

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

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

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 private record declaration, and the subprogram includes a parameter or 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 represents 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. If the object is a constant, the IsConstant property is set to True. If the object is a named number, Type field of the relationship is set to “constant.”

If the package where the object is declared contains at least one class, and all subprograms map 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

As Reverse Engineer does its analysis, it tracks the with’s that would be automatically generated by “has” relationships in package specification. The remaining with clauses, those that are used for parameter types and return types, become dependency relationships in the model.

Ada Constructs not Mapped

There are several constructs that are ignored by Reverse Engineer and not included in the Rose model:

- Representation clauses
- Use clauses
- Incomplete Types
- Pragmas
- Library-level subprogram specifications

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 Engineer examines only the type declarations in these subsystems, and does not evaluate attributes or operations.

Chapter 4

Ada 83 Code Generation Properties

This chapter contains the following topics:

- "Design Properties" on page 28
- "Class Properties" on page 33
- "Operation Properties" on page 42
- "Has Properties" on page 43
- "Attribute Properties" on page 49
- "Association Role Properties" on page 50
- "Association Properties" on page 53
- "UML Package Properties" on page 56
- "Module Spec Properties" on page 56
- "Module Body Properties" on page 59

Design Properties

This section contains the following topics:

- "Spec File Extension" on page 28
- "Spec File Backup Extension" on page 28
- "Spec File Temporary Extension" on page 29
- "Body File Extension" on page 29
- "Body File Backup Extension" on page 29
- "Body File Temporary Extension" on page 29
- "Create Missing Directories" on page 30
- "Generate Bodies" on page 30
- "Generate Standard Operations" on page 30
- "Implicit Parameter" on page 31
- "Stop On Error" on page 31
- "Error Limit" on page 31
- "File Name Format" on page 31
- "Directory" on page 32

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 “Spec File Backup Extension” on page 28) 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 filename 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 False.

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, a UML package Directory property, or a module File Name 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 file name 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

This section contains the following topics:

- “Code Name” on page 34
- “Class Name” on page 34
- “Class Access” on page 34
- “Implementation Type” on page 35
- “Is Subtype” on page 35
- “Polymorphic Unit” on page 35
- “Handle Name” on page 35
- “Handle Access” on page 36
- “Discriminant” on page 36
- “Variant” on page 36
- “Enumeration Literal Prefix” on page 37
- “Record Field Prefix” on page 37
- “Generate Standard Operations” on page 37
- “Implicit Parameter” on page 38
- “Class Parameter Name” on page 38
- “Default Constructor Kind” on page 38
- “Default Constructor Name” on page 39
- “Inline Default Constructor” on page 39
- “Copy Constructor Kind” on page 39
- “Inline Copy Constructor” on page 40
- “Destructor Name” on page 40
- “Inline Destructor” on page 40
- “Class Equality Operation” on page 41
- “Handle Equality Operation” on page 41
- “Inline Equality” on page 41
- “Is Task” on page 41

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.

Class Name

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

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

if Class Name is set to **`Object`**, the Ada Generator will output:

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

You have the option of setting the Class Name property to ${class}, where the Ada Generator will use the name of the Rose class for the name of the type.

Class Access

The Class Access property controls the definition of the Ada type used by the Ada Generator to represent a Rose class.

Private	The type declared by the Ada Generator to represent the Rose class will be a private type. The corresponding complete type declaration will appear in the private part of the Ada specification.
Limited Private	The type will be a limited private type. The corresponding complete type declaration will appear in the private part of the Ada specification.
Public	The type will be a public type. Only classes with the Implementation Type property set can be public.
Do Not Create	No type declaration will be output by the Ada Generator.

Implementation Type

The Implementation Type property allows a Rose class to be defined as something other than a private or limited private record type. For example, if Implementation Type is set to range 1 .. 500, the Ada Generator will output (assuming Class Name is set to Object):

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

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

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

Is Subtype

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

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

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

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

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

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.

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_”.

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.

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.

Class Parameter Name

All operations of a class have as an implicit parameter a class object. The Class Parameter Name property specifies the formal parameter name used by the Ada Generator for this class object. For example, if the Class Parameter Name is set to This (and all other properties have their default values), the class destructor will be declared as:

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

If Class Parameter Name is changed to **`The_Object`**, the class destructor would be:

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

The Class Parameter Name 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.

Default Constructor Kind

The Default Constructor Kind 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 Default Constructor Kind is set to Function, the declaration output by the Ada Generator will be of the form:

```
function Create return Object;
```

If Default Constructor Kind is set to Procedure, the declaration output by the Ada Generator will be of the form:

```
procedure Create (This : in out Object);
```

The properties Class Name, Class Parameter Name 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.

