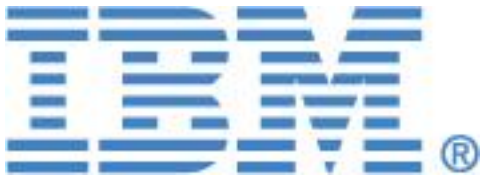


IBM® Rational® Rhapsody® Developer for Ada Code Generator

User's Guide





1. Notices

© Copyright IBM Corporation 1997, 2009.

US Government Users Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send written license inquiries to:

IBM Director of Licensing IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send written inquiries to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome
Minato-ku Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions. Therefore, this statement may not apply to you. ii This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you. Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software
IBM Corporation
1 Rogers Street
Cambridge, Massachusetts 02142 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1997, 2009.

IBM, the IBM logo, ibm.com, Rhapsody, and Statemate are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.html.

Table of Contents

1.	NOTICES	2
INTRODUCTION.....		14
2.	INSTALLATION NOTES	14
2.1.	SUPPORTED COMPILERS.....	14
2.2.	BEHAVIOR SERVICES AND ANIMATION LIBRARIES	14
2.2.1.	Automatically building behavior services and animation library.....	15
2.2.2.	Manually rebuilding the behavior services	15
2.2.3.	Manually rebuilding the animation C libraries	17
2.3.	BOOCH COMPONENTS	20
2.4.	JAVA ENVIRONMENT	20
2.4.1.	JDK-JRE requirements	20
3.	COVERAGE	21
3.1.	RHAPSODY MODEL ELEMENT COVERAGE.....	21
3.2.	ADA CODE COVERAGE	26
4.	ADA CODE GENERATION	26
4.1.	CLASSES	27
4.1.1.	Class definition	27
4.1.2.	Class record type visibility.....	27
4.1.3.	Inheritance	29
4.1.4.	Initialization code	30
4.1.5.	Static class	31
4.2.	ATTRIBUTES	33
4.2.1.	Accessor and mutator.....	33
4.2.2.	Non-static attributes.....	33
4.2.3.	Static attributes	36
4.2.4.	Static attributes visibility.....	36
4.2.5.	Static attributes declaration position	38
4.2.6.	Non-Predefined Attribute Types.....	41
4.2.7.	Guarded Attributes.....	44
4.2.8.	<<Discriminant>> Attributes	44
4.2.9.	Overriding and redefining discriminant attributes	47
4.3.	OPERATIONS	50
4.3.1.	Guarded operations	53
4.3.2.	Template operations and their instantiations.....	53
4.3.3.	Access parameters.....	57
4.3.4.	Class-wide parameters.....	58
4.4.	DEPENDENCIES	60
4.4.1.	<<Usage>> dependencies	60
4.4.2.	<<Renames>> dependencies	62
4.5.	ACTORS	64
4.6.	PACKAGES	67
4.6.1.	Child Packages	69
4.6.2.	Nested Packages	70
4.6.3.	Private Packages	73
4.6.4.	Elaboration Pragmas.....	75
4.6.5.	<<Container>> Packages	76
4.7.	TYPES	78
4.7.1.	Type declaration	78
4.7.2.	Type visibility.....	81
4.7.3.	Type declaration position.....	82
4.7.4.	Type defined as a class.....	82

4.8.	TEMPLATE CLASSES AND THEIR INSTANTIATION.....	87
4.8.1.	<i>template classes</i>	87
4.8.2.	<i>template instantiations</i>	87
4.8.3.	<i>template inheritance</i>	88
4.8.4.	<i>template instantiation inheritance</i>	89
4.9.	CONCURRENT TYPES	91
4.9.1.	<i>Tasks</i>	91
4.9.2.	<i>Protected Objects</i>	97
4.10.	ENTRYPOINTS.....	103
4.11.	SINGLETON CLASSES.....	104
4.11.1.	<i>Ada 95</i>	104
4.11.2.	<i>Ada 83</i>	106
4.12.	UNIDIRECTIONAL RELATIONS	109
4.12.1.	<i>Multiplicity = 1</i>	109
4.12.2.	<i>Multiplicity > 1, general notes</i>	109
4.12.3.	<i>Details on the Booch components</i>	110
4.12.4.	<i>Multiplicity > 1, bounded</i>	112
4.12.5.	<i>Multiplicity > 1, unbounded</i>	112
4.12.6.	<i>Multiplicity > 1, qualified relations</i>	112
4.13.	BIDIRECTIONAL RELATIONS	114
4.13.1.	<i>SubtypingAndRenaming scheme</i>	114
4.13.2.	<i>IntermediateParentClasses scheme</i>	114
4.14.	PORTS	115
4.14.1.	<i>Limitations</i>	115
4.14.2.	<i>Using ports</i>	115
4.14.3.	<i>Example 1 : behavioral port</i>	116
4.14.4.	<i>Example 2 : fast ports</i>	117
4.14.5.	<i>Multicast ports</i>	117
4.15.	ADA LIBRARIES	120
4.15.1.	<i>Creating an Ada Library</i>	120
4.15.2.	<i>Linking an Ada Library</i>	121
4.16.	CONFIGURATION OF MAIN FILE GENERATION	122
4.16.1.	<i>With Clauses</i>	122
4.16.2.	<i>Configuration Prolog</i>	122
4.16.3.	<i>Instance Creation</i>	122
4.16.4.	<i>RiADefaultActive Initialization</i>	122
4.16.5.	<i>Reactive Instance Hookup</i>	123
4.16.6.	<i>Start Behavior</i>	123
4.16.7.	<i>User defined local variables</i>	123
4.16.8.	<i>User Initialization Code</i>	123
4.16.9.	<i>Configuration Epilog</i>	123
4.17.	INSTANCES DEFINED ON A PACKAGE.....	124
4.17.1.	<i>Package Modifications</i>	124
4.18.	USER-DEFINED HEADER AND FOOTERS	127
4.18.1.	<i>Available properties</i>	127
4.18.2.	<i>Keyword substitution</i>	127
4.18.3.	<i>Script Evaluation</i>	128
4.19.	CUSTOM MAKEFILES	129
4.19.1.	<i>Introduction</i>	129
4.19.2.	<i>Features</i>	129
4.19.3.	<i>Standard Macros and property Keywords</i>	131
4.19.4.	<i>New environment creation</i>	137
4.19.5.	<i>Use cases</i>	137
5.	SPARK CODE GENERATION.....	143
5.1.	ENABLING SPARK CODE GENERATION	143
5.1.1.	<i>Adding the SPARK profile to the model</i>	143
5.1.2.	<i>Setting the SPARK environment</i>	144

5.1.3.	<i>Examination level</i>	145
5.2.	DIFFERENCES BETWEEN CODE GENERATED WITH AND WITHOUT THE SPARK PROFILE	146
5.3.	GENERAL USAGE NOTES ON SPARK PROFILE TAGS	146
5.3.1.	<i>Capturing annotations with string tags</i>	146
5.3.2.	<i>Annotations often come in pairs</i>	146
5.3.3.	<i>Multiple modeling approaches</i>	147
5.4.	INHERIT CLAUSES	147
5.4.1.	<i>Using inheritance</i>	147
5.4.2.	<i>Using <<Usage>> dependencies</i>	148
5.4.3.	<i>Using the inherits tag on a class or a package</i>	149
5.5.	OWN VARIABLES.....	150
5.5.1.	<i>Modeling through tags on attributes</i>	150
5.5.2.	<i>Using the OwnSpec and OwnBody tags</i>	153
5.6.	INITIALIZES ANNOTATIONS	155
5.6.1.	<i>Using tags on attributes</i>	155
5.6.2.	<i>Using tags on class and package</i>	155
5.6.3.	<i>Using <<SPARK_Initializes>> dependencies</i>	155
5.7.	PROOF TYPES AND PROOF FUNCTIONS ANNOTATIONS	155
5.8.	GLOBAL ANNOTATIONS	157
5.8.1.	<i>Using <<SPARK_Global>> dependencies</i>	157
5.8.2.	<i>Using tags on operations</i>	160
5.9.	DERIVES ANNOTATION.....	162
5.10.	PRECONDITON, POSTCONDITION AND RETURN ANNOTATIONS	163
5.11.	HIDE ANNOTATION	166
5.11.1.	<i>On a class or a package</i>	166
5.11.2.	<i>On an operation</i>	169
5.12.	MAIN PROGRAM ANNOTATION.....	169
6.	BEHAVIORAL CODE GENERATION	170
6.1.	OVERVIEW OF THE BEHAVIORAL FRAMEWORKS.....	170
6.1.1.	<i>Selecting the behavioral framework implementation</i>	170
6.1.2.	<i>Differences between the Ada 83 and the Ada 95 implementations</i>	170
6.1.3.	<i>Common features of both frameworks</i>	170
6.2.	USING THE ADA 83 BEHAVIORAL FRAMEWORK	171
6.2.1.	<i>Limitations</i>	171
6.2.2.	<i>Event-based reactive classes</i>	172
6.2.3.	<i>Reactive Class Generation</i>	172
6.2.4.	<i>Active Class Generation</i>	177
6.2.5.	<i>Working with Active and Reactive Classes</i>	179
6.2.6.	<i>Active Reactive Class</i>	180
6.2.7.	<i>Default Active Class</i>	181
6.2.8.	<i>Triggered Operations</i>	182
6.3.	USING THE ADA 95 BEHAVIORAL FRAMEWORKS	184
6.3.1.	<i>Limitations</i>	184
6.3.2.	<i>New Ada 95 Framework changes</i>	184
6.3.3.	<i>Reactive classes</i>	185
6.3.4.	<i>Event-based reactive classes</i>	185
6.3.5.	<i>Sending events</i>	185
6.3.6.	<i>Using triggered operations</i>	185
6.3.7.	<i>Accessing the current event parameters</i>	185
6.3.8.	<i>Testing if a state is active</i>	186
6.3.9.	<i>Working with Active and Reactive Classes</i>	186
6.3.10.	<i>Default Active Class</i>	186
6.3.11.	<i>User Active class for ravenstar</i>	186
7.	CODE ORDER RESPECT TOOL.....	187
7.1.	INTRODUCTION	187
7.2.	ACTIVATION AND USAGE.....	187

7.3.	FREQUENT ERRORS.....	187
7.3.1.	Syntax error in Ada file.....	187
7.3.2.	Syntax error due to model.....	188
7.3.3.	Adding a new element.....	188
8.	ANIMATION IN IBM® RATIONAL® RHAPSODY® DEVELOPER FOR ADA.....	188
8.1.	ENABLING ANIMATION	188
8.1.1.	Animation of a user defined type.....	190
8.2.	ANIMATION ON REMOTE HOST	192
9.	GENERATION RULES CUSTOMIZATION	193
9.1.	OVERVIEW	193
9.2.	RULES MODIFICATION	193
9.3.	LEGACY UML 1.3 METAMODEL BASED RULESET	193
10.	COMPILERS AND RELATED TOOLS SUPPORT	194
10.1.	SUPPORTED COMPILERS/IDES, TOOLS & ENVIRONMENTS	194
10.2.	ENVIRONMENT SPECIFIC INSTRUCTIONS	194
10.2.1.	Using the INTEGRITY simulator with Rhapsody.....	194
10.2.2.	INTEGRITY BSP support.....	194
10.2.3.	Raven/PPC BSP Support	194
10.2.4.	GNAT issues.....	195
10.3.	COMPILER USAGE NOTE FOR OBJECTADA AND GREENHILLS COMPILERS	195
10.4.	COMPILER SUPPORT LIMITATIONS.....	195
10.4.1.	Rhapsody Frameworks support.....	195
10.4.2.	Compilation error messages	195
10.4.3.	Notes on Pre-compiled libraries	195
10.5.	COMPILER AND ASSIMILATED TOOLS RELATED PROPERTIES	197
11.	MODEL LIMITATIONS.....	200
11.1.	LIMITATIONS FOR ADA 83.....	200
11.2.	GENERAL LIMITATIONS	200
APPENDIX A: PROPERTIES FOR IBM® RATIONAL® RHAPSODY® DEVELOPER FOR ADA.....		201
APPENDIX B: TAGS FOR IBM® RATIONAL® RHAPSODY® DEVELOPER FOR ADA.....		215
APPENDIX C: STEREOTYPES FOR IBM® RATIONAL® RHAPSODY® DEVELOPER FOR ADA.....		218

Table of Figures

Figure 1: Rebuilding Ada 83 Behavioral Framework.	16
Figure 2: Changing the behavioral framework generation directory for an Integrity-based configuration	17
Figure 3: A simple class in Rhapsody.	27
Figure 4: The generated Ada code for a simple class.	27
Figure 5: Controlling the location of the record type definition.	28
Figure 6: Inheritance in Rhapsody.	29
Figure 7: The package specification for a specialized class.	29
Figure 8: Setting the initialization code property for a class	30
Figure 9: Generated body for a class with initialization code	31
Figure 10: setting IsStatic property for a class	32
Figure 11: Non-static attributes of a class.	33
Figure 12: The package specification for non-static attributes.	34
Figure 13: The package body for non-static attributes.	35
Figure 14: Controlling the visibility of the accessor and mutator.	36
Figure 15: Static attributes of a class.	36
Figure 16: The package specification for static attributes.	37
Figure 17: The package body for static attributes.	38
Figure 18: Static attributes of a class with overridden declarationPosition property	39
Figure 19: Setting the DeclarationPosition for a static attribute to BeforeClassRecord	40
Figure 20: Setting the DeclarationPosition for a static attribute to AfterClassRecord	41
Figure 21: Generated code for static attributes with overridden DeclarationPosition property	41
Figure 22: Non-static attribute definition.	42
Figure 23: Static attribute definition.	43
Figure 24: The package specification for non-predefined type attributes.	43
Figure 25: Guarding an Attribute.	44
Figure 26: Modeling a class with a <<Discriminant>> attribute	44
Figure 27: Defining an unconstrained array type	45
Figure 28: Setting an attribute stereotype to <<Discriminant>>	45
Figure 29: Defining an attribute with a type definition based on the class record type discriminant	46
Figure 30: Generated code for a class with a discriminant	47
Figure 31: Class with overriding and redefining discriminant	47
Figure 32: Overriding discriminant	48
Figure 33: Overriding and redefining discriminant	48
Figure 34: Redefining discriminant	48
Figure 35: Operations defined on a class.	50
Figure 36: Operation Features.	50
Figure 37: The implementation of <i>myOperation</i>	51
Figure 38: The local variables for <i>myOperation</i>	51
Figure 39: Operations in the package specification.	52
Figure 40: Operations in the package body.	52
Figure 41: Making an Operation Guarded.	53
Figure 42: Modeling a template operation and a template operation instantiation.	53
Figure 43: features of a template operation.	54
Figure 44: setting up template parameters for a template operation.	54
Figure 45: generated code for a template operation specification	55
Figure 46: generated code for a template operation implementation	55
Figure 47: features of a template operation instantiation	56
Figure 48: setting up template arguments for a template operation instantiation	56
Figure 49: generated code for a template operation instantiation.	57
Figure 50: Making a parameter passing mode "access"	57
Figure 51: Making an operation this parameter passing mode "access"	58
Figure 52: Operation using access mode parameters	58
Figure 53: <<Usage>> dependencies in IBM® Rational® Rhapsody® in Ada.	60
Figure 54: An implementation dependency.	60
Figure 55: Creating a "Use" statement.	61
Figure 56: The package specification for dependencies.	61

Figure 57: The package body for dependencies.	62
Figure 58: An Actor in Rhapsody.	64
Figure 59: Package specification for an Actor.	65
Figure 60: Package body for an Actor.	66
Figure 61: A package defined in Rhapsody.	67
Figure 62: The package specification for a Rhapsody package.	68
Figure 63: The package body for a Rhapsody package.	69
Figure 64: Packages and classes used as namespaces.	70
Figure 65: The package specification for class_2.	70
Figure 66: The resulting files including the namespaces.	70
Figure 67: Example of a nested package and a nested class.	71
Figure 68: Setting a class to be generated as a nested package.	71
Figure 69: Setting a package to be generated as a nested package.	72
Figure 70: Controlling the location of the specification of a nested package.	72
Figure 71: Exampe of a private package and a private class.	73
Figure 72: Setting a class to be generated as a private package.	73
Figure 73: Setting a package to be generated as a private package.	74
Figure 74: Specification of a private class.	74
Figure 75: Specification of a private package.	74
Figure 76: Example of a class and a package with elaboration pragmas.	75
Figure 77: Enabling generation of elaboration pragmas on a class.	75
Figure 78: Enabling generation of elaboration pragmas on a package.	75
Figure 79: Specifcation of a class with elaboration pragmas.	76
Figure 80: Specifcation of a package with elaboration pragmas.	76
Figure 81: A Sample <<Container>> Package.	77
Figure 82: Types defined in Rhapsody.	78
Figure 83: The declaration of a private type on a class.	78
Figure 84: The declaration of a public type on a class.	79
Figure 85: The declaration of a private type on a package.	80
Figure 86: The declaration of a public type on a package.	81
Figure 87: Controlling the visibility of a type.	81
Figure 88: The package specification for a class with types.	82
Figure 89: The package specification for a package with types.	82
Figure 90: Representation of a typed class.	83
Figure 91: Generated code of a typed class.	83
Figure 92: Representation of a subtype.	83
Figure 93: Generated code of a subtype.	83
Figure 94: Representation of a range type.	84
Figure 95: Generated code of a range type.	84
Figure 96: Representation of a range type with dependency to a constant.	84
Figure 97: Generated code of a range type.	84
Figure 98: Representation of an array type.	85
Figure 99: Generated code of an array type.	85
Figure 100: Representation of a variant record type.	85
Figure 101: Generated code of a variant record type.	86
Figure 102: Definition of a template class.	87
Figure 103: Package specification for a generic package.	87
Figure 104: An instantiation of a template class.	88
Figure 105: The generated Ada package for a generic instantiation.	88
Figure 106 Inheritance between template classes.	88
Figure 107 generated code for a template class derived from another template class.	89
Figure 108 Modeling instantiation of a template inheritance hierarchy.	89
Figure 109 generated code for a base template instantiation class.	89
Figure 110 generated code for a derived template instantiation class.	89
Figure 111 generate code for another derived template instantiation class.	90
Figure 112 Modeling template inheritance hierarchy across (Ada) children packages.	90
Figure 113 Generated code for a derived template class that is a child package of its base class.	90

Figure 114: Generated code for the instantiation of a derived template class that is a child package of its base class.....	90
Figure 115: Ada tasks in Rhapsody.....	91
Figure 116: Setting the record type visibility to “Private” for an <<AdaTaskType>> class.....	91
Figure 117: Setting an operation on a <<AdaTaskType>> class to generate as a regular operation.	92
Figure 118: Ada task specification.....	92
Figure 119: Ada task body.....	93
Figure 120: Ada task type specification.....	94
Figure 121: Ada task type body.....	95
Figure 122: Ada task with default entry.....	96
Figure 123: Specification of Ada Task with default entry.....	96
Figure 124: Implementation of Ada Task with default entry.....	97
Figure 125: Protected objects in Rhapsody.....	97
Figure 126: Setting the record type visibility to “Private” for an <<AdaProtectedType>> class.....	98
Figure 127: Applying the <<entry>> stereotype to a protected object operation.....	98
Figure 128: Setting the guard for a protected object/type entry.....	99
Figure 129: Protected object specification.....	99
Figure 130: Protected object body.....	100
Figure 131: Protected type specification.....	101
Figure 132: Protected type body.....	102
Figure 133: An entrypoint in Rhapsody.....	103
Figure 134: The entrypoint definition.....	103
Figure 135: A singleton class in Rhapsody.....	104
Figure 136: The package specification for a singleton class in Ada 95.....	105
Figure 137: The package body for a singleton class in Ada 95.....	106
Figure 138: Changing the component to generate Ada 83 code.....	107
Figure 139: The package specification for a singleton class in Ada 83.....	107
Figure 140: The package body for a singleton class in Ada 83.....	108
Figure 141: Class relations with multiplicity = 1.....	109
Figure 142: Class relations with multiplicity > 1, bounded.....	112
Figure 143: Setting the Component to Create a Library.....	120
Figure 144: Using an Ada Library.....	121
Figure 145: Configuration Instances.....	122
Figure 146: Auto-generated Entrypoint.....	123
Figure 147: Global Instances on a Package.....	124
Figure 148: Global Instance with Multiplicity = 1.....	124
Figure 149: Global Instance with Multiplicity > 1.....	125
Figure 150: The Instances Package Specification.....	126
Figure 151: The Instances Package Body.....	126
Figure 152: Defining custom header and footer at the component level.....	127
Figure 153: Inserting keywords inside user-defined header and footer.....	128
Figure 154: Example of generated code using user-defined header and footer.....	128
Figure 155: Modeling inherit clauses via inheritance.....	147
Figure 156: Generated code for derived class using the SPARK profile.....	148
Figure 157: Modeling inherit clauses via <<Usage>> dependencies.....	148
Figure 158: Generated code for dependency client class using the SPARK profile.....	149
Figure 159: Modeling inherit clauses via Inherit tag.....	149
Figure 160: Setting the inherit tag on a class.....	150
Figure 161: Generated code for inherit tag on a class using the SPARK profile.....	150
Figure 162: Modeling an own annotation on a package via tags on an attribute.....	151
Figure 163: Setting some of the tags related to own variables on an attribute.....	152
Figure 164: Setting a default value on an initialized own variable attribute.....	153
Figure 165: Generated annotations for an initialized own variable.....	153
Figure 166: Modeling an own annotation on a package via tags on packages.....	154
Figure 167: Disabling the generation of an own annotation at the attribute level.....	154
Figure 168: Setting some of the tags related to own variables on a package.....	155
Figure 169: Modeling a package with a proof type and a proof function.....	156
Figure 170: Setting the stereotype of a function to <<SPARK_Proof>>.....	156

Figure 171: Setting the stereotype of a type to <<SPARK_Proof>>	157
Figure 172: Generated code for package with proof type and proof function	157
Figure 173: Modeling global annotations via dependencies from operation to attribute	158
Figure 174: Setting the stereotype of a dependency to <<SPARK_Global>>	158
Figure 175: Setting the mode of a <<SPARK_Global>> dependency	159
Figure 176: Controlling where the annotation is generated for a <<SPARK_Global>> dependency	159
Figure 177: Specification for a package with a <<SPARK_Global>> dependency from an operation to a package	160
Figure 178: Implementation for a package with a <<SPARK_Global>> dependency from an operation to a package	160
Figure 179: Modeling global annotations via GlobalSpec tag	160
Figure 180: Setting the GlobalSpec tag on an operation	161
Figure 181: Specification for a package with an operation with a GlobalSpec tag	161
Figure 182: Implementation for a package with an operation with a GlobalSpec tag	162
Figure 183: Modeling derives annotations via DerivesSpec tag	162
Figure 184: Setting the DerivesSpec tag on an operation	163
Figure 185: Specification for a package with an operation with a DerivesSpec tag	163
Figure 186: Modeling post conditions annotations via PostConditionSpec tag	164
Figure 187: Setting the PostConditionSpec tag on an operation	164
Figure 188: Specification for a package with an operation with a PostConditionSpec tag	165
Figure 189: Implementation for a package with an operation with a PostConditionSpec tag	166
Figure 190: Modeling hide annotations on packages and operations	167
Figure 191: Generated body code for a package body with its HideBody tag set to true	167
Figure 192: Setting the elaboration code on a package	168
Figure 193: Generated code for a package body with elaboration code and its HideElaborationCode tag set to true	168
Figure 194: Generated code for a package specification with its HidePrivatePart tag set to true	168
Figure 195: Generated code for an operation body with its HideBody tag set to true	169
Figure 196: Operations to Control the Reactive Class Statechart.	171
Figure 197: Definition of an Active Class	171
Figure 198: Operations to Control the Active Class	171
Figure 199: A Reactive Class and its Active Class.	172
Figure 200: The "With" Clauses for a Reactive Class.	172
Figure 201: The Reactive Class Record.	173
Figure 202: Operations for the Current Event Information.	174
Figure 203: Initialization and Finalization of the Reactive Class.	174
Figure 204: Example Reactive Class Project.	174
Figure 205: Statechart for the Reactive Class.	174
Figure 206: The Event Types for the Reactive Class.	175
Figure 207: Using relative naming for current event data	176
Figure 208: Event data record type using relative naming	176
Figure 209: The Parent Package of the Reactive Class.	176
Figure 210: Operations to Generate Events for a Reactive Class	176
Figure 211: Accessing a trigger parameter value (using full namespace based naming)	177
Figure 212: Accessing a trigger parameter value (using relative naming)	177
Figure 213: Setting the Record Type Visibility for an Active Class.	178
Figure 214: The Task Generated for an Active Class.	178
Figure 215: The Record Definition for the Active Class.	178
Figure 216: "With" Statements for an Active Class.	179
Figure 217: The Public Operations of an Active Class.	179
Figure 218: Initialization and Finalization of the Active Instance.	179
Figure 219: Using an Active and Reactive Class.	180
Figure 220: An Active-Reactive Class.	180
Figure 221: Using an Active-Reactive Class	181
Figure 222: Using the Default Active Class	182
Figure 223: A Sample Model of a Synchronous Reactive Class	182
Figure 224: A Statechart Using Triggered Operations	182
Figure 225: Triggered Operation Unique Identifiers.	183

Figure 226: Generated Procedures for Triggered Operations.	183
Figure 227: Using a Synchronous Reactive Class.	184
Figure 228: The State_Type enumeration type for a reactive class.	185
Figure 229: A Reactive Class and its Active Class.	185
Figure 230: Operations to Generate Events for a Reactive Class.	185
Figure 231: Accessing a trigger parameter value.	186
Figure 232: Enabling Animation in the Configuration.	189

Introduction

This document is the user's guide for the code Generator of *IBM® Rational® Rhapsody® Developer for Ada*.

2. Installation Notes

2.1. Supported compilers

The code generator has been tested against the following Ada compilers

- AdaCore's GNAT 3.15p
- AdaCore's GNAT Pro 6.01
- AdaCore's GNAT Pro 6.02
- AONIX ObjectAda ® Enterprise Edition 8.4
- Green Hills Software's AdaMULTI® 3.5 for PowerPC on INTEGRITY® v4.0.8, with updates from Green Hills Support services.
- Green Hills Software's AdaMULTI® 3.5 for x86 on Win32, with updates from Green Hills Support services.

For AdaMULTI 3.5, updates may be necessary in order to work with IBM® Rational® Rhapsody® Developer for Ada. Please contact Green Hills Support staff at support-ada@ghs.com (or support-adauk@ghs.com if you are in Europe) for details, stating that you intend to use IBM® Rational® Rhapsody® Developer for Ada.

In order to determine if you need such updates, try to rebuild the behavior services. If you get an assertion failure message from the compiler, you need the updates.

- Green Hills Software's AdaMULTI® 4.0.7 for PowerPC on INTEGRITY® v5.0.4.
- Green Hills Software's AdaMULTI® 4.0.7 for x86 on Win32

If user needs to use another compiler, he can customize IBM Rational Rhapsody, in order to generate appropriate files to compile the project. See §4.19 CUSTOM MAKEFILES.

Warning :

If user uses AdaCore's GNAT 3.15p compiler with the default installation path, the compilation will fail, because the path contains a white space which is not recognized by the compiler. In this case, IBM Rational Rhapsody must be installed into another path which doesn't contain white spaces.

2.2. Behavior services and animation Libraries

As part of IBM Rational Rhapsody installation you are receiving the following files:

- Basic behavior services sources.
- Animation C libraries (currently supported only for Win32 and Integrity Targets).

These files are not compiled during install, and must be compiled by user. They can be compiled manually or automatically from Rhapsody menu. This is explained in the following chapter.

These files have been tested against the following Ada compilers

- AdaCore's GNAT 3.15p
- AdaCore's GNAT Pro 6.01
- AdaCore's GNAT Pro 6.02
- AONIX ObjectAda ® Enterprise Edition
- Green Hills Software's AdaMULTI® 3.5 for PowerPC on INTEGRITY® v4.0.8, with updates from Green Hills Support services.
- Green Hills Software's AdaMULTI® 3.5 for x86 on Win32, with updates from Green Hills Support services.

For AdaMULTI 3.5, updates may be necessary in order to work with IBM® Rational® Rhapsody® Developer for Ada. Please contact Green Hills Support staff at support-ada@ghs.com (or support-adauk@ghs.com if you are in Europe) for details, stating that you intend to use IBM® Rational® Rhapsody® Developer for Ada.

In order to determine if you need such updates, try to rebuild the behavior services. If you get an assertion failure message from the compiler, you need the updates.

- Green Hills Software's AdaMULTI® 4.0.7 for PowerPC on INTEGRITY® v5.0.4.
- Green Hills Software's AdaMULTI® 4.0.7 for x86 on Win32

In some cases, where the compiler version used by you is different from the version used on installation, or when you need versions of the libraries targeted for a specific operating system / hardware architecture combination (e.g. a specific board running with INTEGRITY), you will need to rebuild them. For GNAT, ObjectAda and AdaMulti compilers on Win32 platforms, rebuilding the libraries can be done by executing the recompile_<ENV>.bat program in the Sodius subdirectory. The following instructions indicate how to rebuild them manually for every supported platform.

2.2.1. Automatically building behavior services and animation library

Behavior services and animation library can be built in a simple way by using the menu Code/Build Framework. This is the recommended method to build them for the first time after Rhapsody install.

This command will automatically build the behavior services and animation library for the environment defined in the current configuration. This command calls a script defined in the property : Ada_CG::<ENV>::buildFrameworkCommand.

Compilation results are logged into the file <Rhp_instal_dir>\Sodius\RI_A_CG\recompile.log

For the environment Integrity5 of the FWKs, it is recommended to regenerate the files, in order to setup correctly the make file with your environment.

2.2.2. Manually rebuilding the behavior services

If you are using the Ada 83 framework then

- Open the RiAServices model under <Rhapsody installation>\Share\LangAda83\model directory.

If you are using the old Ada 95 framework then

- Open the RiA_Framework model under <Rhapsody installation>\Share\LangAda\model directory.

If you are using the New Ada 95 framework then

- Open the RiA_Framework model under <Rhapsody installation>\Share\LangAda95\Ada_FWK directory.

- Select the appropriate configuration among the available ones :
 - GNAT_Win32
 - OBJECTADA_Win32
 - AdaMULTI_Win32
 - AdaMULTI_4_Win32
 - AdaMULTI_INTEGRITY_<target>
 - AdaMULTI_4_INTEGRITY_<target>
- Select *Rebuild* from the code menu.

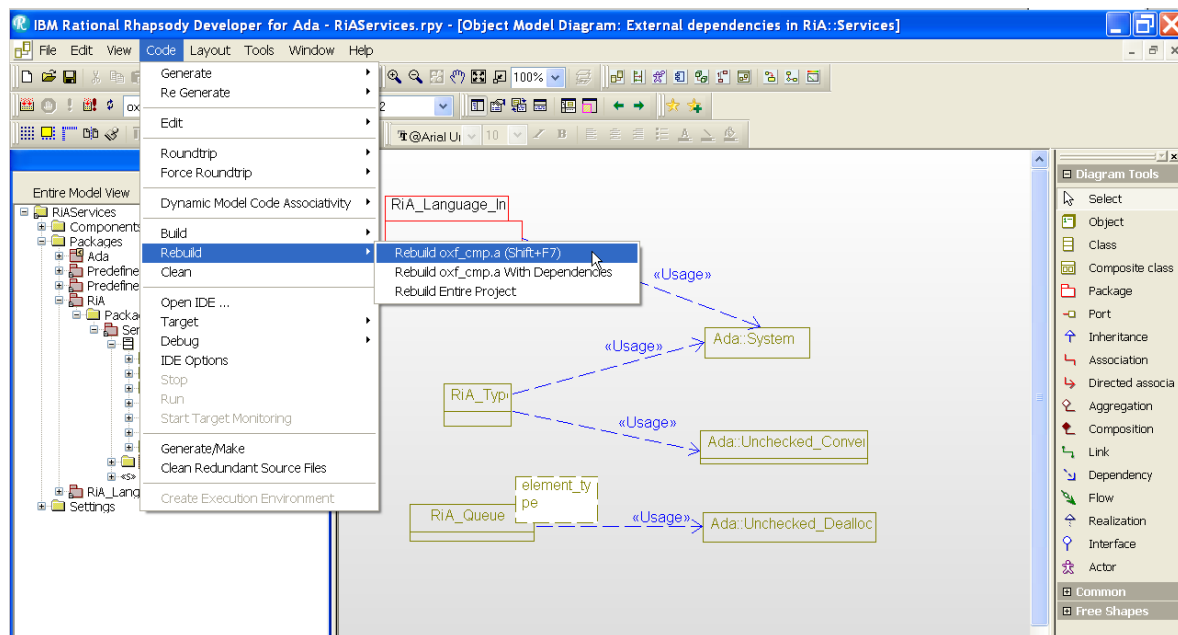


Figure 1: Rebuilding Ada 83 Behavioral Framework.

Adding a platform specific target for Integrity :

- Make a copy of one of the existing Integrity configuration (for example ADAMULTI_4_INTEGRITY_sim800)
- Rename it appropriately by giving it the appropriate target suffix (for example mbx800)
- Open the settings tab of the configuration, and edit the directory field, replacing the old target suffix by the new one (sim800 => mbx800)

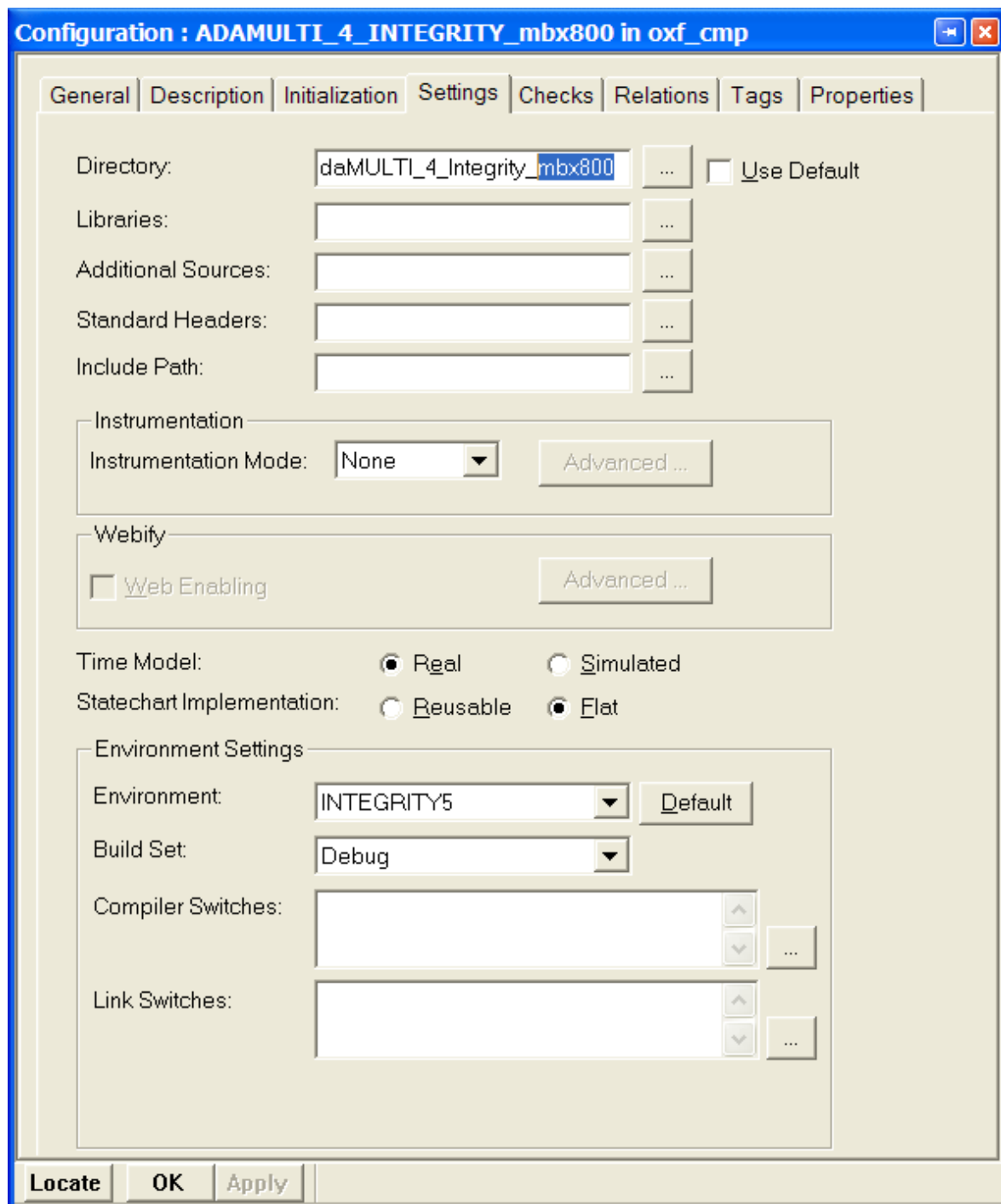


Figure 2: Changing the behavioral framework generation directory for an Integrity-based configuration

- Generate code and build component

2.2.3. Manually rebuilding the animation C libraries

Rebuilding for GNAT

The build is done using the GCC compiler supplied as part of the GNAT package. The process differs a bit depending on the version of GNAT you've installed.

If you are using GNAT v3.13 or earlier version follow these steps :

- Open a command prompt.

- Go to the <Rhapsody installation>\Share\LangC directory.
- Type “***make -f AdaWinbuild.mak PATH_SEP=***”

If you are using a later version of GNAT, you might have to follow these steps :

- Make sure the GNAT Win32 support package is installed
- Open a command prompt.
- Go to the <Rhapsody installation>\Share\LangC directory.
- Type “***make -f AdaWinbuild.mak buildLibs***”

The following files will be generated in the <RhapsodyInAdaInstallDir>\Share\LangC\lib directory:

- AdaWinaomanim.lib
- AdaWinomcomappl.lib
- AdaWinoxfirst.lib

Notes:

- The ***GNAT_HOME*** environment variable must be set to the location of the GNAT install directory. The path must use forward slashes and not backslashes in its path.
- If your environment contains UNIX like Shell utilities, you will need to remove them from the path in order to compile.
- The Win32 libraries may be included as part of a separate install for GNAT. For example, for 3.15p, the install file is called gnatwin-3.15p.exe.

If you are using a later version of GNAT and wish to use animation, make sure to have GNU Make installed in the same directory as GNAT executables, rename it as “make.exe”, and follow the previous instructions.

Rebuilding for ObjectAda

The build is done using the GCC compiler supplied as part of the cygwin package. :

- Open a command prompt.
- Go to the <Rhapsody installation>\Share\LangC directory.
- Type “***..\\etc\\cygwinMake.bat AdaWinBuild.mak adaBuildLibs***”

The following files will be generated in the <RhapsodyInAdaInstallDir>\Share\LangC\lib directory:

- AdaWinaomanim.lib
- AdaWinomcomappl.lib
- AdaWinoxfirst.lib

Rebuilding for MultiWin32

- Open a command prompt.

If you are using Multi 3.5 or an older version follow these steps :

- Go to the <Rhapsody installation>\Share\LangC directory.

- Type “**MultiWin32Build.bat <AdaMultiWin32InstallDir> ada bld**”, replacing <AdaMultiWin32InstallDir> by the appropriate directory for your machine

The following files will be generated in the <RhapsodyInAdaInstallDir>\Share\LangC\lib directory:

- AdaMultiWin32Aomanim.dba
- AdaMultiWin32Aomanim.lib
- AdaMultiWin32OmComAppl.dba
- AdaMultiWin32OmComAppl.lib
- AdaMultiWin32OxfInst.dba
- AdaMultiWin32OxfInst.lib

If you are using Multi 4.0 or a newer version follow these steps :

- Go to the <Rhapsody installation>\Share\LangC directory.
- Type “**MultiWin32Build.bat <AdaMultiWin32InstallDir> ada**”, replacing <AdaMultiWin32InstallDir> by the appropriate directory for your machine
- AdaMulti4Win32Aomanim.dba
- AdaMulti4Win32Aomanim.lib
- AdaMulti4Win32OmComAppl.dba
- AdaMulti4Win32OmComAppl.lib
- AdaMulti4Win32OxfInst.dba
- AdaMulti4Win32OxfInst.lib

Rebuilding for Integrity

- Open a command prompt.

If you are using Multi 3.5 or an older version follow these steps :

- Go to the <Rhapsody installation>\Share\LangC directory.
- Type “**IntegrityBuild.bat <AdaMultiIntegrityInstallDir> <TargetCPU> <AdaMultiIntegrityInstallDir> ADA bld**”, replacing <AdaMultiIntegrityInstallDir> by the appropriate directory for your machine and <TargetCPU> by the desired target.

The following files will be generated in the <RhapsodyInAdaInstallDir>\Share\LangC\lib directory:

- AdaIntegrityAomAnim<TargetCPU>.a
- AdaIntegrityAomAnim<TargetCPU>.dba
- AdaIntegrityOmComAppl.dba
- AdaIntegrityOmComAppl<TargetCPU>.a
- AdaIntegrityOxfInst<TargetCPU>.a
- AdaIntegrityOxfInst<TargetCPU>.dba

If you are using Multi 4.0 or a newer version follow these steps :

- Go to the <Rhapsody installation>\Share\LangC directory.

- Type “***IntegrityBuild.bat*** <IntegrityOSInstallDir> <TargetCPU> <AdaMultiIntegrityInstallDir> ADA”, replacing <AdaMultiIntegrityInstallDir> and <IntegrityOSInstallDir> by the appropriate directories for your machine and <TargetCPU> by the desired target.

The following files will be generated in the <RhapsodyInAdaInstallDir>\Share\LangC\lib directory:

- AdaIntegrity5AomAnim<TargetCPU>.a
- AdaIntegrity5AomAnim<TargetCPU>.dba
- AdaIntegrity5OmComAppl.dba
- AdaIntegrity5OmComAppl<TargetCPU>.a
- AdaIntegrity5OxfInst<TargetCPU>.a
- AdaIntegrity5OxfInst<TargetCPU>.dba

2.3. Booch components

The Booch components are copyrighted© by Grady Booch.

The Booch components and their license terms are available at the AdaPower website at the following URL http://www.adapower.com/original_booch/original_booch.html and <http://www.adapower.net/booch/documentation.html>.

The Booch components are not distributed with IBM® Rational® Rhapsody® Developer for Ada. User must install them manually by following the procedure.

Install Booch Components 95

- Get the files from the following URL for example:
- <http://sourceforge.net/projects/booch95/files/>
- Unzip the files
- Copy the folder “src” into <Rhapsody_install_folder>\Share\LangAda95\Booch_ada_95\src

Install Booch Components 83

- Get the files from the following URL for example:
- http://www.adapower.com/original_booch/original_booch.html
- Unzip the files
- Copy the folder “src” into <Rhapsody_install_folder>\Share\LangAda83\Booch_ada_83\src
- The extension of the files must be changed in order to respect the convention of your compiler:
 - *.1.ada must be changed to *.ads,
 - *.2.ada must be changed to *.adb.

2.4. Java environment

2.4.1. JDK-JRE requirements

The code generator is written using Java technology, and uses the Java API of Rhapsody. Consult the Rhapsody user documentation for details on this API.

3. Coverage

Two views of coverage are provided. The first one answers the question of “Which Rhapsody model elements will be considered when generating Ada code?” while the other one is from the perspective of “Which Ada constructs can be generated from a Rhapsody model?”.

3.1. Rhapsody Model Element Coverage

The following table indicates the model elements that are covered by the code generation in this release.

Model Element	Fields	Covered?
Project		Yes
Active component		No
Component		No
Configuration		No
Folder		No
File		No
Package		
Name		Yes
Description		Yes
CG.Package.FileName		No
CG.Package.UseAsExternal		Yes
Class		
Name		Yes
Description		Yes
CG.Class.FileName		No
CG.Class.UseAsExternal		Yes
Template Class		Yes
Template arguments		Yes
Template Instantiation Class		Yes
Template instantiation arguments		Yes
Reactive Class		Yes
Active Class		Yes
Guarded Class		No
Class Stereotypes		
Abstract		Yes

Model Element	Fields	Covered?
AdaProtectedType		Yes
AdaProtectedObject		Yes
AdaTask		Yes
AdaTaskType		Yes
Entrypoint		Yes
EventFlag		No
MessageQueue		No
Mutex		No
Resource		No
Semaphore		No
Singleton		Yes
Task		No
Timer		No
Nested Class		Yes
Actor		Yes
Type		Yes
Name		Yes
Declaration		Yes
Description		Yes
CG.Type.UseAsExternal		Yes
Event		Yes
Event Reception		No
Timeout		No
Function		Yes
Name		Yes
Description		Yes
Arguments		Yes
Return type		Yes
Implementation		Yes
CG.Operation.Generate		Yes
Template Function		Yes

Model Element	Fields	Covered?
	Template arguments	Yes
Operation		
	Name	Yes
	Description	Yes
	Arguments	Yes
	Return type	Yes
	Implementation	Yes
	Visibility	Yes
	Virtual	Yes
	Static	Yes
	Constant	Yes
	ADA_CG.Operation.Kind	Yes
	ADA_CG.Operation.Inline	Yes
	CG.Operation.Concurrency	Yes
	CG.Operation.Generate	Yes
Triggered Operation		Yes
Constructor		
	Description	Yes
	Arguments	Yes
	Initializer	No
	Implementation	Yes
	Visibility	Yes
	CG.Operation.Generate	Yes
Destructor		
	Description	Yes
	Implementation	Yes
	Visibility	Yes
	Virtual	Yes
	CG.Operation.Concurrency	No
	CG.Operation.Generate	Yes
Attribute		

Model Element	Fields	Covered?
	Name	Yes
	Type	Yes
	Description	Yes
	Visibility	Yes
	Static	Yes
	Default value	Yes
	CG.Attribute.Generate	Yes
	CG.Attribute.AccessorGenerate	Yes
	CG.Attribute.MutatorGenerate	Yes
	CG.Attribute.IsConst	Yes
	CG.Attribute.IsGuarded	Yes
	Ada_CG.Attribute. Visibility	Yes
Variable		
	Name	Yes
	Description	Yes
	Type	Yes
	Default value	Yes
	CG.Attribute.Generate	Yes
	CG.Attribute.Visibility	Yes
Relation		
	Type	No
	Description	No
	Multiplicity	Partial
	Qualifier	Yes
	CG.Realtion.Generate	Yes
	CG.Realtion.Implementation	Partial
	CG.Relation.IsConst	No
	CG.Relation.IsGuarded	Yes
	CG.Relation.Ordered	No
	CG.Relation.GenerateRelationWithActors	Yes
Symmetric Relation		No

Model Element	Fields	Covered?
Part Relation		No
Link instances		No
Generalization		
	Super class	Yes
	CG.Generalization.Generate	Yes
Dependency		
	Dependent	Yes
	CG.Dependency.ConfigurationDependencies	No
	CG.Dependency.UsageType	Yes
	ADA_CG.Dependency.CreateUseStatement	Yes
	CG.Dependency.GenerateRelationWithActors	Yes
Argument		
	Name	Yes
	Type	Yes
	Default value	Yes
	Direction	Yes
	Description	Yes
Statechart		Yes
Activity diagram		Yes

3.2. Ada Code Coverage

The following table indicates the Ada constructs that are covered by the code generation.

Ada Construct	Category	Examples
Overall Structure	Library Subprogram	
	Package	
	With clause	with A_Package;
	Use clause	use A_Package;
	(95) Use Type clause	use type A_Package.A_Type;
	Separate	procedure Proc is separate;
	Renaming as specification	procedure Proc renames A_Package.A_Proc;
	Renaming as body	procedure Proc renames A_Package.A_Proc;
	(95) Child Package	package Parent.Child is ...
	(95) Private Child Package	private package Parent.Child is ...
Exceptions	Predefined exception	program_Error, ...
Generics	Generic formal type	type T is (<>);
	Generic formal subprogram	with procedure Update is Default_Update;
	(95) generic formal package	with package A is new G_A(<>);
	Generic package	Generic type T is (<>); package P is ...
Tasking	Task specification	task T is entry E(...); end;
	Task Type	task type T is entry E(...); end T;
	(95) Protected Type	protected type PT is ...
SPARK	SPARK Annotations	--# global in out A;

4. Ada Code Generation

The detailed rules used to generate Ada code from a Rhapsody model follow this section. But before giving the low-level rules, this section gives an overview of the generation concepts by showing simple examples of the Ada code that is generated from a particular model.