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 (and all other properties have their default values), the Ada Generator will output:

```
function Create return Object;
```

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

```
function New_Item return Object;
```

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.

Copy Constructor Kind

The Copy Constructor Kind 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 Copy Constructor Kind is set to Function, the declaration output by the Ada Generator will be of the form:

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

If Copy Constructor Kind is set to Procedure, the declaration output by the Ada Generator will be of the form:

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

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.

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.

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 (and all other properties have their default values), the Ada Generator will output:

```
procedure Free (This in out Object);
```

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

```
procedure Deallocate_Item (This in out Object);
```

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.

Class Equality Operation

The Class Equality Operation property controls the designator of the equality function declared by the Ada Generator to compare class objects. For example, if the Class 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 Object) return Boolean;
```

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

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

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

Handle Equality Operation

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

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

Operation Properties

This section contains the following topics:

- "Code Name" on page 42
- "Subprogram Implementation" on page 42
- "Class Parameter Mode" on page 43
- "Inline" on page 43
- "Entry Code" on page 43
- "Exit Code" on page 43

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

- If Subprogram Implementation is set to Body, a normal body is generated.
- If Subprogram Implementation is set to Separate, a stub is generated instead of a normal body.
- If Subprogram Implementation is set to Spec, no body is generated. 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.

Class Parameter Mode

The Class Parameter Mode property determines the mode of the class parameter for a particular operation.

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

The default setting is In Out.

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.

Has Properties

This section contains the following topics:

- "Code Name" on page 44
- "Name If Unlabeled" on page 44
- "Data Member Name" on page 45
- "Get Name" on page 45
- "Inline Get" on page 45
- "Set Name" on page 45
- "Inline Set" on page 46
- "Is Constant" on page 46
- "Initial Value" on page 46
- "Variant" on page 47
- "Container Generic" on page 48
- "Container Type" on page 48
- "Container Declarations" on page 48

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. In the last example, the string The_${supplier}_Of_The_Message resolves to The_Priority_Of_The_Message.

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

Data Member Name

The Data Member Name property specifies the name the Ada Generator outputs for the component corresponding to 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}”.

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.

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

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

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 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 component 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

This section contains the following topics:

- “Code Name” on page 49
- “Data Member Name” on page 49
- “Get Name” on page 49
- “Inline Get” on page 50
- “Set Name” on page 50
- “Inline Set” on page 50

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.

Data Member Name

The Data Member Name property specifies the name the Ada Generator outputs for the component corresponding to 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}.

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.

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

This section contains the following topics:

- “Code Name” on page 51
- “Name If Unlabeled” on page 51
- “Data Member Name” on page 51
- “Get Name” on page 52
- “Inline Get” on page 52
- “Set Name” on page 52
- “Inline Set” on page 52
- “Initial Value” on page 52
- “Container Generic” on page 53
- “Container Type” on page 53
- “Container Declarations” on page 53

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 NameIfUnlabeled property, it expands to the name of the association.

The default setting is The_${targetClass}.

Data Member Name

The Data Member Name property specifies the name the Ada Generator outputs for the component corresponding to 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. If there is not an association class, this is the name of the supplier role.

The default setting is ${target}.

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. 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.

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. 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 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 component 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.

Association Properties

This section contains the following topics:

- "Name If Unlabeled" on page 54
- "Get Name" on page 54
- "Inline Get" on page 55
- "Set Name" on page 55