4.1. Classes

4.1.1. Class definition

A class in Rhapsody is represented as an Ada package, and produces a package specification and an optional package body in Ada. The name of the Ada package is the name of the class.

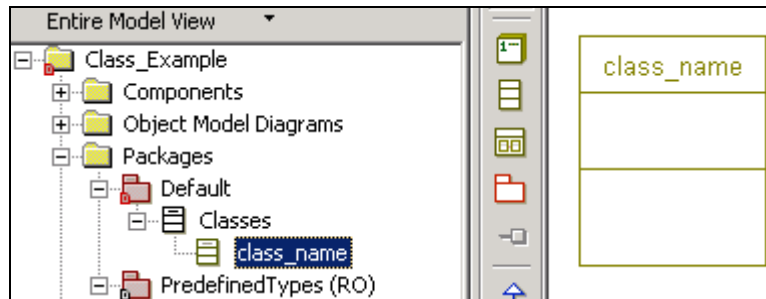


Figure 3: A simple class in Rhapsody.

```
--++ class class_name
package class_name is

    type class_name_t;
    type class_name_acc_t is access all class_name_t;

    type class_name_t is tagged null record;

private

end class_name;
```

Figure 4: The generated Ada code for a simple class.

The name of the file generated is *class_name.ads*.

In addition, two new types are declared in the public part of the package specification – a record type, and an optional access to that type.

4.1.2. Class record type visibility

The definition of this record type appears in the public part of the specification package where the record type is declared, but it could also appear in the private part by setting the “Ada_CG.Class.Visibility” property.

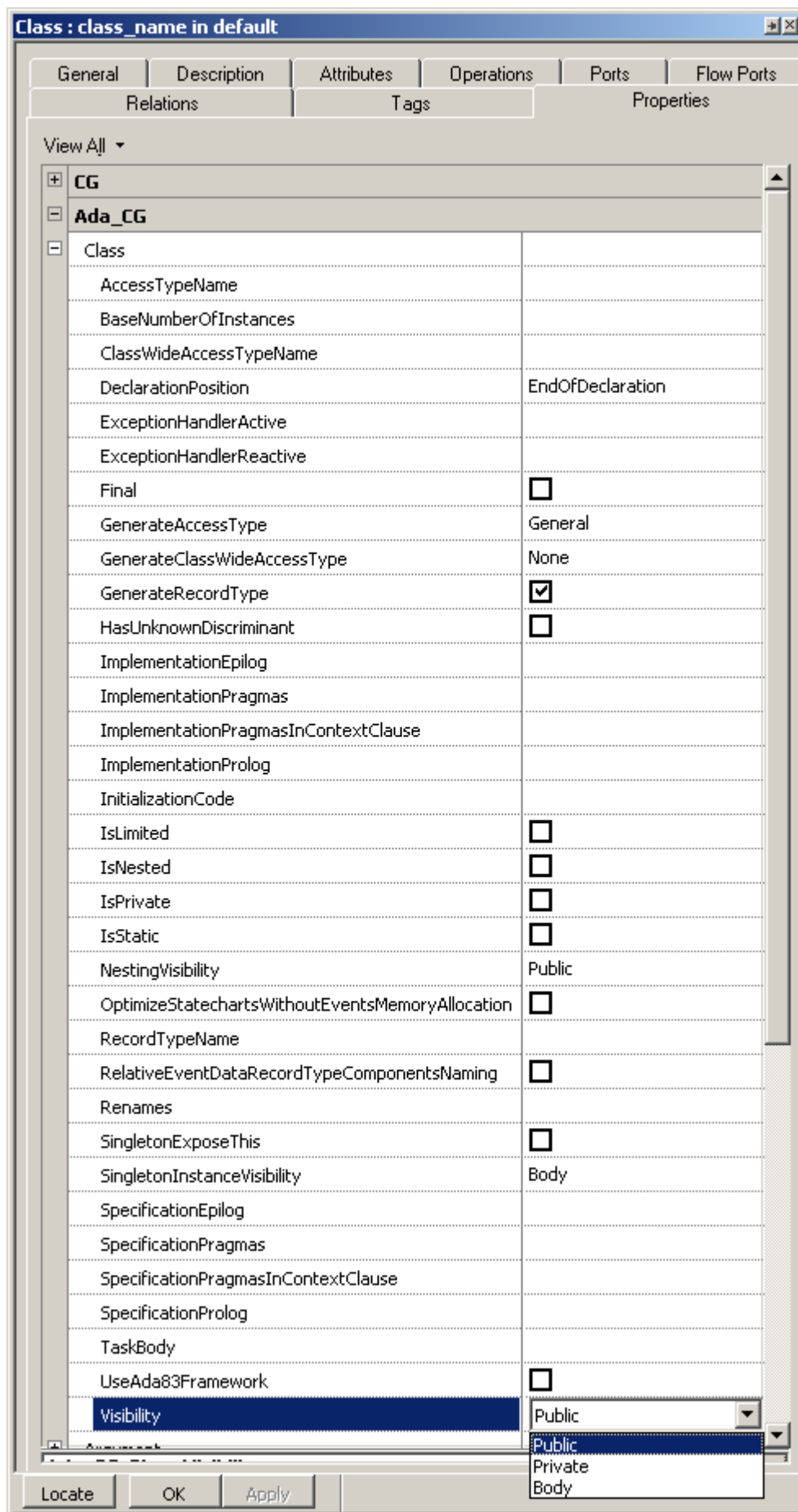


Figure 5: Controlling the location of the record type definition.

4.1.3. Inheritance

When a class inherits from another class, the record type for the subclass is an extension of the record type of the parent class.

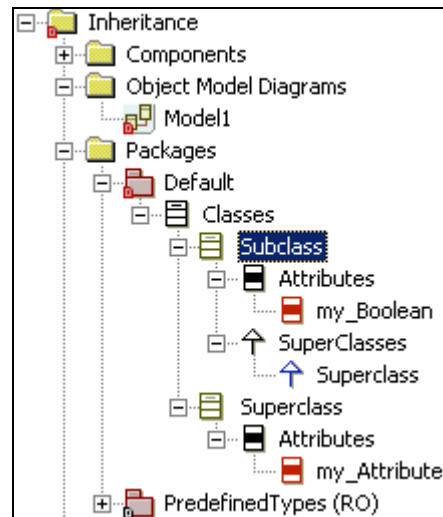


Figure 6: Inheritance in Rhapsody.

```
With Superclass;

--++ class Subclass
package Subclass is

    type Subclass_t is new Superclass.Superclass_t with private;

    type Subclass_acc_t is access all Subclass_t;

    --Public Fields/Variables accessors -----
    function get_my_Boolean (this : in Subclass_t) return Boolean;
    pragma inline (get_my_Boolean);

    procedure set_my_Boolean (this : in out Subclass_t; value : in Boolean);
    pragma inline (set_my_Boolean);

private

    type Subclass_t is new Superclass.Superclass_t with

    record

        -- Fields --
        my_Boolean : Boolean;    --++ attribute my_Boolean

    end record;

end Subclass;
```

Figure 7: The package specification for a specialized class.

Notice that a “With” statement has been generated for the superclass in the package specification of the subclass.

4.1.4. Initialization code

A non-abstract class can have initialization code that is executed during elaboration of the associated package. In order to generate such code, you'll need to edit the InitializationCode property for that class.

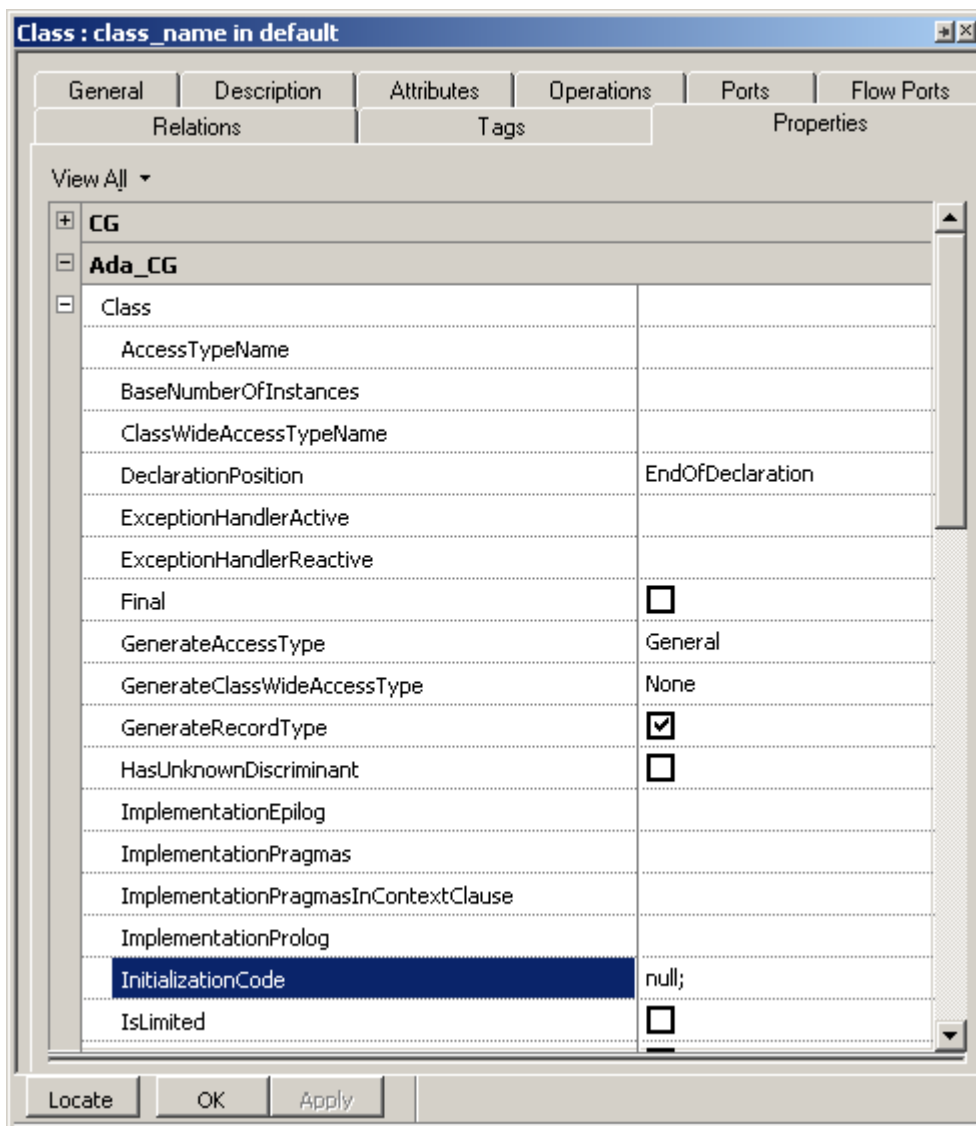


Figure 8 Setting the initialization code property for a class

```
--++ class class_name
package body class_name is

    --Functions/Procedures section -----
    procedure myOperation (this : in out class_name_t) is
    begin
        null;
        --+[ operation myOperation()

        --+]
    end myOperation;

begin
    null;
end class_name;
```

Figure 9 Generated body for a class with initialization code

4.1.5. Static class

A static class is a class with only static attributes and operations. No record type and no "this" parameters are generated for this kind of class.

A Static class is used for safety critical application.

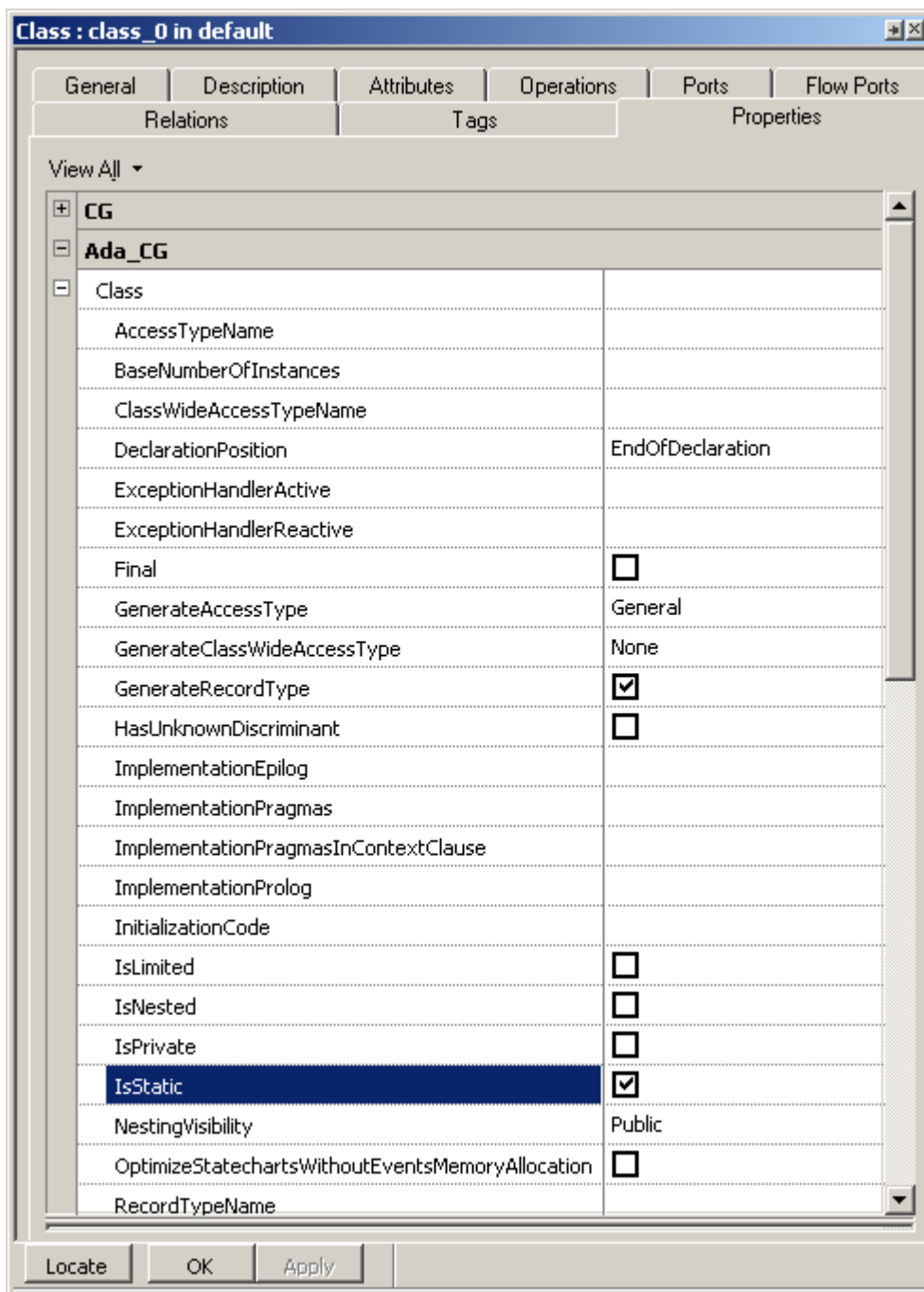


Figure 10 setting IsStatic property for a class

4.2. Attributes

4.2.1. Accessor and mutator

By default, a mutator and accessor are created for each attribute.

4.2.2. Non-static attributes

When non-static attributes are added to a class, these attributes are added to the record type.

When non-static attributes are added to a package, these attributes are handled as static attributes.

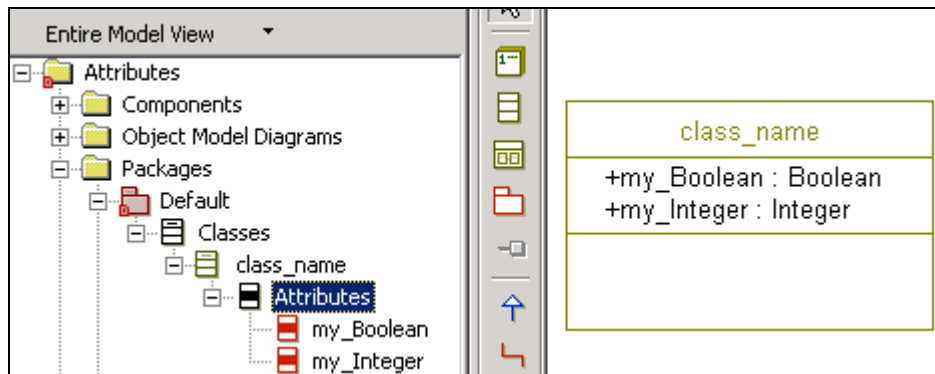


Figure 11: Non-static attributes of a class.

```

--++ class class_name
package class_name is

    type class_name_t;
    type class_name_acc_t is access all class_name_t;

    type class_name_t is tagged

    record

        -- Fields --
        my_Boolean : Boolean;    --++ attribute my_Boolean
        my_Integer : Integer;    --++ attribute my_Integer

    end record;

    --Public Fields/Variables accessors -----
    function get_my_Boolean (this : in class_name_t) return Boolean;
        pragma inline (get_my_Boolean);

    procedure set_my_Boolean (this : in out class_name_t; value : in Boolean);
        pragma inline (set_my_Boolean);

    function get_my_Integer (this : in class_name_t) return Integer;
        pragma inline (get_my_Integer);

    procedure set_my_Integer (this : in out class_name_t; value : in Integer);
        pragma inline (set_my_Integer);

private

end class_name;

```

Figure 12: The package specification for non-static attributes.

Because accessor and mutator methods are created, a package body file is created with the name *class_name.adb*.

```

--++ class class_name
package body class_name is

    --Fields/Variables accessors -----
    function get_my_Boolean(this : in class_name_t) return Boolean is
    begin
        return this.my_Boolean;
    end get_my_Boolean;

    procedure set_my_Boolean (this : in out class_name_t; value : in Boolean) is
    begin
        this.my_Boolean := value;
    end set_my_Boolean;

    function get_my_Integer(this : in class_name_t) return Integer is
    begin
        return this.my_Integer;
    end get_my_Integer;

    procedure set_my_Integer (this : in out class_name_t; value : in Integer) is
    begin
        this.my_Integer := value;
    end set_my_Integer;

end class_name;

```

Figure 13: The package body for non-static attributes.

The record type now contains the two non-static attributes that were added in Rhapsody.

In addition, the attribute accessor and mutator operations contain a *this* parameter that is used to pass in an instance of the type being affected. This is true for all non-static operations that are not for singleton classes.

The accessor and mutator are generated in the public part of the package specification by default, but they can be moved to private part by clicking the “Private” radio button on the features page.

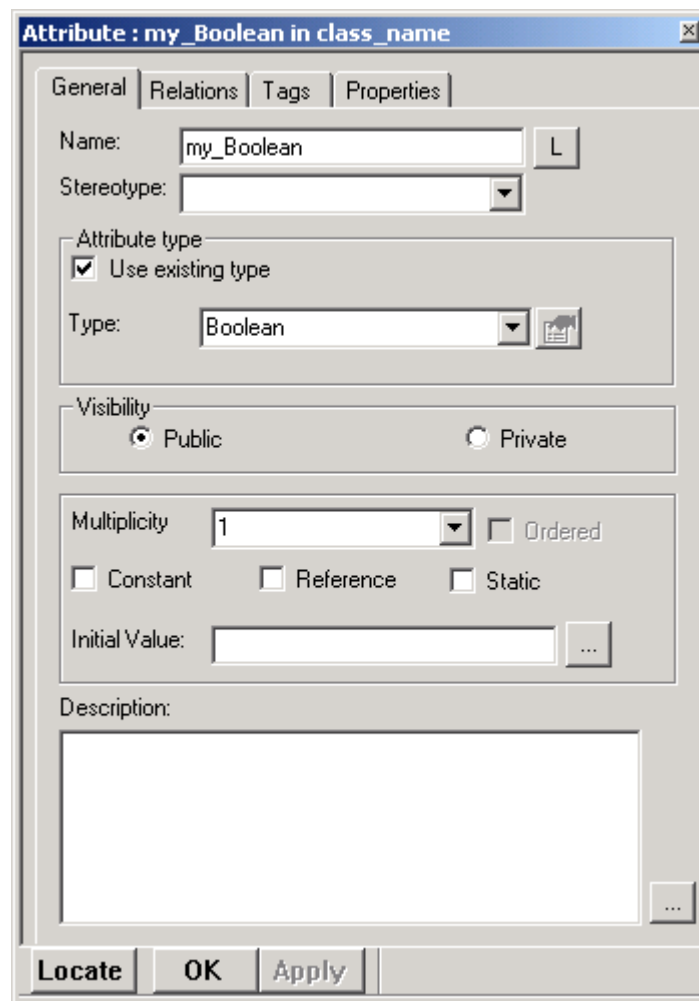


Figure 14: Controlling the visibility of the accessor and mutator.

4.2.3. Static attributes

Attributes marked as static do not create record elements in classes, but instead are represented as variables in the Ada package.

4.2.4. Static attributes visibility

These variables can be defined in the public or private part of the package specification, or in the package body, depending on the setting of property “Ada_CG.Attribute.Visibility”.

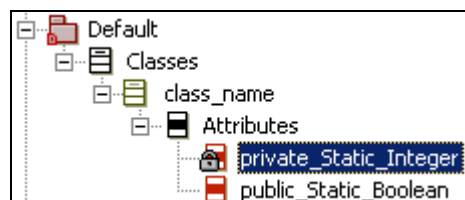


Figure 15: Static attributes of a class.

In this example, both attributes are marked as static. However, the attribute *privateStaticInt* is marked as private, which means that the accessors will appear in the private part of the package specification, and its property “Ada_CG.Attribute.Visibility” is set to “Private”, forcing the variable definition to appear in the private part as well.

```
--++ class class_name
package class_name is

    type class_name_t;
    type class_name_acc_t is access all class_name_t;

    type class_name_t is tagged null record;

    -- Public Variables/Constants -----
    public_Static_Boolean : Boolean;  --++ attribute public_Static_Boolean

    --Public Fields/Variables accessors -----
    function get_public_Static_Boolean return Boolean;
    pragma inline (get_public_Static_Boolean);

    procedure set_public_Static_Boolean (value : in Boolean);
    pragma inline (set_public_Static_Boolean);

private

    -- Private Variables/Constants -----
    private_Static_Integer : Integer;  --++ attribute private_Static_Integer

    --Private Fields/Variables accessors -----
    function get_private_Static_Integer return Integer;
    pragma inline (get_private_Static_Integer);

    procedure set_private_Static_Integer (value : in Integer);
    pragma inline (set_private_Static_Integer);

end class_name;
```

Figure 16: The package specification for static attributes.

```

--++ class class_name
package body class_name is

    --Fields/Variables accessors -----
    function get_private_Static_Integer return Integer is
    begin
        return private_Static_Integer;
    end get_private_Static_Integer;

    procedure set_private_Static_Integer (value : in Integer) is
    begin
        private_Static_Integer := value;
    end set_private_Static_Integer;

    function get_public_Static_Boolean return Boolean is
    begin
        return public_Static_Boolean;
    end get_public_Static_Boolean;

    procedure set_public_Static_Boolean (value : in Boolean) is
    begin
        public_Static_Boolean := value;
    end set_public_Static_Boolean;

end class_name;

```

Figure 17: The package body for static attributes.

4.2.5. Static attributes declaration position

In Ada, declaration order is important. For example, a type declaration might depend on a constant that has to be declared before being used.

In order to provide some degree of control over the declaration order of attributes, the `Ada_CG.Attribute.DeclarationPosition` property can be used. The table below summarizes the different values that this property can take and its effects on the attribute it is being applied.

Value	Description
Default	<p>This is the default setting provided for compatibility reasons. It is similar to the <code>AfterClassRecord</code> setting with the following exception :</p> <ul style="list-style-type: none"> For static attributes defined in a class with an <code>"Ada_CG.Attribute.Visibility"</code> property set to <code>"Public"</code>, these attributes get generated after types with an <code>"Ada_CG.Type.Visibility"</code> property set to <code>"Public"</code>. <p>On new models, it is advised not to use this value. Should you change this value on previous models, make sure the code compiles once you've regenerated it.</p>
BeforeClassRecord	The attribute will be generated immediately before the class record
AfterClassRecord	The attribute will be generated immediately after the class record
StartOfDeclaration	The attribute will be generated immediately after the start of the section (public part of the specification, private part of the specification, package body)

EndOfDeclaration	The attribute will be generated immediately before the end of the section (public part of the specification, private part of the specification, package body)
------------------	---

Table 1 Ada_CG.Attribute.DeclarationPosition property values description

A few special cases :

- if the attributes have their Ada_CG.Attribute.Visibility property set to “Body”
- If the attributes are defined on a package
- If the Ada_CG.Class.Visibility property of the class they are defined in has a different setting

These attributes then have no actual class record around which they can be positioned. In such cases, they are generated around a “virtual” class record location that gives a declaration order as close as possible to the one that would exist if there was a class record definition in the section the attributes are being generated into.

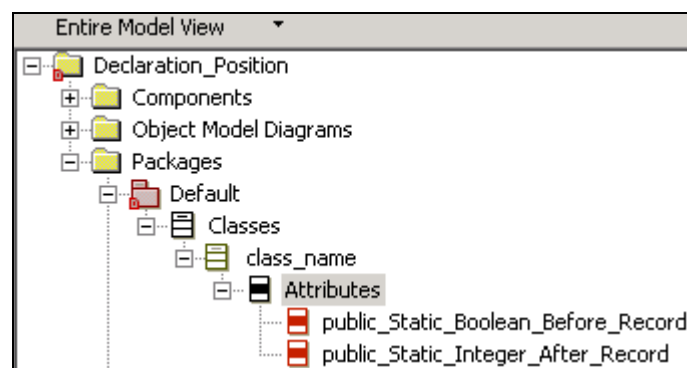


Figure 18 Static attributes of a class with overridden declarationPosition property

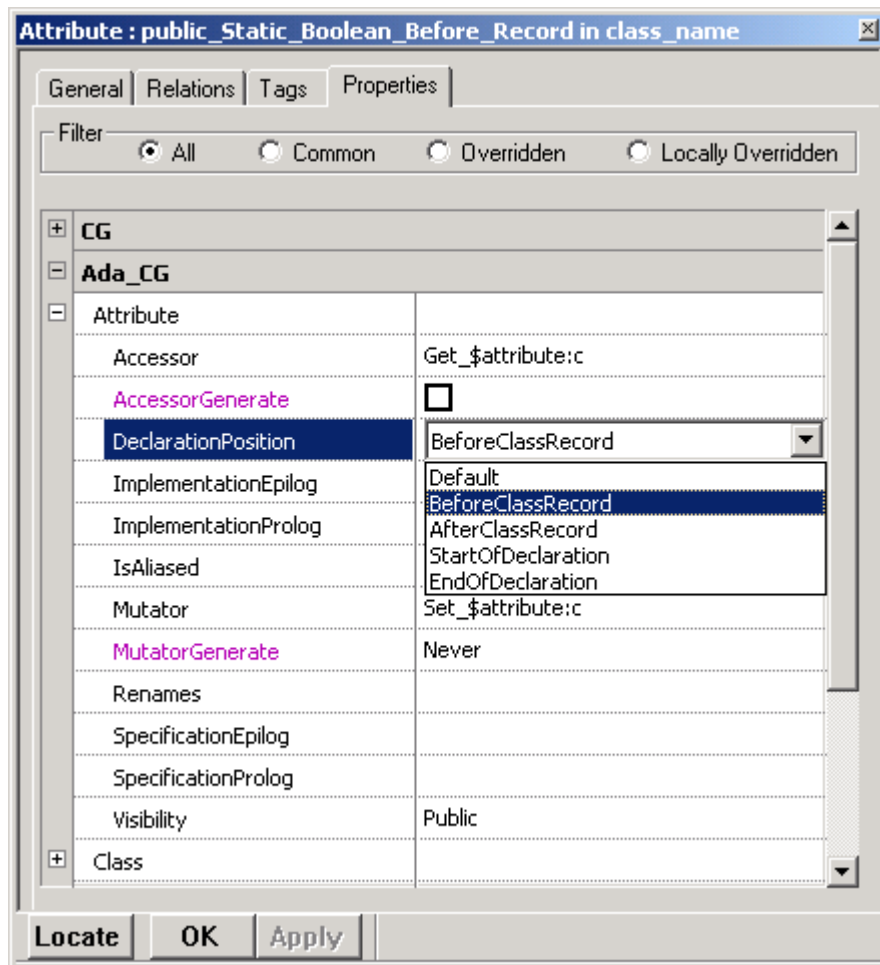


Figure 19 Setting the DeclarationPosition for a static attribute to BeforeClassRecord

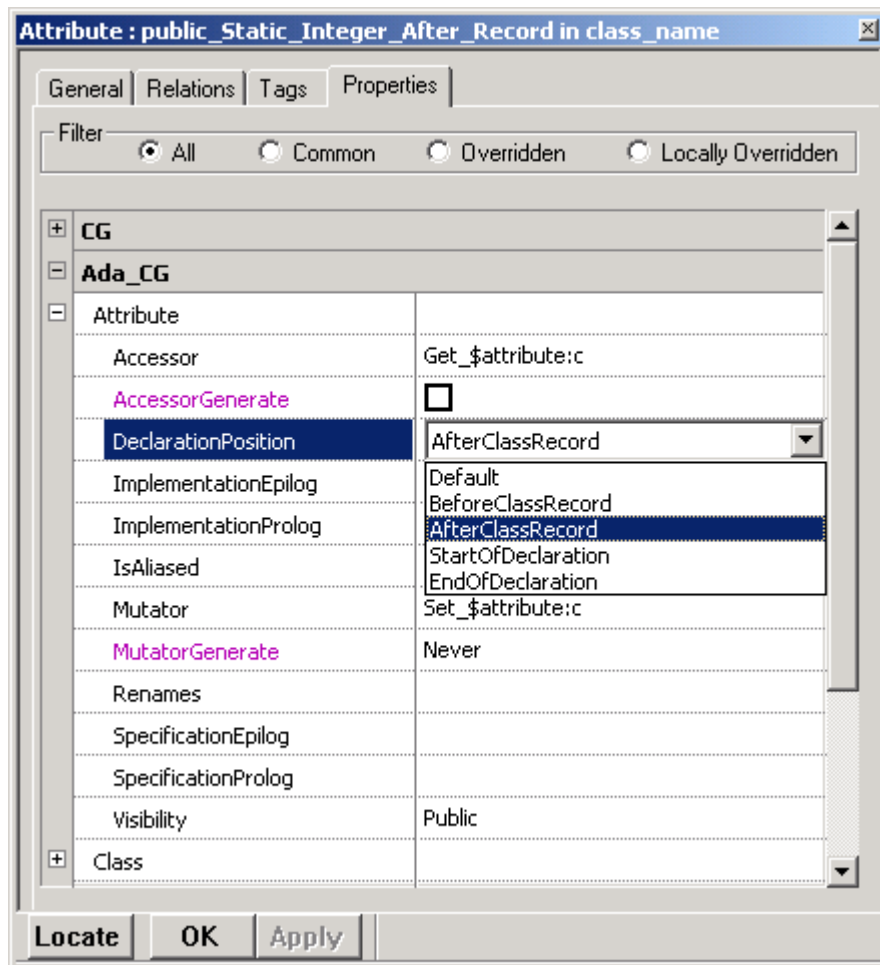


Figure 20 Setting the DeclarationPosition for a static attribute to AfterClassRecord

```

--++ class class_name
package class_name is

  type class_name_t;
  type class_name_acc_t is access all class_name_t;

  -- Public Variables/Constants -----
  public_Static_Boolean_Before_Record : Integer;  --++ attribute public_Static_Boolean_Before_Record

  type class_name_t is tagged null record;

  -- Public Variables/Constants -----
  public_Static_Integer_After_Record : Integer;  --++ attribute public_Static_Integer_After_Record

  --Public Fields/Variables accessors -----
private
end class_name;

```

Figure 21: Generated code for static attributes with overridden DeclarationPosition property

4.2.6. Non-Predefined Attribute Types

The attribute definitions can be entered directly in the “Ada declaration” field instead of choosing a predefined type. In this case, it might be necessary to set the two properties “Ada_CG.Attribute.AccessorGenerate” and “Ada_CG.Attribute.MutatorGenerate” to false so that the default accessors are not generated.

Attribute: my_String in class_name

General Relations Tags Properties

Name: my_String L

Stereotype:

Attribute type

☒ Use existing type

Type: Integer

Visibility

☒ Public ☐ Private

Multiplicity: 1 Ordered ☐

☐ Constant ☐ Reference ☐ Static

Initial Value: testtest

Description:

This is a string attribute

Locate OK Apply

Figure 22: Non-static attribute definition.

Attribute : my_Static_String in class_name

General Relations Tags Properties

Name: L

Stereotype:

Attribute type

☐ Use existing type

Ada Declaration:

Visibility

☒ Public ☐ Private

Multiplicity: ☐ Ordered

☐ Constant ☐ Reference ☒ Static

Initial Value: ...

Description:

Locate OK Apply

Figure 23: Static attribute definition.

```

--++ class class_name
package class_name is

    type class_name_t;
    type class_name_acc_t is access all class_name_t;

    type class_name_t is tagged

    record

        -- Fields --
        -- This is a string attribute
        my_String : Integer := "testtest"; --++ attribute my_String

    end record;

    -- Public Variables/Constants -----
    -- This is a public string variable
    my_Static_String : String (1..8) := "stpublic"; --++ attribute my_Static_String

    --Public Fields/Variables accessors -----

private
end class_name;

```

Figure 24: The package specification for non-predefined type attributes.

4.2.7. Guarded Attributes

Access to attributes can be guarded by setting the “isGuarded” property. There are two possible settings. One setting is “all” which guards both the accessor and mutator of the attribute. The other setting is “mutator” which will only guard the mutator.

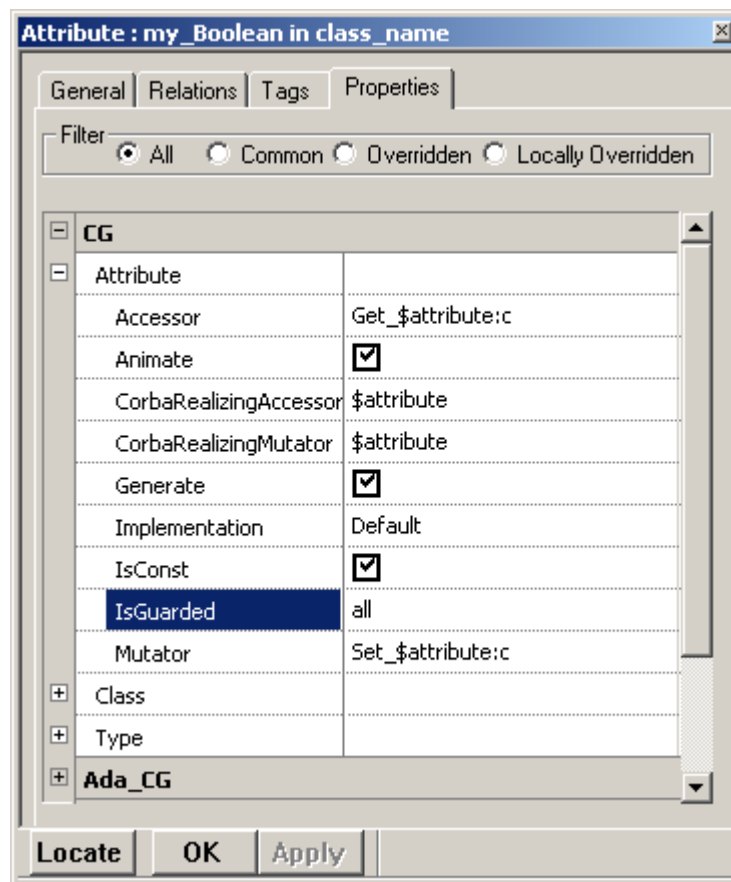


Figure 25: Guarding an Attribute.

When an attribute is guarded, a mutex is used to synchronize access to the attribute. Depending on the value of the “Ada_CG.<Class|Package>.UseAda83Framework” property of the attribute owner, an Ada83 task based Mutex or an Ada95 protected object based Mutex will be used.

4.2.8. <<Discriminant>> Attributes

By setting a class instance attribute or a struct attribute stereotype to <<Discriminant>>, it is possible to generate a discriminated record type.

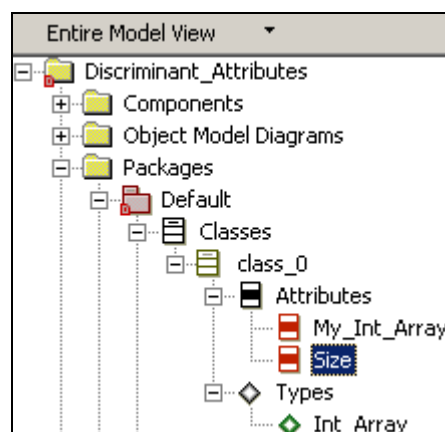


Figure 26 Modeling a class with a <<Discriminant>> attribute

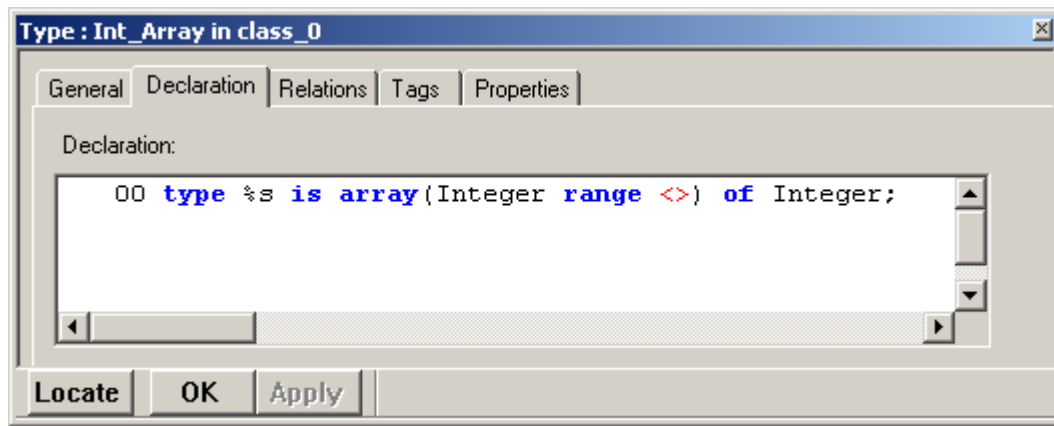


Figure 27 Defining an unconstrained array type

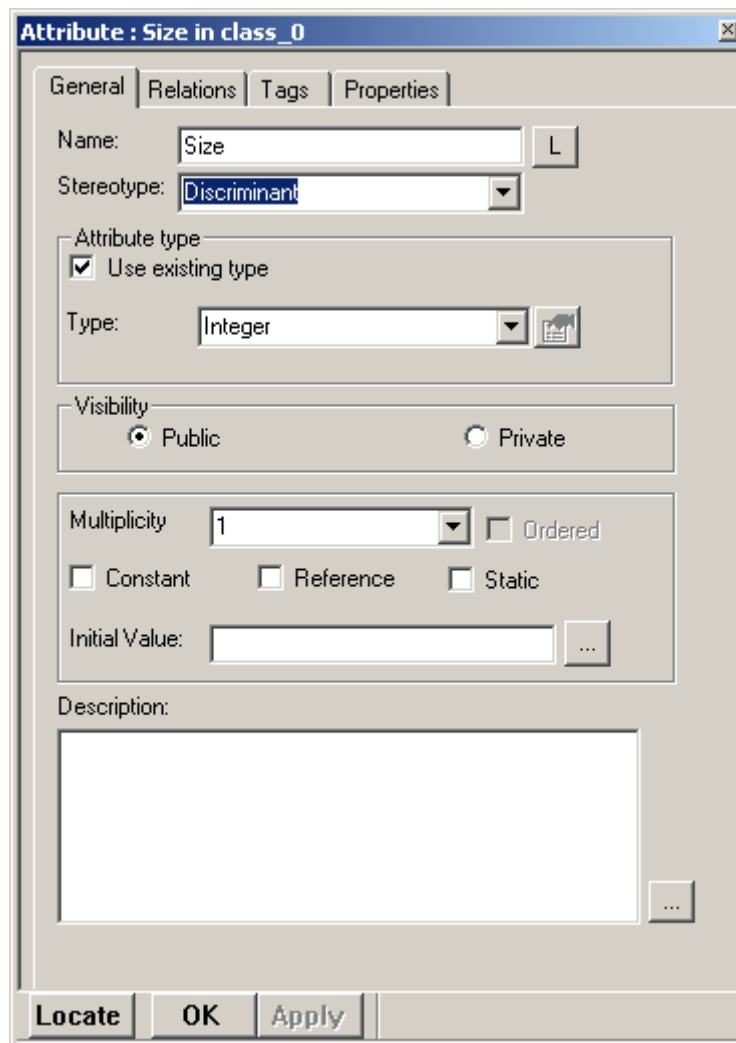


Figure 28 Setting an attribute stereotype to <<Discriminant>>

Attribute: My_Int_Array in class_0

General Relations Tags Properties

Name: L

Stereotype:

Attribute type

☐ Use existing type

Ada Declaration:

Visibility

☒ Public ☐ Private

Multiplicity: Ordered ☐

☐ Constant ☐ Reference ☐ Static

Initial Value: ...

Description:

Locate OK Apply

Figure 29 Defining an attribute with a type definition based on the class record type discriminant

```

--++ class class_0
package class_0 is

  type class_0_t (
    Size : Integer  --++ attribute Size
  );
  type class_0_acc_t is access all class_0_t;

  --Public types -----
  type Int_Array is array(Integer range <>) of Integer;

  type class_0_t (
    Size : Integer  --++ attribute Size
  ) is tagged

  record

    -- Fields --
    My_Int_Array : Int_Array(1..Size);  --++ attribute My_Int_Array

  end record;

private

end class_0;

```

Figure 30 Generated code for a class with a discriminant

Note that no setter is generated for attributes with a <<Discriminant>> stereotype, no matter the setting of its Ada_CG.Attribute.MutatorGenerate property.

4.2.9. Overriding and redefining discriminant attributes

If the attribute is a <<Discriminant>> non-static attribute and it has an initial value and it is defined as a <<Discriminant>> attribute in at least one of its parent classes then the attribute may be generated as a constraint on the parent discriminant, as a new discriminant hiding the one from the parent or as both.

This behavior is controlled by using the “Ada_CG.Attribute.RedefiningDiscriminantPolicy “ and “Ada_CG.Attribute.ParentDiscriminantValue” properties.

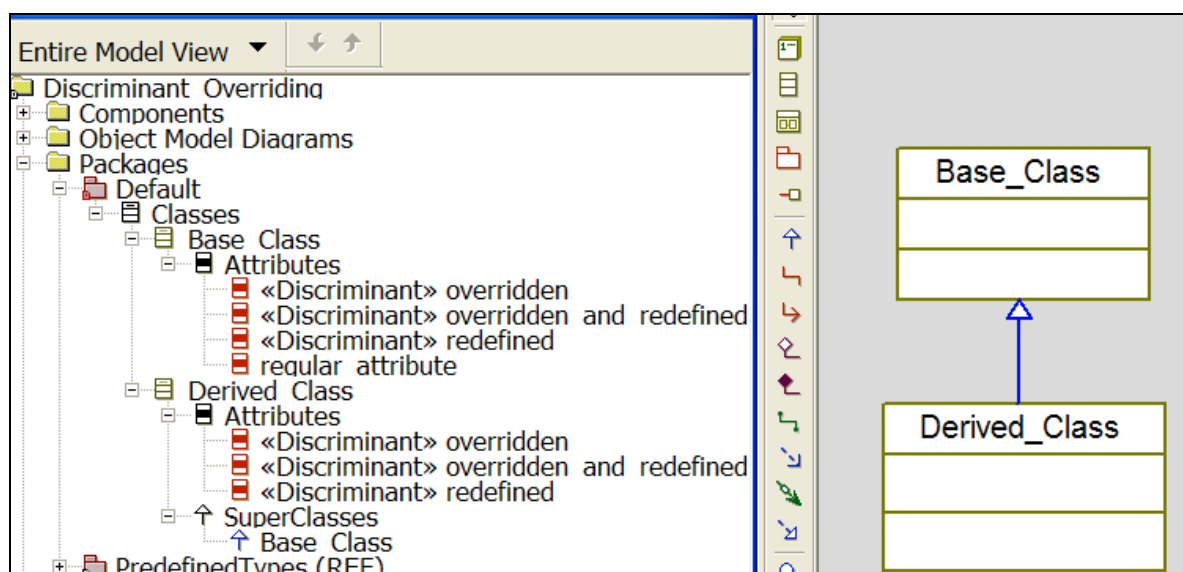


Figure 31 Class with overriding and redefining discriminant

Attribute : overridden in Derived_Class		
General Description Relations Tags Properties		
View Overridden ▾		
[-] Ada.CG		
[-] Attribute		
ParentDiscriminantValue		10
RedefiningDiscriminantPolicy		AsOverriding

Figure 32 Overriding discriminant

Attribute : overridden_and_redefined in Derived_Class		
General Description Relations Tags Properties		
View Overridden ▾		
[-] Ada.CG		
[-] Attribute		
ParentDiscriminantValue		100
RedefiningDiscriminantPolicy		AsNewAndOverriding

Figure 33 Overriding and redefining discriminant

Attribute : redefined in Derived_Class		
General Description Relations Tags Properties		
View Overridden ▾		
[-] Ada.CG		
[-] Attribute		
RedefiningDiscriminantPolicy		AsNew

Figure 34 Redefining discriminant

```
With Base_Class;

--++ class Derived_Class
package Derived_Class is

    type Derived_Class_t (
        overridden_and_redefined : Integer; --++ attribute overridden_and_redefined
        redefined : Integer --++ attribute redefined
    );
    type Derived_Class_acc_t is access all Derived_Class_t;

    type Derived_Class_t (
        overridden_and_redefined : Integer; --++ attribute overridden_and_redefined
        redefined : Integer --++ attribute redefined
    ) is new Base_Class.Base_Class_t(
        overridden => 10, --++ attribute overridden
        overridden_and_redefined => 100 --++ attribute overridden_and_redefined
    ) With null record;

private

end Derived_Class;
```

4.3. Operations

Operations created in Rhapsody will result in Ada functions if there is a return type, or procedures if there is not. In this example, *myOperation* will be a function with two parameters that returns an Integer. And *myProc* will be a procedure with one parameter.

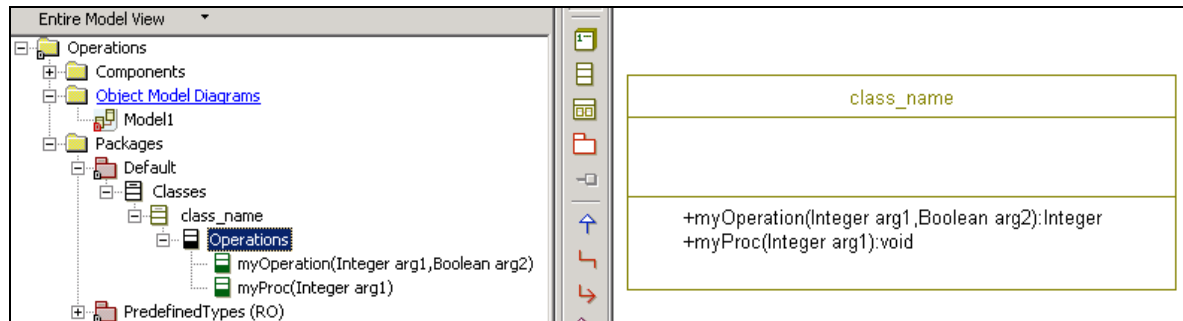


Figure 35: Operations defined on a class.