- "Inline Set" on page 55
- "Generate Associate" on page 55
- "Associate Name" on page 55
- "Inline Associate" on page 56
- "Generate Dissociate" on page 56
- "Dissociate Name" on page 56
- "Inline Dissociate" on page 56

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}.

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.

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 subsystem. This property defaults to “AUTO GENERATE”.

Module Spec Properties

This section contains the following topics:

- "Generate" on page 57
- "Copyright Notice" on page 57
- "Return Type" on page 57
- "Generic Formal Parameters" on page 58
- "Additional Withs" on page 58

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

This section contains the following topics:

- "Generate" on page 59
- "Copyright Notice" on page 59
- "Return Type" on page 60
- "Additional Withs" on page 60

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

abstract classes 5

access
- has relationship 23
- provide to components 15

Ada 83
- code generation properties 28
- representation 22

Ada code
- traverse to 20

Ada constructs
- not mapped 24

Ada file
- determine directory 12

Ada Generator 7

Ada type used for By Reference class instances 36

Ada unit
- diagram 19
- generate from Rose model 7
- map to class and module 13
- specify additional contents 17

add
- global declarations 18
- structured comments 17
- uses relationship 18
- With clauses 17

Add Classes command 22

Add-In Manager 8

Additional Withs property 58, 60
- insert with clauses 18

Apex
- generate Rose model 19
- reverse engineer 19
- start Rose 20

APEX_BASE subsystems 25

As Booch command 3

Associate Name property 55

Associate operation
- inline pragma 56
- name 55

association class
- get accessor 54
- set accessor 55

association classes 5

Association Properties 53

Association Relationships 5

association role
- get accessor 52
- name of corresponding component 51
- set accessor 52

specify name 51
Association Role properties 50
attribute name
specify 49
Attribute Properties 49

B

backup specify
backup file name extension 28, 29
basic operations
reverse engineering 19
bidirectional associations 5
bodies
generate 30
Body File backup extension property 29
Body File extension property 29
Body File temporary extension property 29
body files
create 30
Bound Classes 3
bounded containers 17
Browse
Browse Spec 20
Component Diagram 20
Browse Spec command 20
By Reference
determine name of type 35
by-reference has relationship 4
by-value has relationship 4

C

class
define as task type 41
define as variant record 47
definition 1
map to Ada unit 13
refine definitions 14
Class Access property 34
class constructor
kind of subprogram 39
class constructor subprogram 38
simple name 39
class destructor subprogram
simple name 40
class diagram 19
create 21
format 20
how Ada 83 is represented 22
Class Equality Operation property 41
class handles 41
equality function 41
class name
specify in code 34
Class Name property 34
class objects
equality function 41
Class Operations 2
class parameter
mode 43
Class Parameter Mode property 43
Class Parameter Name property 38
Class properties 33

clauses
add With 17
closure
include 21
code
complete 11
content of generated 9
insert at beginning 43
insert at end 43
regenerate 11
code development
steps 8
code generation properties 28
code generator 7
Code Name property 34, 42, 44, 49, 51
comments
add structured 17
insert at beginning 43
insert at end 43
complete
generated code 11
component diagram 19
create 21
display 20
format 20
Component Diagram command 20
component type 16
complex 16
simple 16
component/discriminant identifier prefix 37
Container Declarations property 48, 53
Container Generic property 48, 53
Container Type property 48, 53
Copy Constructor
inline pragma 40
copy constructor 15
Copy Constructor Kind property 39
copyright notice
include in Ada body 59
include in Ada spec 57
Copyright Notice property 57, 59
create
Ada body files 30
class diagram 21
component diagram 21
declarations to support Container Type 48
missing directories 30
model file 20
Create Missing Directories property 30

D

Data Member Name property 45, 49, 51
decl
associate type with enclosing package 23
declaration
initial value 46
type mapping 22
default constructor 15
Default Constructor Kind property 38
Default Constructor Name property 39
default language 8
define
class as task type 41

class as variant record 47
Rose class as other type 35
subtype declaration 35
definitions
refine class 14
Dependency Relationships 3
design properties 28
Destructor
inline pragma 40
destructor 15
Destructor Name property 40
determine
directory for Ada file 12
dialog box options
Reverse Engineer dialog box options 21
directory
create missing 30
determine 12
Directory property 32, 56
Discriminant property 36
display
classes and utilities in view 20
component diagram 20
implementation types 22
model 20
units in view 20
Dissociate Name property 56
Dissociate operation
inline pragma 56
name 56

E

Entry Code property 43
Enumeration Literal Prefix property 37
enumeration literal values
specify prefix 37
equality function 41
class handles 41
class objects 41
equality operation 15
Equality operations
inline pragma 41
Error Limit property 31
error threshold 31
evaluate
generated code 10
exception declaration 24
exclude
views or units 21
Exclude Views/Units with prefix option 21
Exit Code property 43
extension
Body file 29
Body file backup 29
Body file temporary 29
Spec file 28, 29

F

field declaration
initial value 52
File Name Format property 31
filename
specify 14
format
class diagram 20

component diagram 20
friendship 5

G

Generalization Relationships (Inheritance) 4
generate
Ada units from Rose model 7
code file for module body 59