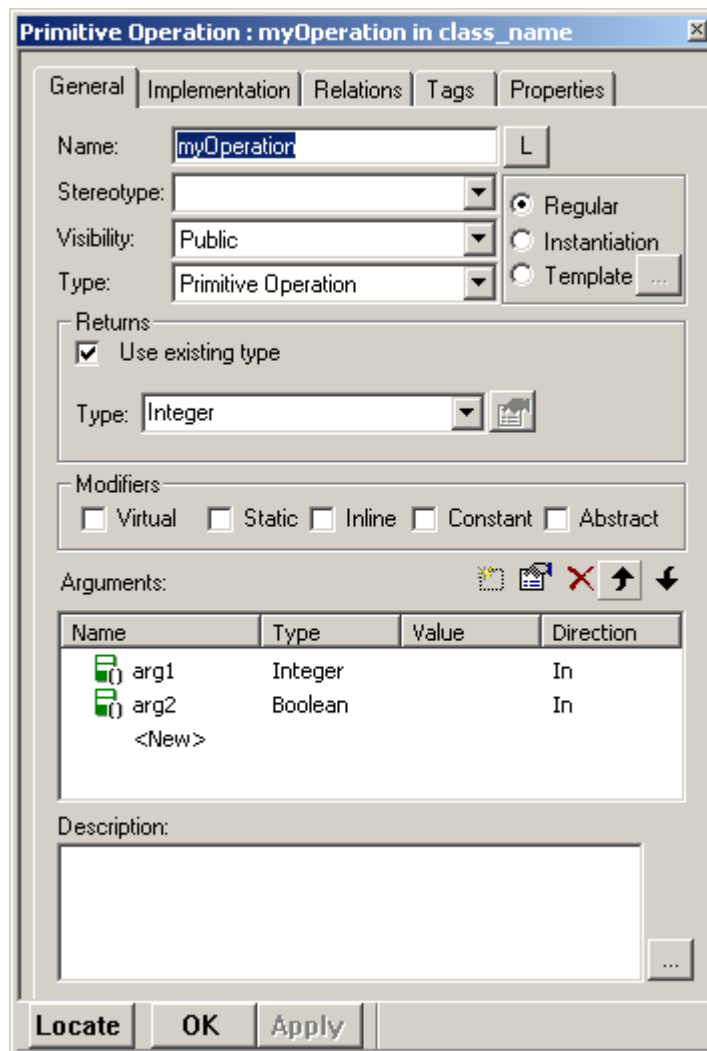
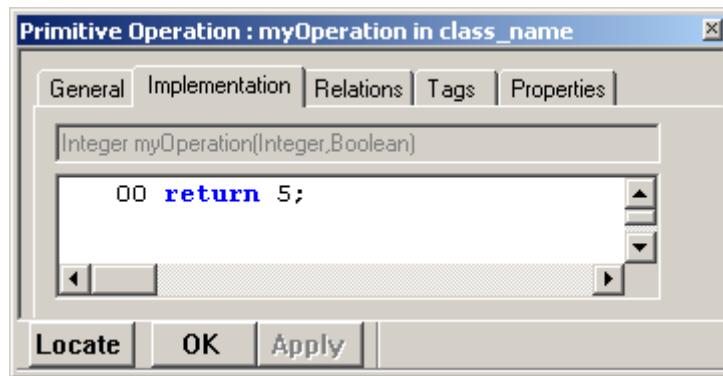
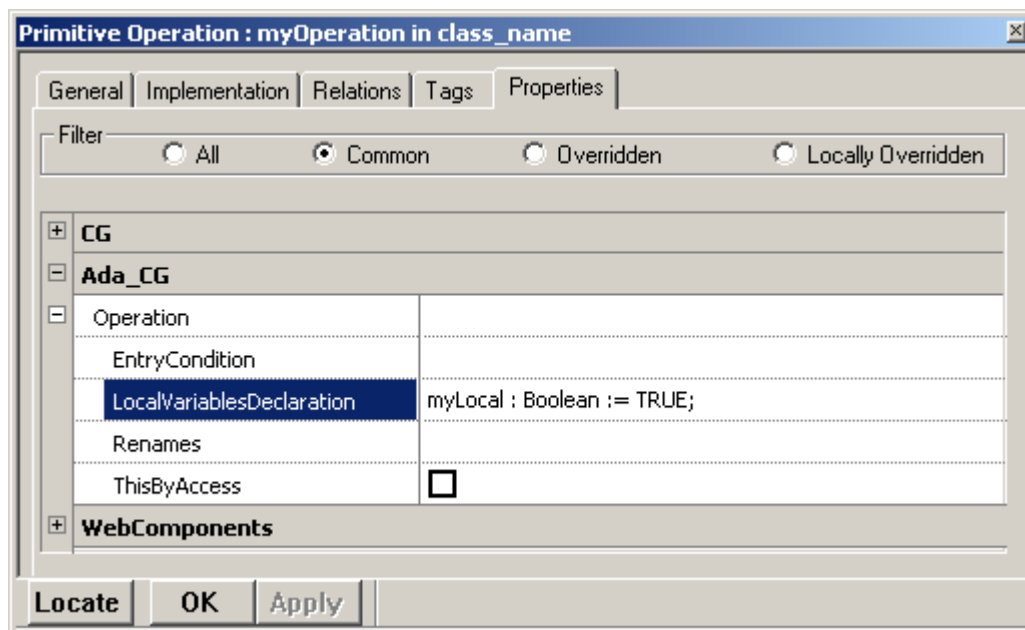


Figure 36: Operation Features

Figure 37: The implementation of *myOperation*.Figure 38: The local variables for *myOperation*.

```

--++ class class_name
package class_name is

    type class_name_t;
    type class_name_acc_t is access all class_name_t;

    type class_name_t is tagged null record;

    --Public Functions/Procedures section -----
    --++ operation myOperation(Integer, Boolean)
    function myOperation (this : in class_name_t;
        arg1 : in Integer;
        arg2 : in Boolean
    ) return Integer;

    --++ operation myProc(Integer)
    procedure myProc (
        arg1 : in Integer
    );

private

end class_name;

```

Figure 39: Operations in the package specification.

```

--++ class class_name
package body class_name is

    --Functions/Procedures section -----
    function myOperation (this : in class_name_t;
        arg1 : in Integer;
        arg2 : in Boolean
    ) return Integer is
        myLocal : Boolean := TRUE;
    begin
        --+[ operation myOperation(Integer, Boolean)
        return 5;
        --+]
    end myOperation;

    procedure myProc (
        arg1 : in Integer
    ) is
    begin
        null;
        --+[ operation myProc(Integer)

        --+]
    end myProc;

end class_name;

```

Figure 40: Operations in the package body.

In the operation bodies, the implementation provided in Rhapsody has been used for *myOperation*, but an appropriate default statement has been created for *myProc* because the implementation field in Rhapsody has been left blank. Any lines entered in this implementation field will replace this default statement.

4.3.1. Guarded operations

An operation can be made guarded by setting the “Concurrency” property to *guarded*.

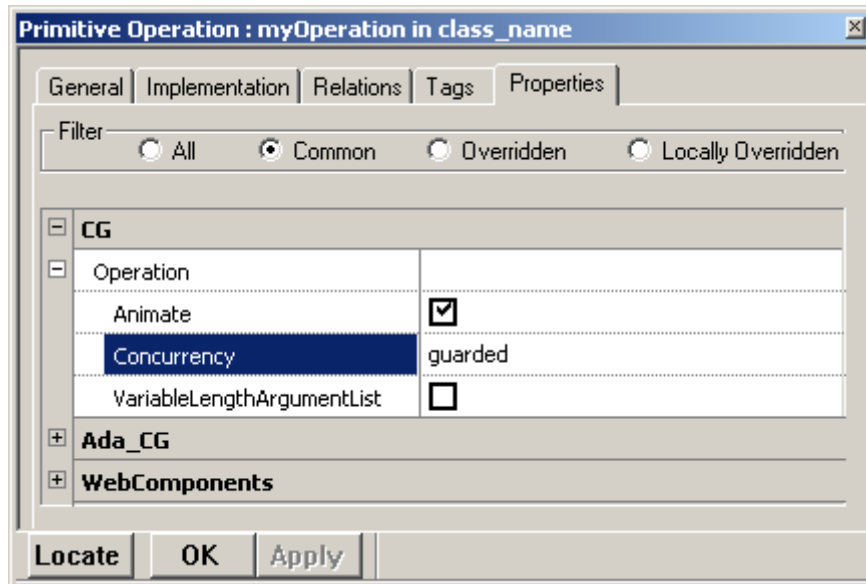


Figure 41: Making an Operation Guarded.

When an operation is guarded, a mutex is used to synchronize access to the operation. Depending on the value of the “Ada_CG.<Class|Package>.UseAda83Framework” property of the operation owner, an Ada83 task based Mutex or an Ada95 protected object based Mutex will be used.

4.3.2. Template operations and their instantiations

The mechanism for supporting template operations and their instantiations is very similar to the one available for template classes and their instantiation.

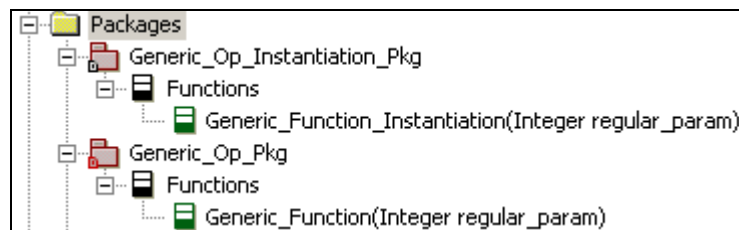


Figure 42 Modeling a template operation and a template operation instantiation

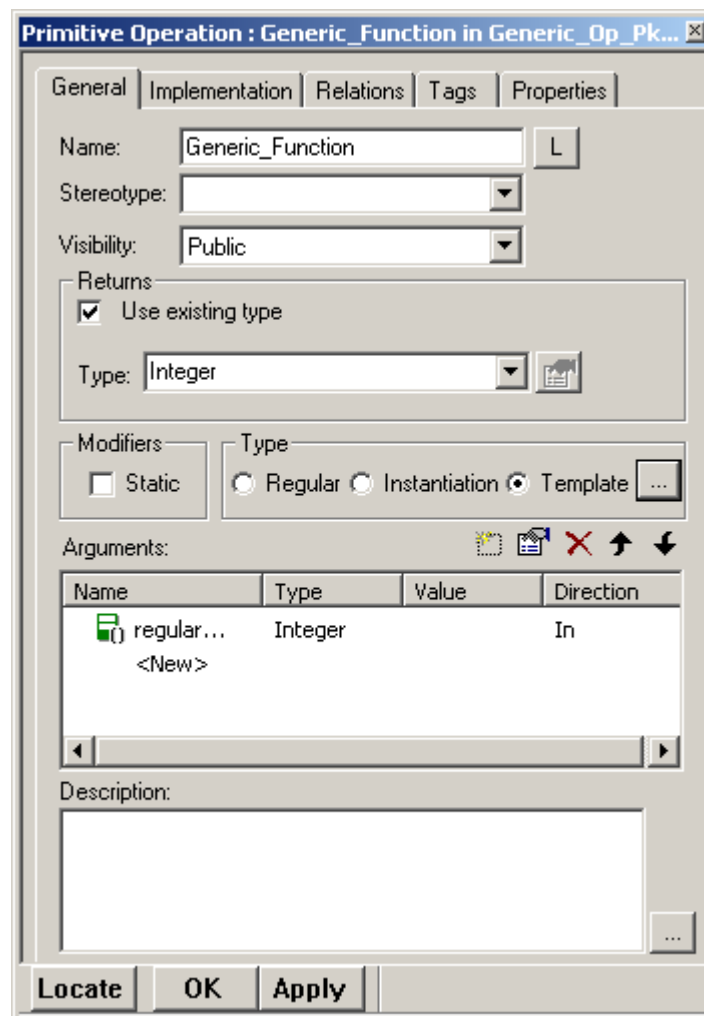


Figure 43 features of a template operation

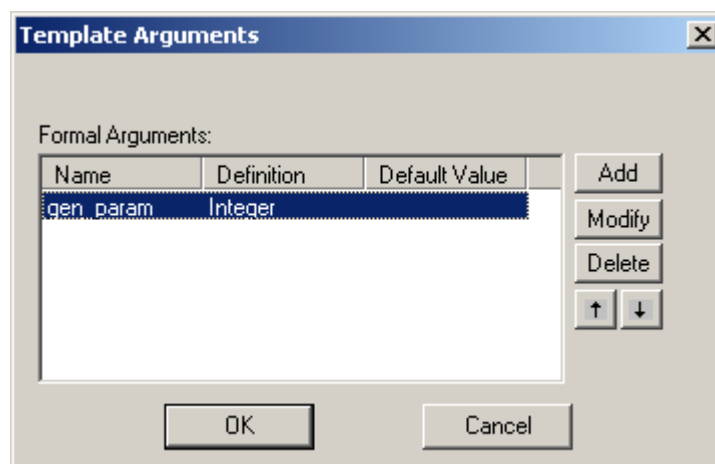


Figure 44 setting up template parameters for a template operation

```

--++ package Generic_Op_Pkg
package Generic_Op_Pkg is

    --Public Functions/Procedures section -----
    --++ operation Generic_Function(Integer)
    generic
        gen_param : Integer;
    function Generic_Function (
        regular_param : in Integer
    ) return Integer;

private

end Generic_Op_Pkg;

```

Figure 45 generated code for a template operation specification

```

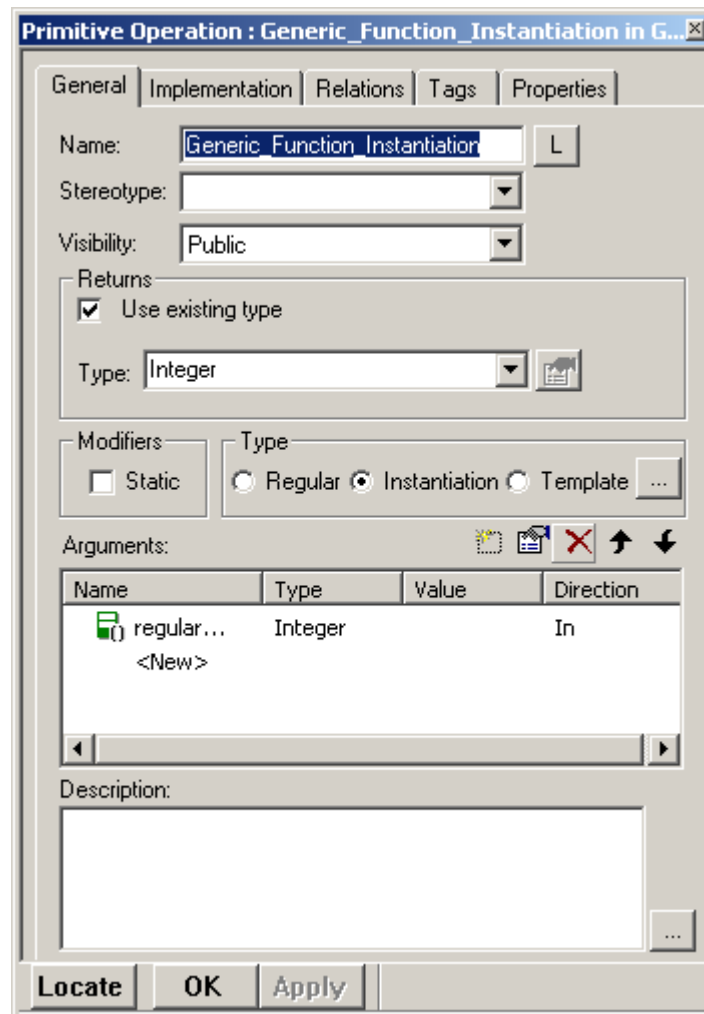
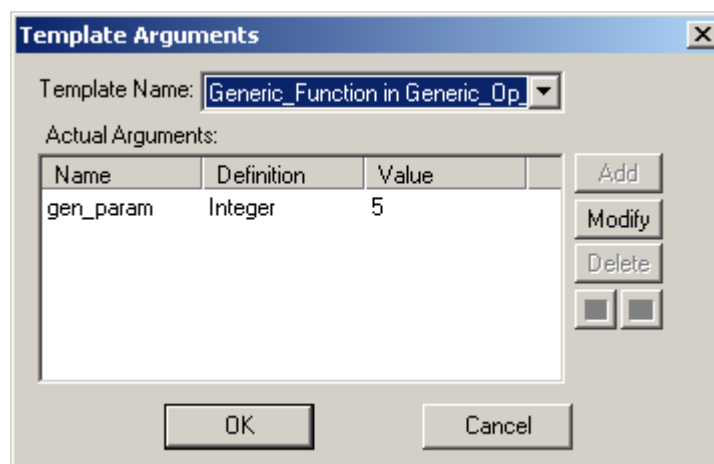
--++ package Generic_Op_Pkg
package body Generic_Op_Pkg is

    --Functions/Procedures section -----
    function Generic_Function (
        regular_param : in Integer
    ) return Integer is
        --+[ operation Generic_Function(Integer).Variables
        --+]
    begin
        return 0;
        --+[ operation Generic_Function(Integer)
        --+]
    end Generic_Function;

end Generic_Op_Pkg;

```

Figure 46 generated code for a template operation implementation

**Figure 47 features of a template operation instantiation****Figure 48 setting up template arguments for a template operation instantiation**

```

--++ package Generic_Op_Instantiation_Pkg
package Generic_Op_Instantiation_Pkg is

    --Public Functions/Procedures section -----
    --++ operation Generic_Function_Instantiation(Integer)
    function Generic_Function_Instantiation is new Generic_Op_Pkg.Generic_Function(
        gen_param => 5
    );

private

end Generic_Op_Instantiation_Pkg;

```

Figure 49 generated code for a template operation instantiation

4.3.3. Access parameters

Ada95 introduced the concept of access parameters. In order to set the mode of a parameter to be "access", as opposed to "in", "out", or "in out", first the package in which is defined the operation has to generate Ada95 code (and not Ada83), second you will need to edit the properties of the parameter to set the AsAccess property to true.

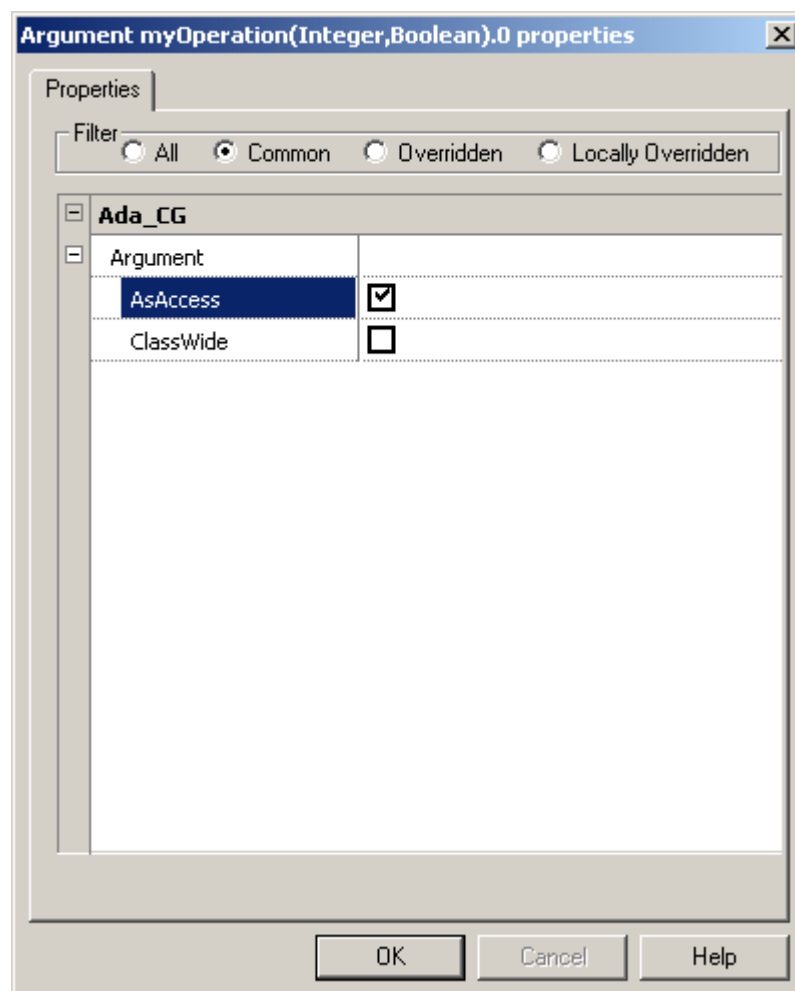


Figure 50 Making a parameter passing mode "access"

You can also choose to pass the this parameter as an access mode parameter for a non-static operation, to do this you need to edit the properties of the operation to set the ThisByAccess property to true.

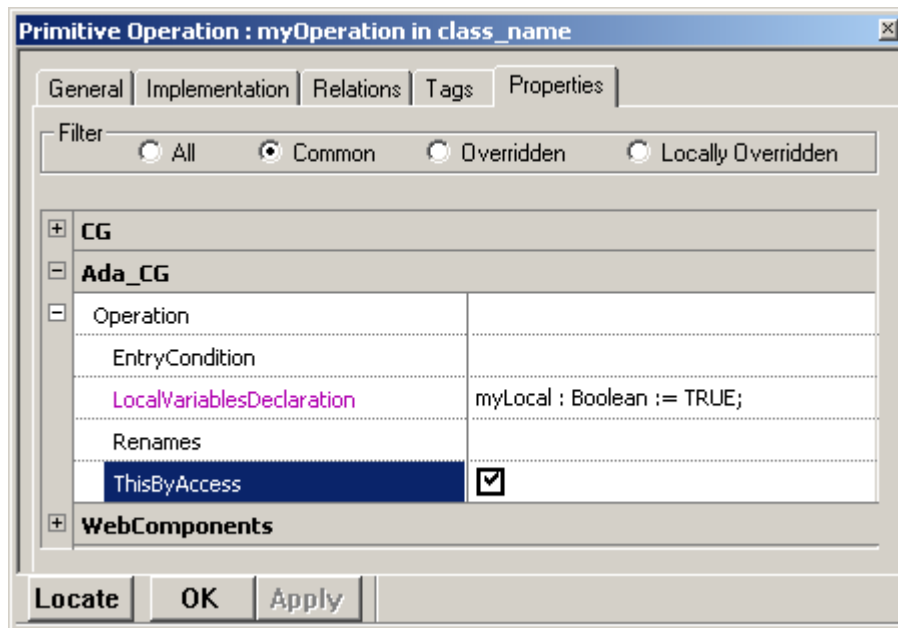


Figure 51 Making an operation this parameter passing mode "access"

```

type class_name_t;
type class_name_acc_t is access all class_name_t;

type class_name_t is tagged null record;

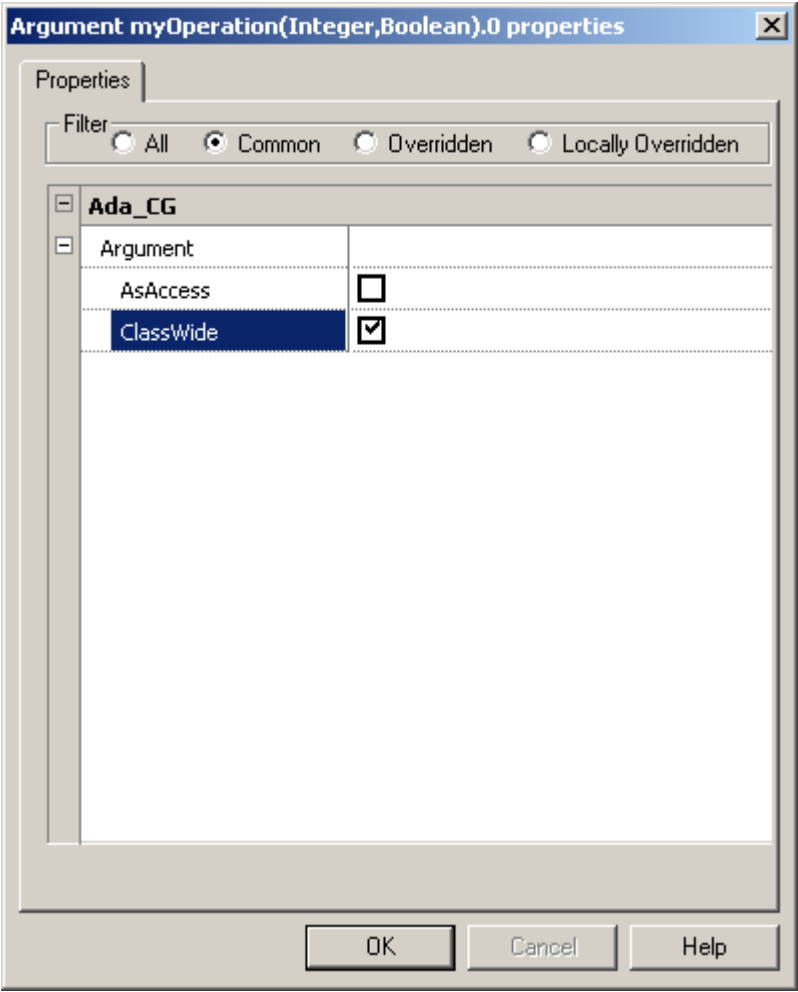
--Public Functions/Procedures section -----
--++ operation myOperation(Integer, Boolean)
function myOperation (this : access class_name_t;
  arg1 : access Integer;
  arg2 : in Boolean
) return Integer;

```

Figure 52 Operation using access mode parameters

4.3.4. Class-wide parameters

In order to specify whether a parameter is to be passed class-wide or not, you will need to set its "ClassWide" property to true.