code file for module spec 57
Rose model from Apex 19
standard operations 30
standard operations for class 37
Generate Associate property 55
Generate Bodies property 30
Generate Dissociate property 56
Generate property 57, 59
Generate Standard Operations property 30, 37
generated code
content 9
evaluation 10
implement package bodies 11
generated files 9
generation
directory and file names 31
Generic Formal Parameters property 58
generic instantiation
bound class 3
generic package
parameterized class 3
get accessor name 45
get accessor of association class 54
get accessor of association role 52
get accessor of attribute 49
Get Name property 45, 49, 52, 54
Get operation
inline pragma 45, 50, 52, 55
get operations 15
global declarations
add 18

H

Handle Access property 36
Handle Equality Operation property 41
Handle Name property 23, 35
Has Properties 43
has relationship
component name 45
constant declaration 46
details 23
generic package instantiation 48
get accessor name 45
multiplicity and access 23
name in code 44
set accessor name 45
specify data type 48
unlabeled 44
has Relationships 3

I

Implementation Type property 22, 35
implementation types 22
implicit class parameter
generate 38
Implicit Parameter property 31, 38
include

closure 21
copyright notice in Ada body 59
copyright notice in Ada spec 57
views or units 21
Include Classes option 21
Include Closure of Views/Units option 21
Include Modules option 21
Include only Views/Units selected option 21
Include Views/Units with prefix option 21
inheritance 4
inherited operations 16
Initial Value property 46, 52
Inline Associate property 56
Inline Copy Constructor property 40
Inline Default Constructor property 39
Inline Destructor property 40
Inline Dissociate property 56
Inline Equality property 41
Inline Get property 45, 50, 52, 55
inline pragma 43
Copy Constructor 40
for Destructor 40
generate 39
Inline property 43
Inline Set property 46, 50, 52, 55
insert
comments/code at subprogram beginning 43
comments/code at subprogram end 43
Is Constant property 46
Is Subtype property 35
Is Task property 41
iterative code development 8

L

Layout Diagram command 20

M

map
classes and modules to Ada units 13
object declarations 24
package specifications 22
subprogram declarations 23
type declarations 22
with clauses 24
Mapping Classes 1
Mapping Relationships 3
metaclasses 5
model
display 20
generate from Apex 19
model file
create 20
module
map to Ada unit 13
module body
generate code for 59
Module Body Properties 59
module spec
generate code for 57
generic formal part 58
Module Spec Properties 56
multiple inheritance 5

multiplicity
    has relationship 23

N

name
    component corresponding to attribute 49
    has relationship in code 44
Name If Unlabeled property 44, 51, 54

O

object
    declaration 24
    declarations 16
    definition 2
    map declarations 24
object oriented development 1
OOD 1
operation name
    specify in generated code 42
Operation properties 42

P

package
    differences in UML and Ada 1
package specifications
    mapping 22
Parameterized Classes 3
Petal File Name option 21
polymorphic class 35
Polymorphic Unit property 35
Polymorphism with Ada 5
prefix
    exclude views/units 21
    include views/units 21
preserved code region 11
private record declaration 9
procedure
    declared or suppressed 55, 56
project directory
    specify 32
properties
    design 28

Q

Query
    Add Classes 22

R

Rational Apex
    reverse engineer 19
Record Field Prefix property 37
record fields 16
reference
    non-generated units 18
refine
    class definitions 14
    subsystem and view structure 12
regenerate code 11
Return Type property 57, 60
Reverse Engineer command 20
reverse engineering
    Ada 83 code 19
    basic engineering 19
Rose
    Ada
        Reverse Engineer 20

Start Rose 20
rose_ada.mdl 20
rename 21

S

set accessor
association class 55
association role 52
attribute 50
name 45
Set Name property 45, 50, 52, 55
Set operation
inline pragma 46, 50, 52, 55
set operations 15
single variant for discriminated record 36
Spec File Backup extension property 28, 29
Spec File extension property 28
specifications
package mapping 22
specify
Ada type name used for Rose class 34
Ada type used to represent Rose class 34
additional Ada unit contents 17
attribute name 49
body backup file name extension 29
body file name extension 29
body temporary file name extension 29
class name in generated code 34
component/discriminant identifier prefix 37
discriminant of Ada type 36
enumeration literal value prefix 37
file name extension 28
filename 14
formal parameter name for class object 38
generic formal part of generic module spec 58
operation name in generated code 42
project directory 32
return type 57
return type of subprogram module 60
Standard Classes 1
standard operations 15
class 2
generate 30, 37
start
Rose from Apex 20
Start Rose command 20
Stop On Error property 31
structured comments
add 17
subprogram
declare as class constructor 38
map declarations 23
subprogram body
control code generation 42
Subprogram Implementation property 42
subsystem
diagram 19
refine 12
specify 56

subsystems
APEX_BASE 25
subtype declaration
define 35

T

Tools
Layout Diagram 20
traverse
to Ada source code 20
type declaration
mapping 22

U

UML notation
mapping to Ada 1
UML Package Properties 56
unbounded containers 17
unidirectional associations 5
unlabeled association
name 54
unlabeled has relationship 44
unlabeled role
name 51
unmapped elements 5
use relationship
add 18
user-defined operations 15
Utilities 2

V

Variant property 36, 47
variant record 47
view
As Booch 3
exclude by prefix 21
include by prefix 21
refine 12

W

with clauses
add 17
map 24
specify additional in body 60
specify additional in module spec 58