4.4. Dependencies

4.4.1. <<Usage>> dependencies

Dependencies stereotyped as <<Usage>> in Rhapsody create “With” statements in the generated Ada packages. If the “Ada_CG.Dependency.CreateUseStatement” property is set to “Use”, a “Use” statement will also be created for the target package. If it is set to “UseType”, a “Use Type” statement will also be created for the target type. By default, the “With” and “Use” statements will appear in the package specification. They can be moved to the package body by setting the “CG.Dependency.UsageType” property to “Implementation”.

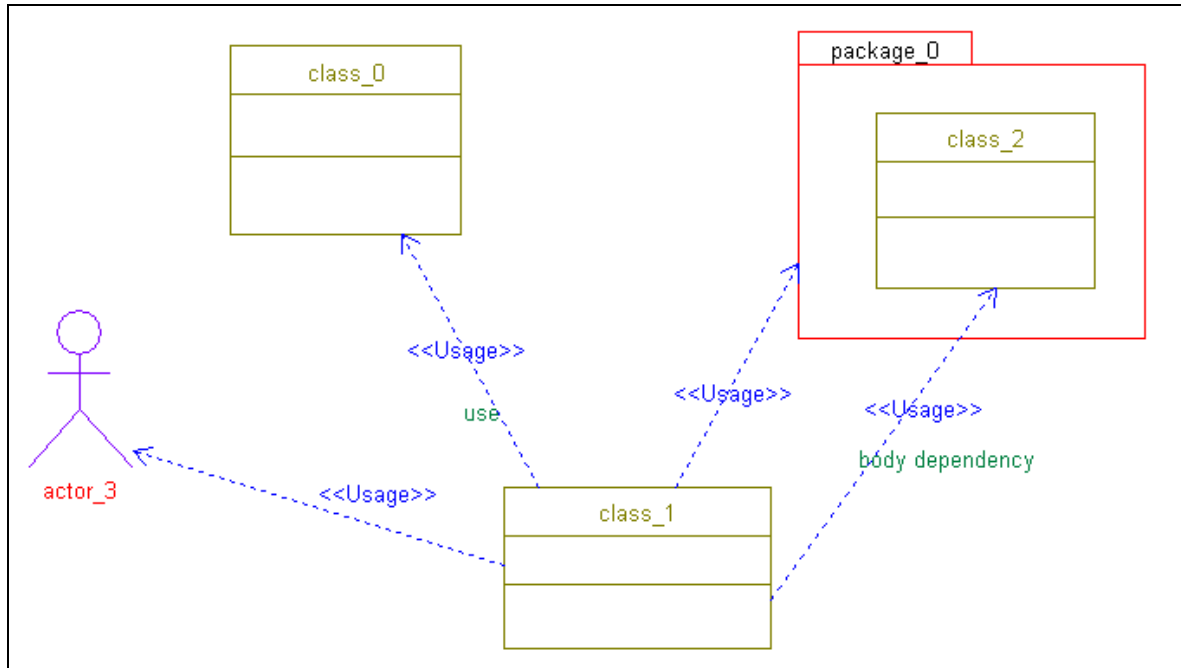


Figure 53: <<Usage>> dependencies in IBM® Rational® Rhapsody® in Ada.

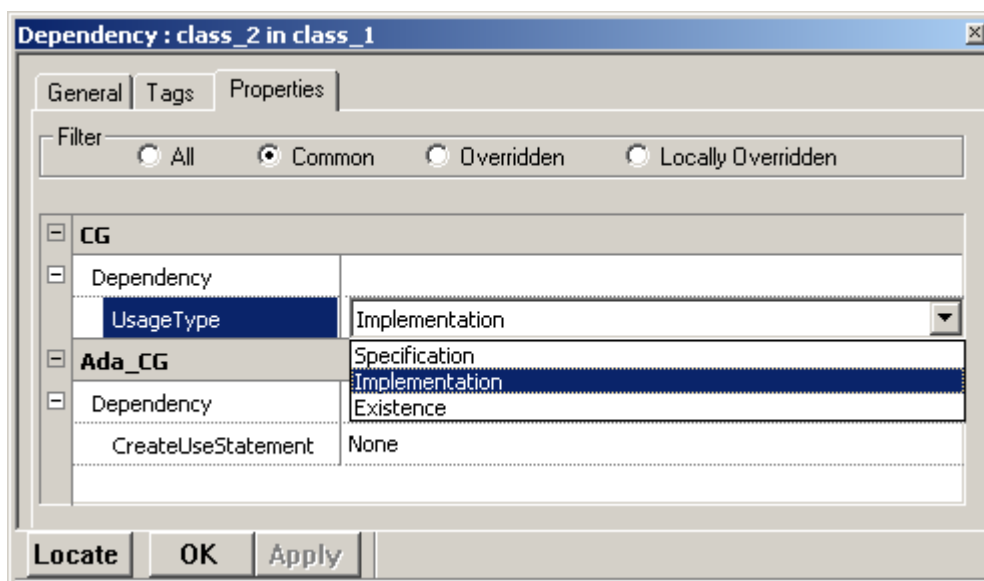


Figure 54: An implementation dependency.

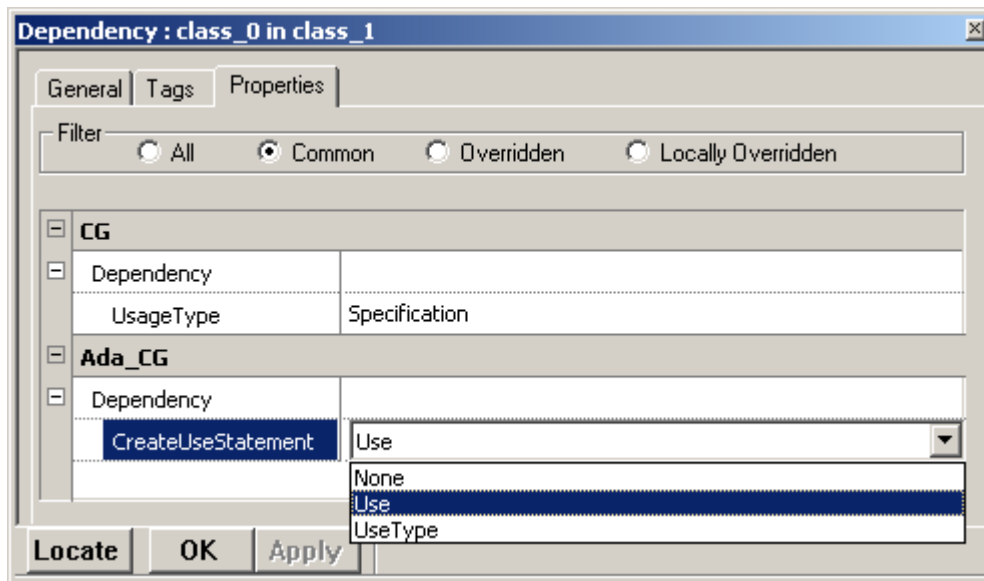


Figure 55: Creating a "Use" statement.

```

With actor_3;
With class_0;
With package_0;
Use class_0;

--++ class class_1
package class_1 is

    type class_1_t;
    type class_1_acc_t is access all class_1_t;

    type class_1_t is tagged null record;

    --Public Functions/Procedures section -----
    --++ operation Message_0()
    procedure Message_0 (this : in out class_1_t);

private

end class_1;

```

Figure 56: The package specification for dependencies.

```

With package_0.class_2;

--++ class class_1
package body class_1 is

    --Functions/Procedures section -----
    procedure Message_0 (this : in out class_1_t) is
    begin
        null;
        --+[ operation Message_0()

        --+}
    end Message_0;

end class_1;

```

Figure 57: The package body for dependencies.

Note that elaboration pragmas can be generated for the supplier class or package of the dependency in the client class or package by setting the appropriate properties on the dependency :

- Ada_CG.Dependency.GeneratePragmaElaborate
- Ada_CG.Dependency.GeneratePragmaElaborateAll

4.4.2. <<Renames>> dependencies

Dependencies stereotyped as <<Renames> in Rhapsody create “renames” statements in the generated Ada packages.

Valid <<Renames>> dependencies can be modeled between any two model elements of the same kind among the following ones :

- Packages
- Classes
- Operations
- Attributes
 - Defined on a package
 - Defined on a class with a static modifier

Be aware that using this feature on classes limits what you can do with the renaming class. More specifically :

- You cannot derive other classes from
- Adding attributes or operations to it has no effect on the generated code

Note that for operations :

- Signatures have to be compatible
- It is possible to have a “renaming as spec” or a “renaming as body” behavior depending on the setting of the “CG.Dependency.UsageType” property (Specification or Implementation).

Only <<renames>> dependencies between classes and packages can be drawn on the Object Model Diagrams of Rhapsody. In order to model <<renames>> dependencies between two attributes or two operations, one has to use the context menu in the Rhapsody browser.

4.5. Actors

Actors generate exactly the same code as classes.

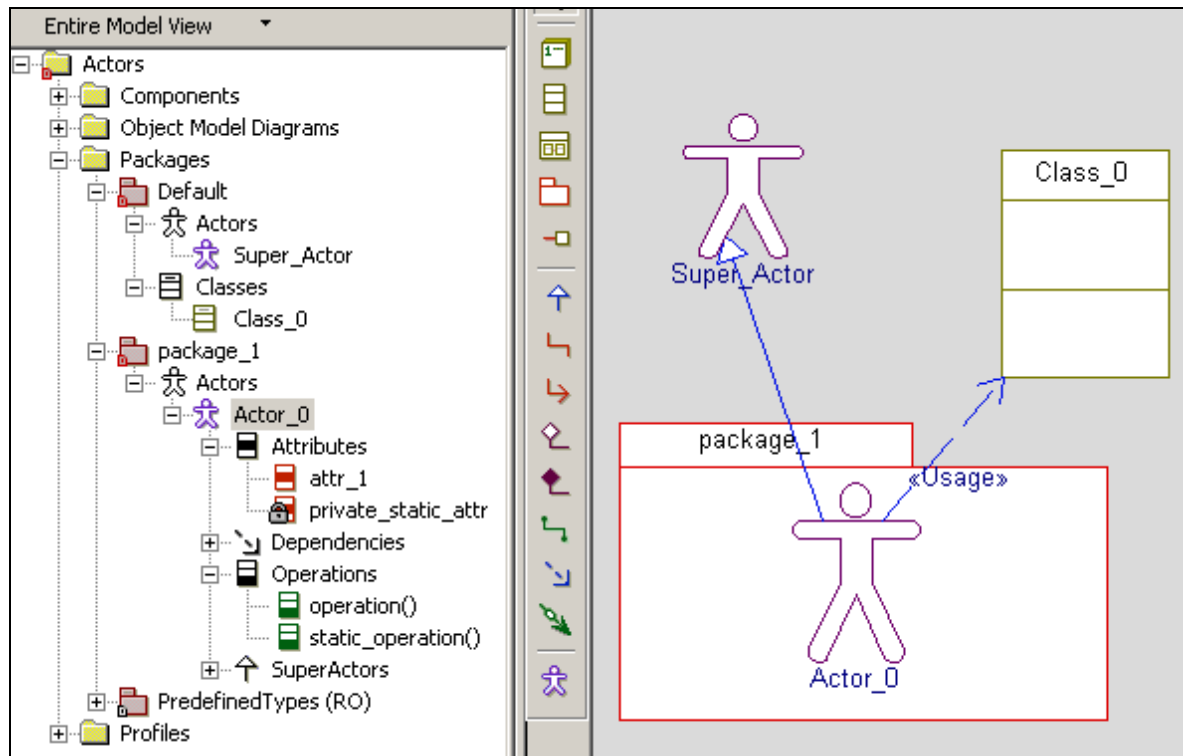


Figure 58: An Actor in Rhapsody.

```

With Class_0;
With Super_Actor;

--++ actor Actor_0
package package_1.Actor_0 is

    type Actor_0_t;
    type Actor_0_acc_t is access all Actor_0_t;

    type Actor_0_t is new Super_Actor.Super_Actor_t with

    record

        -- Fields --
        attr_1 : Integer;    --++ attribute attr_1

    end record;

    -- Public Variables/Constants -----
    private_static_attr : Integer;    --++ attribute private_static_attr

    --Public Functions/Procedures section -----
    --++ operation operation()
    procedure operation (this : in out Actor_0_t);

    --++ operation static_operation()
    procedure static_operation;

    --Public Fields/Variables accessors -----
    function get_attr_1 (this : in Actor_0_t) return Integer;
        pragma inline (get_attr_1);

    procedure set_attr_1 (this : in out Actor_0_t; value : in Integer);
        pragma inline (set_attr_1);

private

    --Private Fields/Variables accessors -----
    function get_private_static_attr return Integer;
        pragma inline (get_private_static_attr);

    procedure set_private_static_attr (value : in Integer);
        pragma inline (set_private_static_attr);

end package_1.Actor_0;

```

Figure 59: Package specification for an Actor.

```

--++ actor Actor_0
package body package_1.Actor_0 is

    --Functions/Procedures section -----
    procedure operation (this : in out Actor_0_t) is
        --+[ operation operation().Variables

        --+]
    begin
        null;
        --+[ operation operation()

        --+]
    end operation;

    procedure static_operation is
        --+[ operation static_operation().Variables

        --+]
    begin
        null;
        --+[ operation static_operation()

        --+]
    end static_operation;

    --Fields/Variables accessors -----
    function get_attr_1(this : in Actor_0_t) return Integer is
    begin
        return this.attr_1;
    end get_attr_1;

    procedure set_attr_1 (this : in out Actor_0_t; value : in Integer) is
    begin
        this.attr_1 := value;
    end set_attr_1;

    function get_private_static_attr return Integer is
    begin
        return private_static_attr;
    end get_private_static_attr;

    procedure set_private_static_attr (value : in Integer) is
    begin
        private_static_attr := value;
    end set_private_static_attr;

end package_1.Actor_0;

```

Figure 60: Package body for an Actor.

4.6. Packages

Like classes, packages in Rhapsody will also be represented as Ada packages. A package in Rhapsody can have functions and variables, which will be handled in the same manner as static operations and static attributes on a class. A package can also have initialization code.

In this example, the package specification will be named “myPackage.ads” and the package body will be “myPackage.adb”.

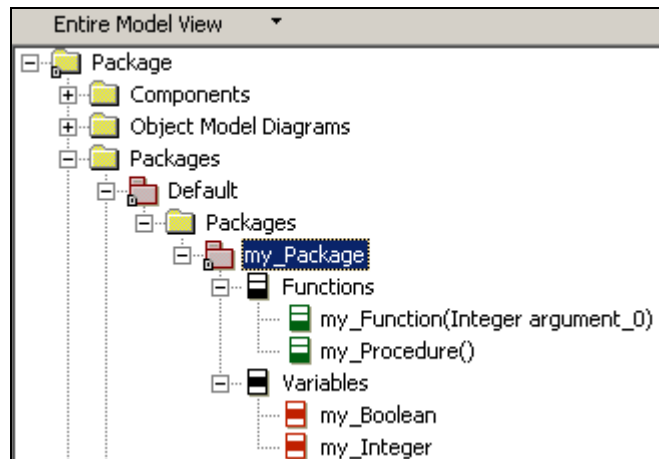


Figure 61: A package defined in Rhapsody.

```
--++ package Default::my_Package
package my_Package is

    -- Public Variables/Constants -----
    my_Integer : Integer; --++ attribute my_Integer

    my_Boolean : Boolean; --++ attribute my_Boolean

    --Public Functions/Procedures section -----
    --++ operation my_Function(Integer)
    function my_Function (
        argument_0 : in Integer
    ) return Boolean;

    --++ operation my_Procedure()
    procedure my_Procedure;

    --Public Fields/Variables accessors -----
    function get_my_Integer return Integer;
        pragma inline (get_my_Integer);

    procedure set_my_Integer (value : in Integer);
        pragma inline (set_my_Integer);

    function get_my_Boolean return Boolean;
        pragma inline (get_my_Boolean);

    procedure set_my_Boolean (value : in Boolean);
        pragma inline (set_my_Boolean);

private

end my_Package;
```

Figure 62: The package specification for a Rhapsody package.

```

--++ package Default::my Package
package body my_Package is

    --Functions/Procedures section -----
    function my_Function (
        argument_0 : in Integer
    ) return Boolean is
        --+[ operation my_Function(Integer).Variables
        --+]
    begin
        return true;
        --+[ operation my_Function(Integer)
        --+]
    end my_Function;

    procedure my_Procedure is
        --+[ operation my_Procedure().Variables
        --+]
    begin
        null;
        --+[ operation my_Procedure()
        --+]
    end my_Procedure;

    --Fields/Variables accessors -----
    function get_my_Integer return Integer is
    begin
        return my_Integer;
    end get_my_Integer;

    procedure set_my_Integer (value : in Integer) is
    begin
        my_Integer := value;
    end set_my_Integer;

    function get_my_Boolean return Boolean is
    begin
        return my_Boolean;
    end get_my_Boolean;

    procedure set_my_Boolean (value : in Boolean) is
    begin
        my_Boolean := value;
    end set_my_Boolean;

end my_Package;

```

Figure 63: The package body for a Rhapsody package.

4.6.1. Child Packages

When working with Ada 95, classes and packages are also used as the namespace for classes and packages contained within them. This containment is used to create child packages. In Ada 83, this containment does not have any effect.

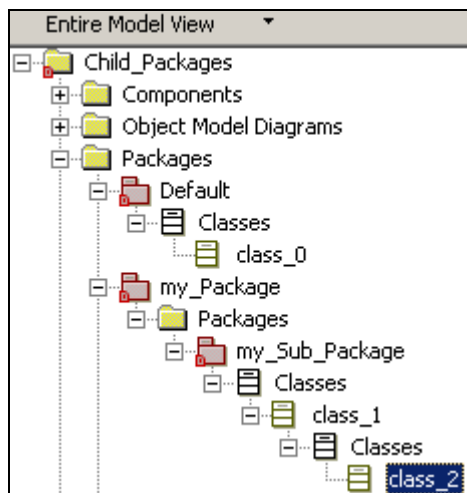


Figure 64: Packages and classes used as namespaces.

```
--++ class class_1::class_2
package my_Package.my_Sub_Package.class_1.class_2 is

    type class_2_t;
    type class_2_acc_t is access all class_2_t;

    type class_2_t is tagged null record;

private

end my_Package.my_Sub_Package.class_1.class_2;
```

Figure 65: The package specification for class_2.

Default\class_0.ads
my_Package\my_Sub_Package\class_1.ads
my_Package\my_Sub_Package\class_1\class_2.ads
Default\Default.ads
my_Package\my_Package.ads
my_Package\my_Sub_Package\my_Sub_Package.ads

Figure 66: The resulting files including the namespaces.

In this example, the Ada package name is prefixed by the parent names. For example, *class_2* is contained in *class_1*, which is found in *my_Sub_Package*, which itself is located inside of *my_Package*. Therefore the package name for *class_2* is *my_Package.my_Sub_Package.class_1.class_2*. This package will be found in the configuration directory in *my_Package/my_Sub_Package/class_1*.

Also notice that *class_0* is located in the “Default” package, and therefore is not considered as a child package.

4.6.2. Nested Packages

Child packages are in the namespace of their parent, but they are defined in separate files. Nested packages are not only in the namespace of their parent, but they are also defined in the same files as their parent.

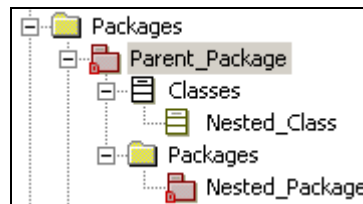


Figure 67: Example of a nested package and a nested class.

In order to generate nested packages, one has to set the `Ada_CG.Package.IsNested` property for a package or the `Ada_CG.Class.IsNested` property for a class.

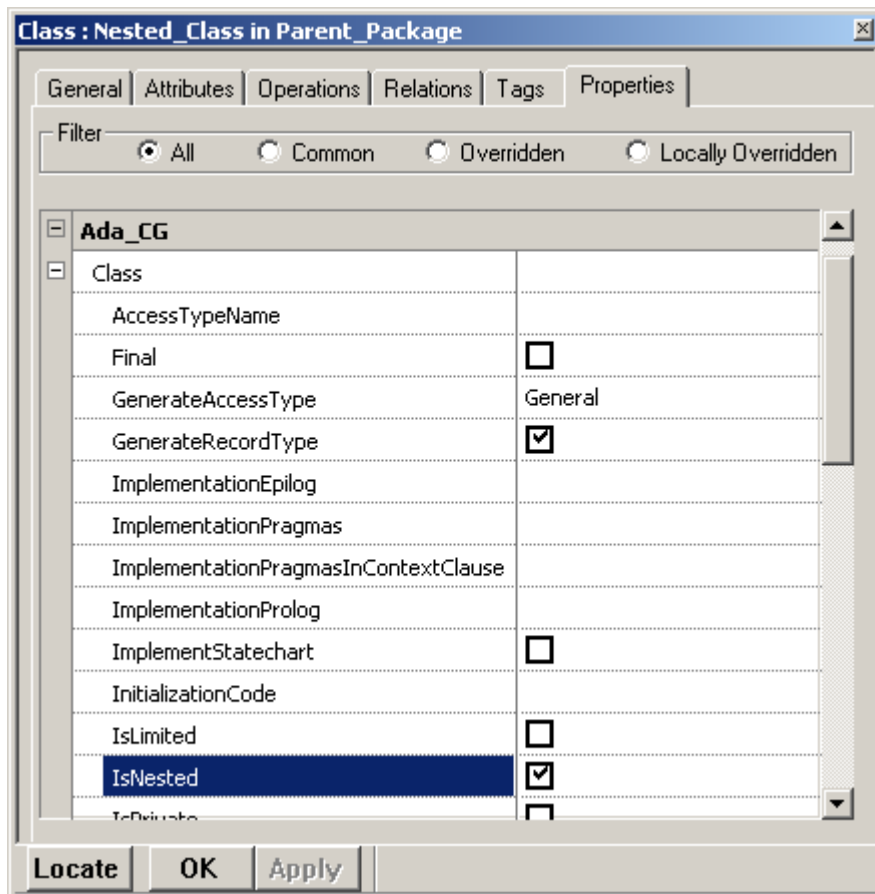


Figure 68: Setting a class to be generated as a nested package.

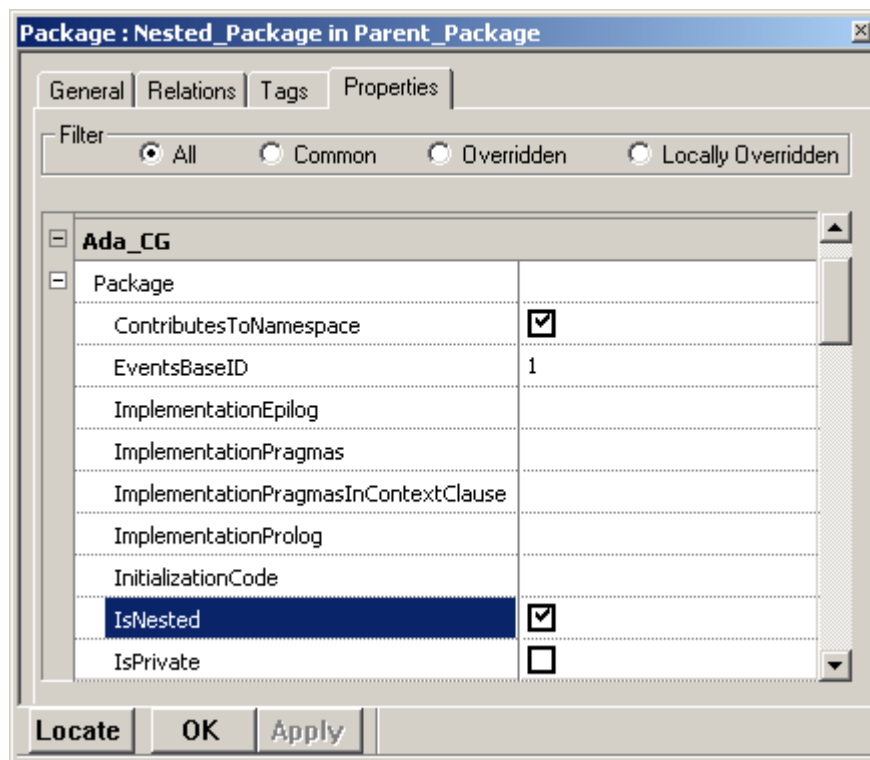


Figure 69: Setting a package to be generated as a nested package.

To determine in which section of the parent package the specification of the nested package will be generated, one can use the `Ada_CG.Package.NestingVisibility` property for a package, or the `Ada_CG.Class.NestingVisibility` property for a class.

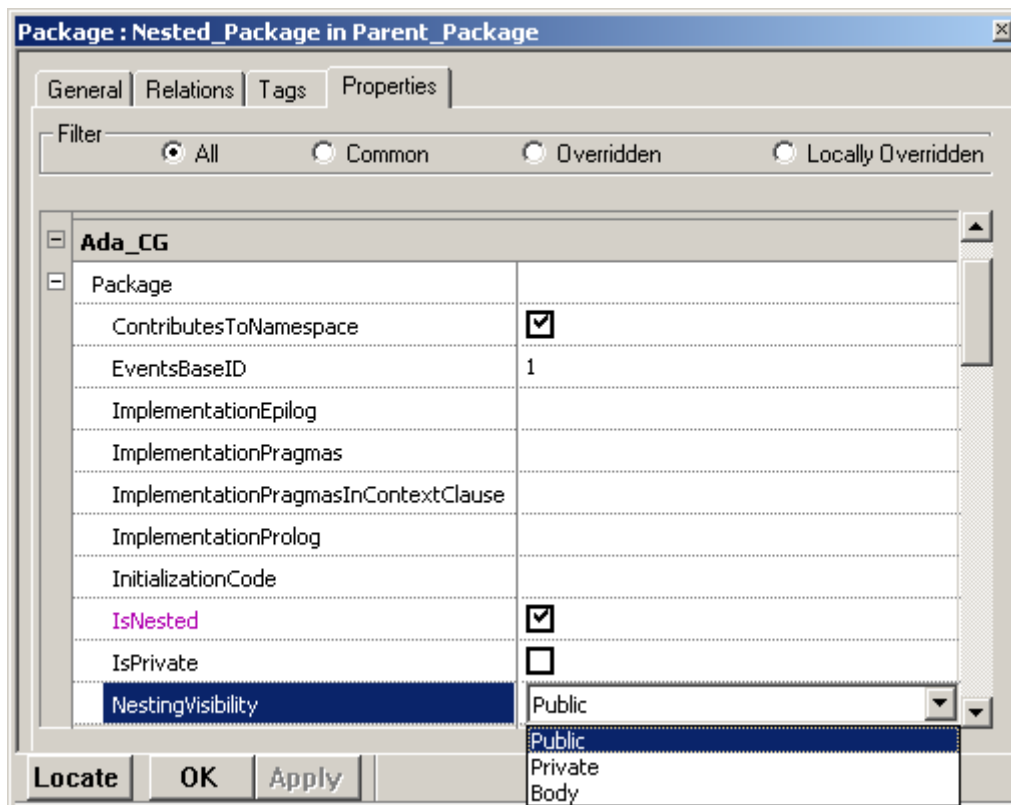


Figure 70: Controlling the location of the specification of a nested package

Note that any package or class defined inside of a package or class that is itself nested will be generated as a nested package too.

4.6.3. Private Packages

Packages and classes can be defined as private via the use of the Ada_CG.Package.IsPrivate property for packages and Ada_CG.Class.IsPrivate property for classes.

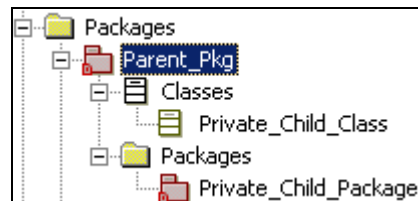


Figure 71: Exemple of a private package and a private class.

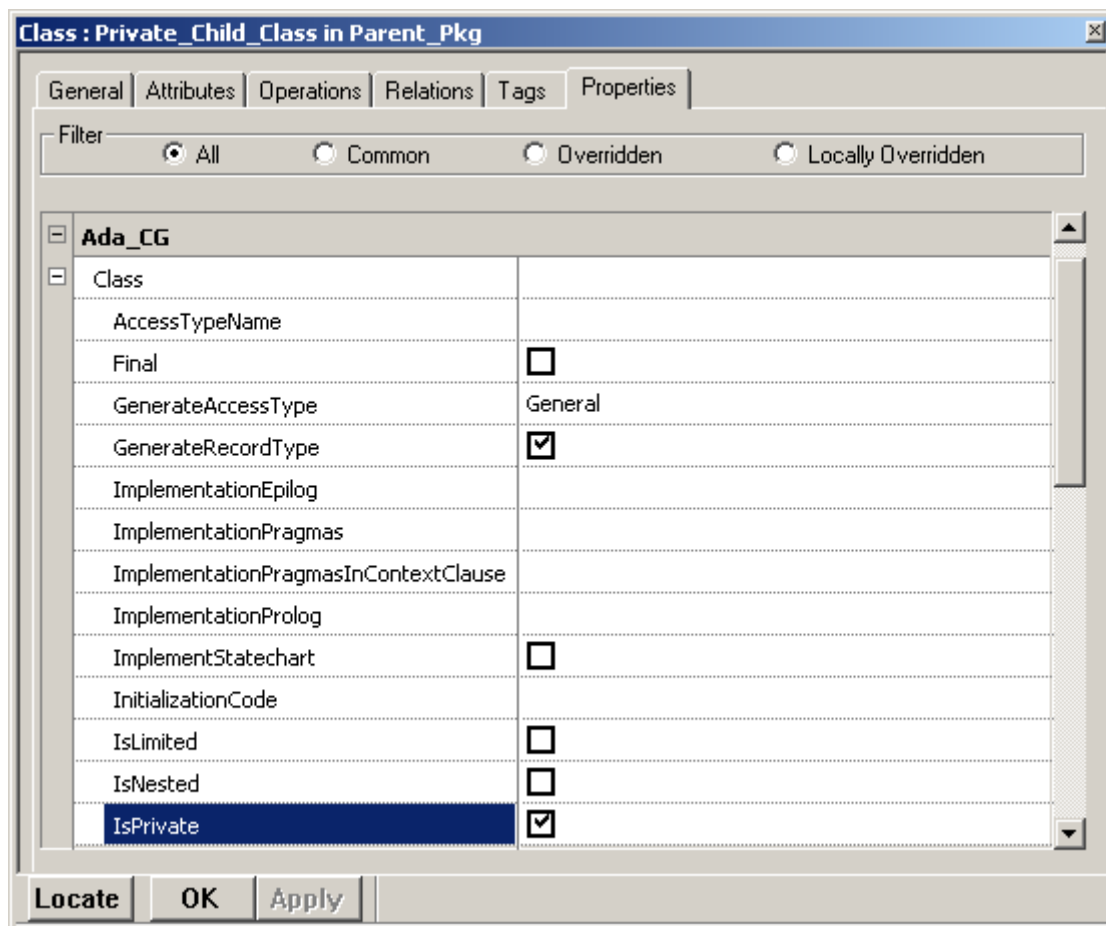


Figure 72: Setting a class to be generated as a private package

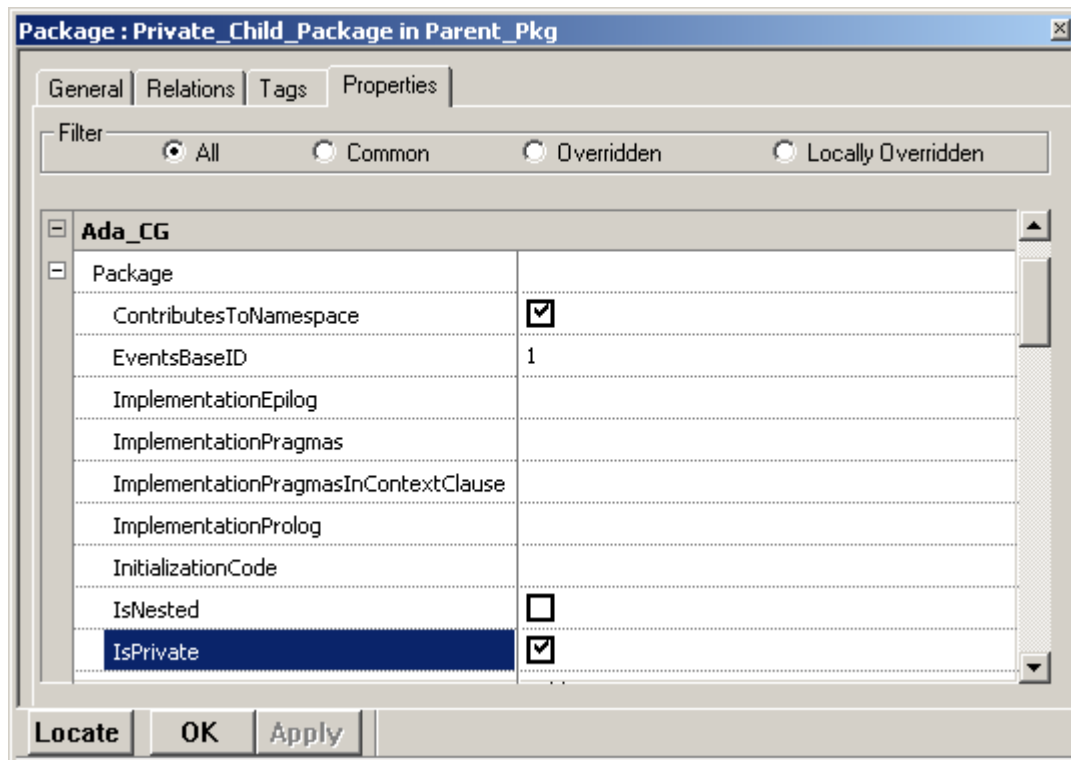


Figure 73: Setting a package to be generated as a private package

```
--++ class Private_Child_Class
private package Parent_Pkg.Private_Child_Class is

    type Private_Child_Class_t;
    type Private_Child_Class_acc_t is access all Private_Child_Class_t;

    type Private_Child_Class_t is tagged null record;

private

end Parent_Pkg.Private_Child_Class;
```

Figure 74: Specification of a private class

```
--++ package Parent_Pkg::Private_Child_Package
private package Parent_Pkg.Private_Child_Package is

private

end Parent_Pkg.Private_Child_Package;
```

Figure 75: Specification of a private package

Note that a nested class or package cannot be private.

4.6.4. Elaboration Pragmas

Via the use of appropriate tags, different pragmas can be generated for a class or a package

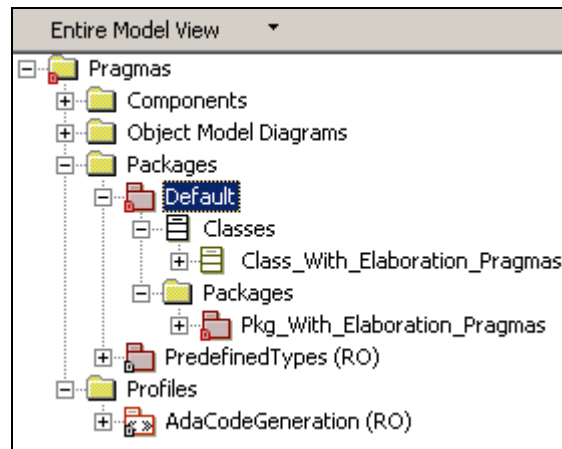


Figure 76: Example of a class and a package with elaboration pragmas

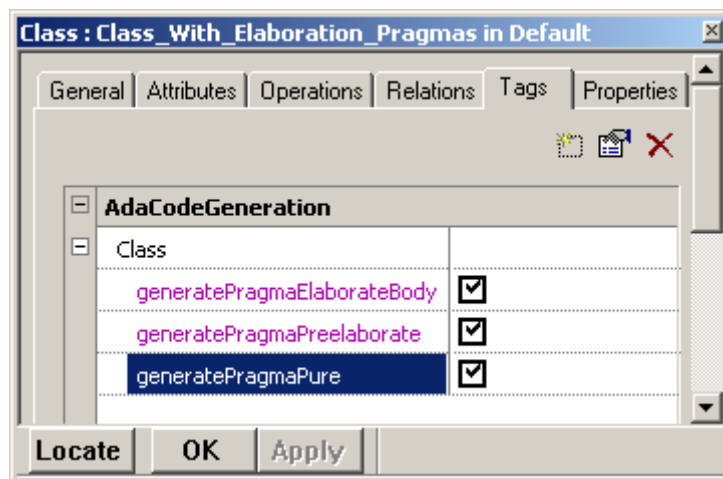


Figure 77: Enabling generation of elaboration pragmas on a class

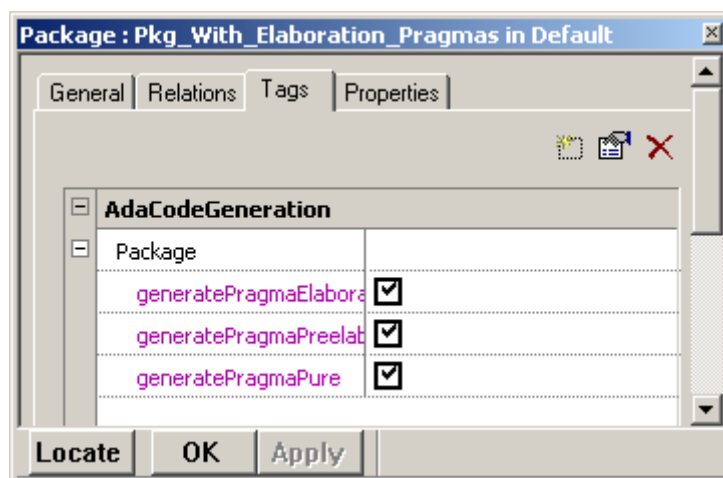


Figure 78: Enabling generation of elaboration pragmas on a package

```

--++ class Class_With_Elaboration_Pragmas
package Class_With_Elaboration_Pragmas is

    pragma elaborate_body;
    pragma preelaborate;
    pragma pure;

    type Class_With_Elaboration_Pragmas_t;
    type Class_With_Elaboration_Pragmas_acc_t is access all Class_With_Elaboration_Pragmas_t;

    type Class_With_Elaboration_Pragmas_t is tagged null record;

private

end Class_With_Elaboration_Pragmas;

```

Figure 79: Specifcation of a class with elaboration pragmas

```

--++ package Default::Pkg_With_Elaboration_Pragmas
package Pkg_With_Elaboration_Pragmas is

    pragma elaborate_body;
    pragma preelaborate;
    pragma pure;

private

end Pkg_With_Elaboration_Pragmas;

```

Figure 80: Specifcation of a package with elaboration pragmas

Please read the section on [<<Usage>> dependencies](#) to see how to generate “elaborate” and “elaborate_all” pragmas.

Note that other pragmas can be generated for a class or a package via the use of the following properties :

- For a class
 - Ada_CG.Class.SpecificationPragmas
 - Ada_CG.Class.SpecificationPragmasInContextClause
 - Ada_CG.Class.ImplementationPragmas
 - Ada_CG.Class.ImplementationPragmasInContextClause
- For a package
 - Ada_CG.Package.SpecificationPragmas
 - Ada_CG.Package.SpecificationPragmasInContextClause
 - Ada_CG.Package.ImplementationPragmas
 - Ada_CG.Package.ImplementationPragmasInContextClause

4.6.5. <<Container>> Packages

Any class or package that is defined within a package stereotyped as <<Container>> will not include the package in its namespace.

In the following example, Package_1 is stereotyped <<Container>>.

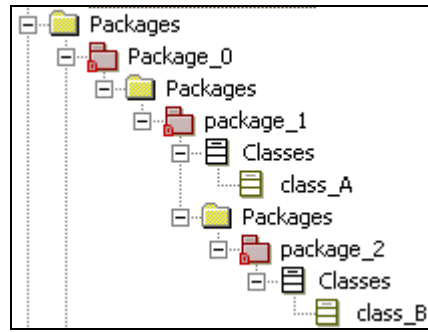


Figure 81: A Sample <<Container>> Package.

In this example, every package is generated, but the namespaces for Class_A, Package_2, and Class_B do not include Package_1.

4.7. Types

4.7.1. Type declaration

Types can be created in either packages or classes. In both cases, the definition of the type is taken from the “Ada declaration” field. In the declaration, a “%s” can be inserted to represent the name of the type.

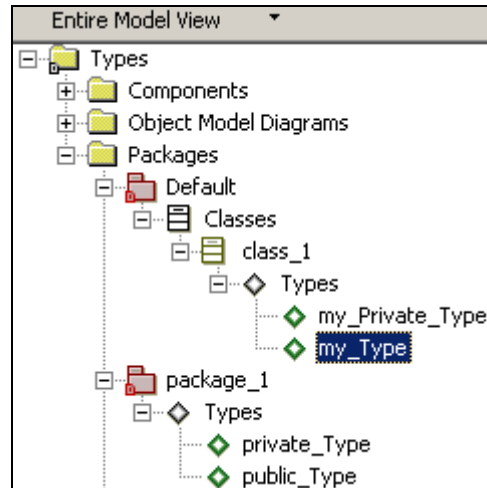


Figure 82: Types defined in Rhapsody.

The dialog box 'Type : my_Private_Type in class_1' has tabs for 'General', 'Declaration', 'Relations', 'Tags', and 'Properties'. The 'General' tab is active. It contains fields for 'Name' (set to 'my_Private_Type'), 'Stereotype' (empty), 'Kind' (set to 'Language'), and 'Description' (containing 'Definition of my private type.'). At the bottom are 'Locate', 'OK', and 'Apply' buttons.

The same dialog box is shown with the 'Declaration' tab active. The 'Declaration' field contains the Ada code: `00 subtype %s is Integer range 1..5;`. The 'Locate', 'OK', and 'Apply' buttons are at the bottom.

Figure 83: The declaration of a private type on a class.

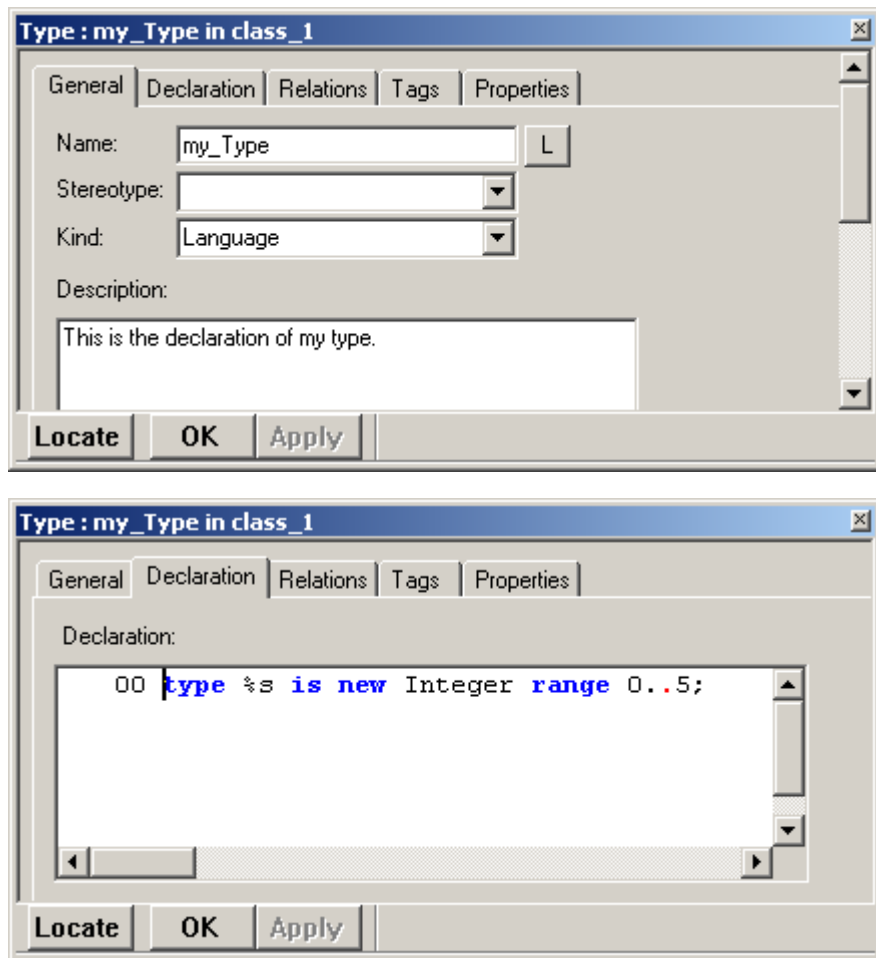


Figure 84: The declaration of a public type on a class.

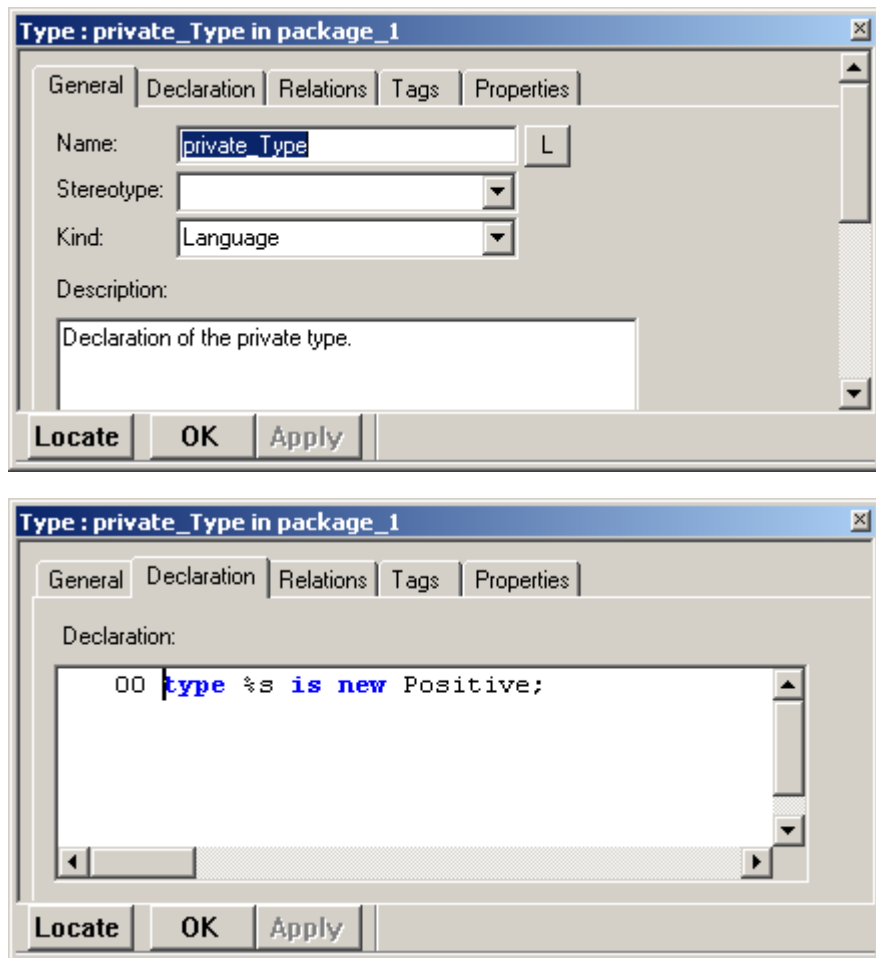


Figure 85: The declaration of a private type on a package.

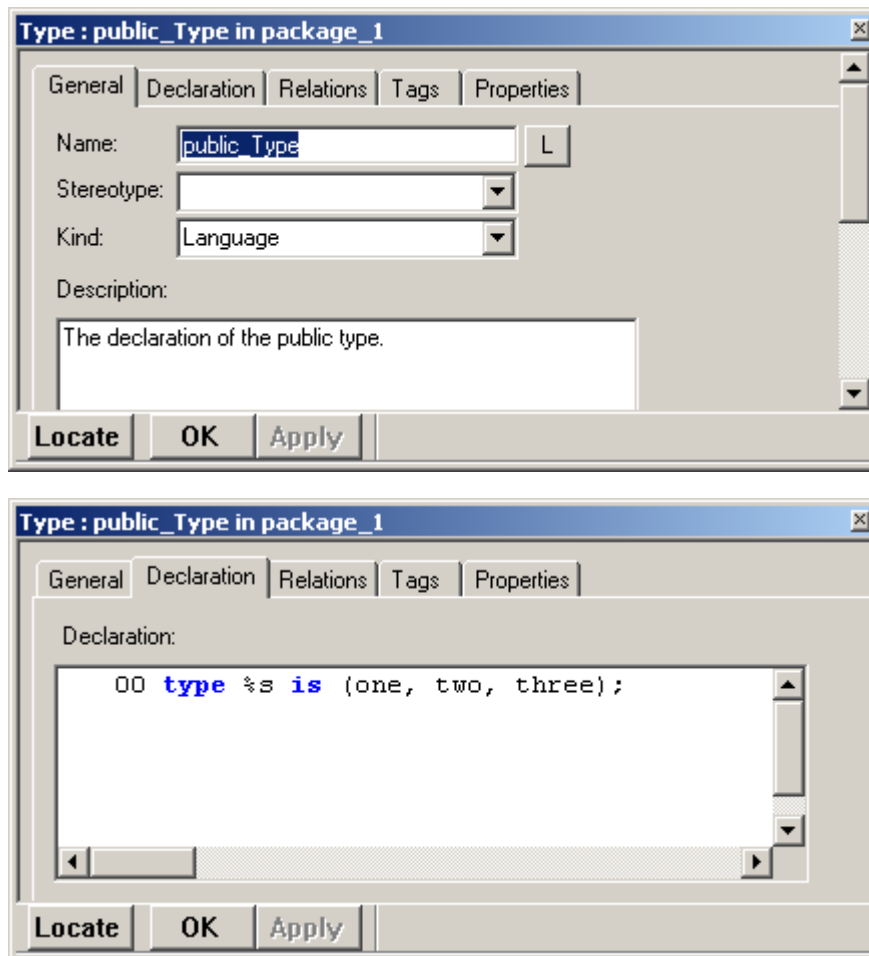


Figure 86: The declaration of a public type on a package.

4.7.2. Type visibility

A type definition can appear in the public or private portion of the resulting package specification, or in the package body.

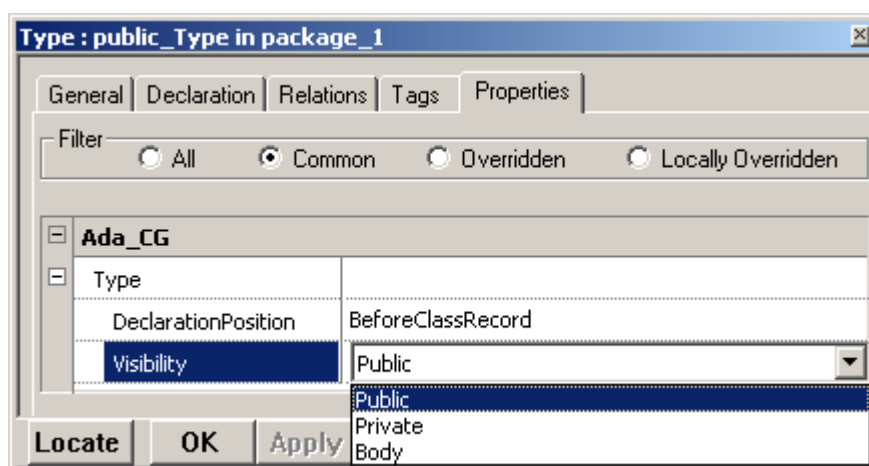


Figure 87: Controlling the visibility of a type.

```

--++ class class_1
package class_1 is

    type class_1_t;
    type class_1_acc_t is access all class_1_t;

    --Public types -----
    -- This is the declaration of my type.
    type my_Type is new Integer range 0..5;

    type class_1_t is tagged limited null record;

private

    --Private types -----
    -- Definition of my private type.
    subtype my_Private_Type is Integer range 1..5;

end class_1;

```

Figure 88: The package specification for a class with types.

```

--++ package package_1
package package_1 is

    --Public types -----
    -- The declaration of the public type.
    type public_Type is (one, two, three);

private

    --Private types -----
    -- Declaration of the private type.
    type private_Type is new Positive;

end package_1;

```

Figure 89: The package specification for a package with types.

4.7.3. Type declaration position

In order to provide some degree of control over the declaration order of types, the `Ada_CG.Type.DeclarationPosition` property can be used. Its behavior is very similar to the one of the `Ada_CG.Attribute.DeclarationPosition` property for attributes, with the following exceptions :

- There is no “default” value for this property on types
- If an attribute and a type have a similar value for their respective `declarationPosition` properties, then the attribute will be generated before the type declaration.

4.7.4. Type defined as a class

A type can be also defined with a class with a stereotype. This enables to have more visibility on type relations. The stereotypes are defined in the profile `AdaCodeGenerator`.

Main stereotype is “Type”. It is applicable on a class. This stereotype has a tag “IsSubtype”. If this tag is set to true, then the class will define a subtype.

Any type has a reference to another Ada type. This reference is represented by a dependency with the stereotype “New”. The dependency can be set to a Rhapsody type or a Rhapsody typed class.

Basic use cases :

Definition of a type

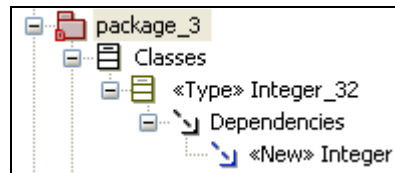


Figure 90: Representation of a typed class

```
--Public types -----
type Integer_32 is new Integer;
```

Figure 91: Generated code of a typed class

Definition of a subtype

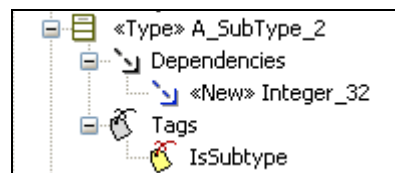


Figure 92: Representation of a subtype

```
--Public types -----
subtype A_SubType is Integer;
```

Figure 93: Generated code of a subtype

Some other stereotypes are derived from “Type”

- RangeType : allow defining range type
- ArrayType : allow defining array type
- VariantRecordType : allow defining variant record type

4.7.1.4 Range type

A range type is defined by a class which has the stereotype “RangeType”.

The range is defined by two different ways.

- First it can be defined in a free text box in the tag “rangeDefinition”.
- It can also be defined with dependencies to a constant of the model. The dependencies have the stereotype “highRangeValue” or “lowRangeValue”.

Basic use cases :

Case I

- “New” dependency to a predefined type
- rangeDefinition set to “1..10”



Figure 94: Representation of a range type

```
--Public types -----
type A_Range_Type is new Integer range 1..10;
```

Figure 95: Generated code of a range type

Case II

- “New” dependency to a typed class
- “HighRangeValue” dependency to a constant of the model
- “IsSubtype” tag set to true
- “rangeDefinition” tag set to “1”

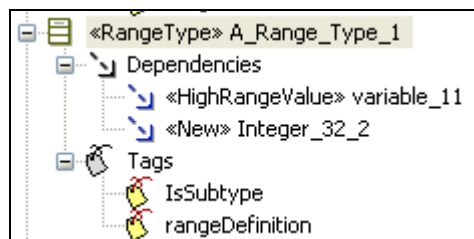


Figure 96: Representation of a range type with dependency to a constant

```
--Public types -----
subtype A_Range_Type_1 is Default.Integer_32_2 range 1 .. Default.variable_11;
```

Figure 97: Generated code of a range type

4.7.2.4 *Array type*

An array type is defined by a class which has the stereotype “ArrayType”.

The size of the array is defined in a free text box of the tag “Size”

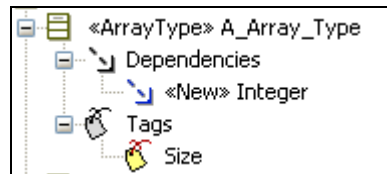


Figure 98: Representation of an array type

```

--Public types -----
type A_Array_Type is array (1..10) of Integer;

```

Figure 99: Generated code of an array type

4.7.3.4 *Variant record type*

A variant record type is defined by a class which has the stereotype “VariantRecordType”.

This class has several elements which describe the structure of this record.

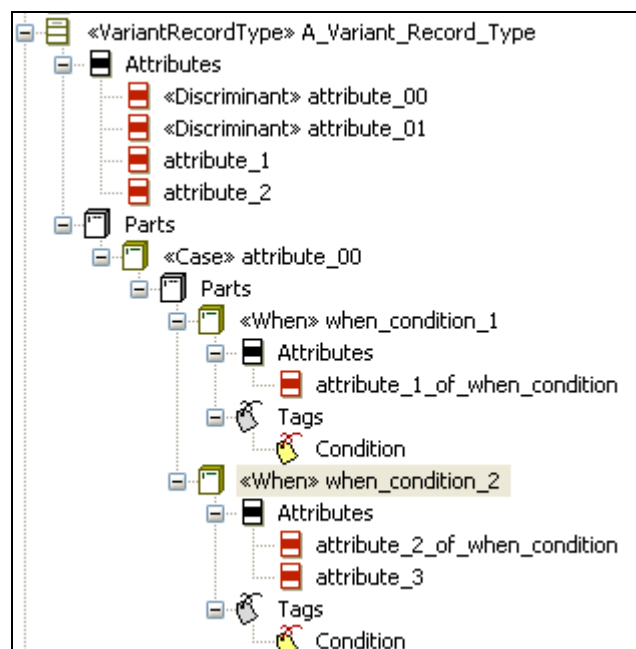


Figure 100: Representation of a variant record type

The class has some attributes which represent the attributes of the record type.

Some attributes have the stereotype discriminant. Those are discriminant attributes of the record type.

The class has a part with stereotype “Case”. It represents the variant part of the record. The name of this part must be the name of the discriminant attribute used in the variant part.

The “Case” part has as many “When” parts as “When” cases in the Ada variant part. Those “When” parts have attributes, and a tag “Condition”, which defines the value of the “when” condition.

```
--Public types -----
type A_Variant_Record_Type (
  attribute_00 : Default.A_Enum;
  attribute_01 : Default.A_Range_Type
) is record

  -- Fields --
  attribute_1 : Integer;
  attribute_2 : Integer;

  case attribute_00 is
    when AA => attribute_1_of_when_condition : Integer;

    when BB..CC => attribute_2_of_when_condition : Integer;
                  attribute_3 : Integer;

  end case;

end record;
```

Figure 101: Generated code of a variant record type

4.8. Template Classes and their instantiation

4.8.1. template classes

Creating a template class in Rhapsody results in the generation of a generic Ada package. The arguments become the generic parameters.

In this example, *arg1* is an Integer argument and *arg2* is a Boolean argument.

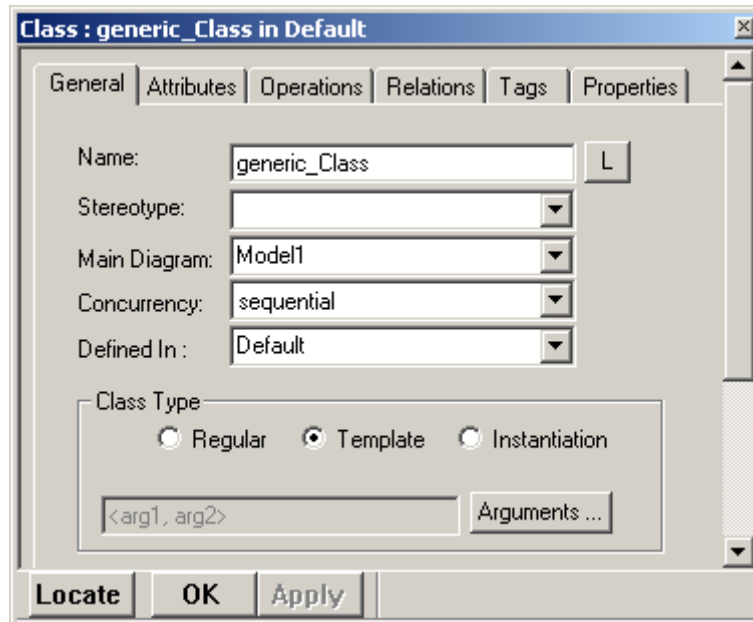


Figure 102: Definition of a template class.

```
--++ class generic_Class
generic
  arg1 : Integer;
  arg2 : Boolean;
package generic_Class is

  type generic_Class_t;
  type generic_Class_acc_t is access all generic_Class_t;

  type generic_Class_t is tagged null record;

private

end generic_Class;
```

Figure 103: Package specification for a generic package.

4.8.2. template instantiations

An instantiation of this generic package is created by using an instantiation class in Rhapsody.

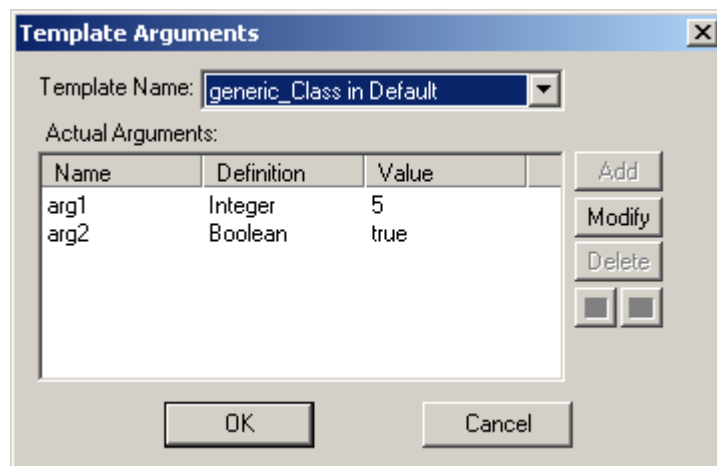


Figure 104: An instantiation of a template class.

The generated result is an instantiation of the generic package using the supplied arguments.

```
With generic_Class;

--++ class generic_Instantiation
package generic_Instantiation is new generic_Class(arg1 => 5, arg2 => true);
```

Figure 105: The generated Ada package for a generic instantiation.

4.8.3. template inheritance

Note that it is possible to have a template class inherit from another template class.

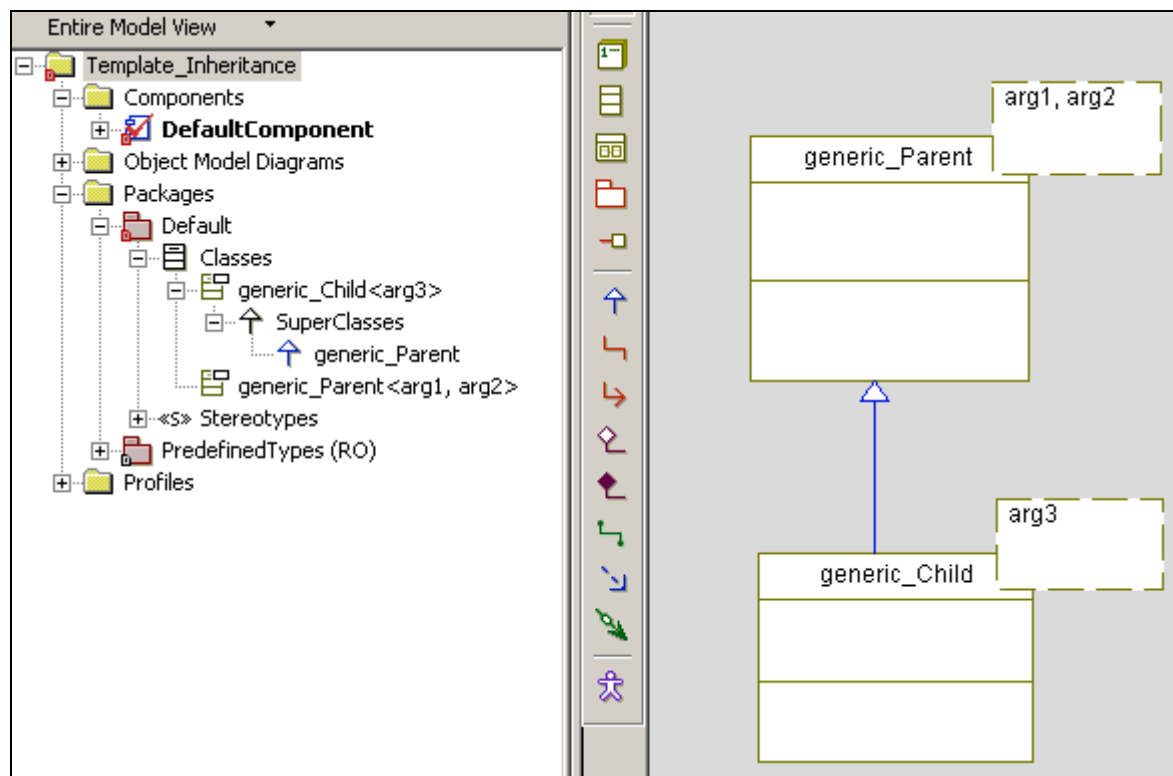


Figure 106 Inheritance between template classes

```

With generic_Parent;

--++ class generic_Child
generic
  with package generic_Parent_Instantiation is new generic_Parent(<>);
  arg3 : Character;
package generic_Child is

  type generic_Child_t;
  type generic_Child_acc_t is access all generic_Child_t;

  type generic_Child_t is new generic_Parent_Instantiation.generic_Parent_t with null record;
private
end generic_Child;

```

Figure 107 generated code for a template class derived from another template class

4.8.4. template instantiation inheritance

In order to fully benefit from the facilities offered by template inheritance, an efficient way to instantiate the whole class hierarchy is needed.

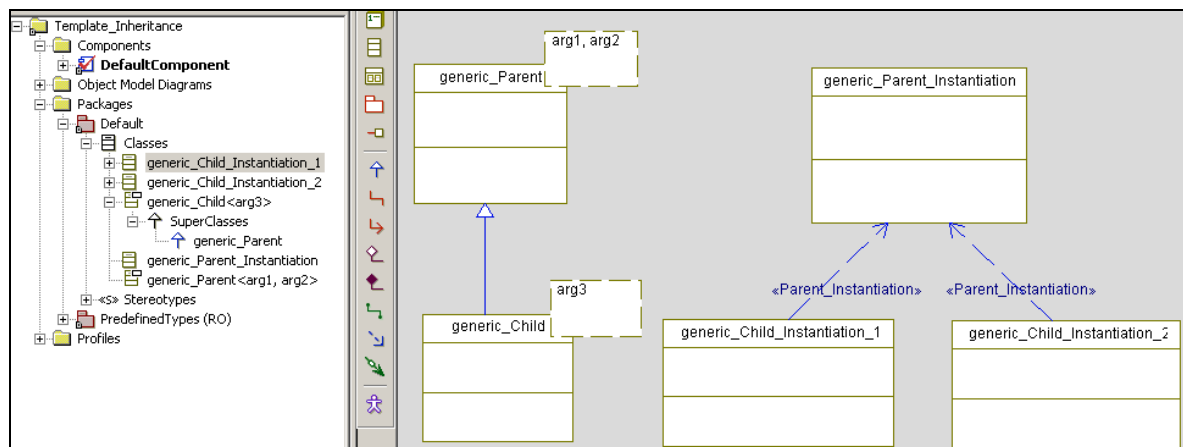


Figure 108 Modeling instantiation of a template inheritance hierarchy

Note that to specify that a derived class instantiation depends on a parent class instantiation, we use a <<Parent_Instantiation>> dependency from the derived class instantiation to the base class instantiation. This approach allows for reusing of the same parent class instantiation by several derived class instantiations

```

With generic_Parent;

--++ class generic_Parent_Instantiation
package generic_Parent_Instantiation is new generic_Parent(arg1 => 5, arg2 => true);

```

Figure 109 generated code for a base template instantiation class

```

With generic_Child;
With generic_Parent_Instantiation;

--++ class generic_Child_Instantiation_1
package generic_Child_Instantiation_1 is new generic_Child(
  generic_Parent_Instantiation => generic_Parent_Instantiation,
  arg3 => 'a'
);

```

Figure 110 generated code for a derived template instantiation class

```

With generic_Child;
With generic_Parent_Instantiation;

--++ class generic_Child_Instantiation_2
package generic_Child_Instantiation_2 is new generic_Child(
    generic_Parent_Instantiation => generic_Parent_Instantiation,
    arg3 => 'b'
);

```

Figure 111 generate code for another derived template instantiation class

Note that if the derived template class is an (Ada) child package of the base class, the generated code will slightly differ to accommodate the special visibility that the child has upon its parent

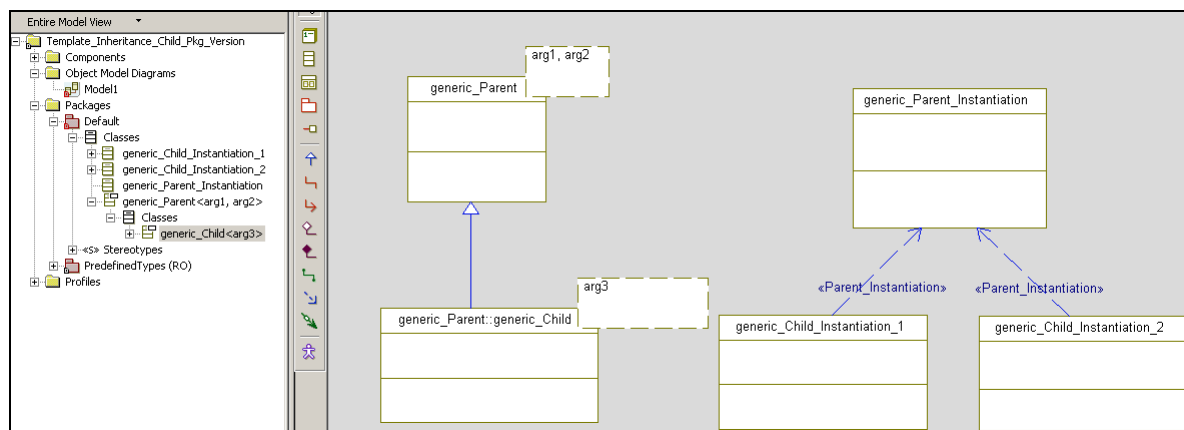


Figure 112 Modeling template inheritance hierarchy across (Ada) children packages

```

--++ class generic_Parent::generic_Child
generic
    arg3 : Character;
package generic_Parent.generic_Child is

    type generic_Child_t;
    type generic_Child_acc_t is access all generic_Child_t;

    type generic_Child_t is new generic_Parent.generic_Parent_t with null record;
private

end generic_Parent.generic_Child;

```

Figure 113 Generated code for a derived template class that is a child package of its base class

```

With generic_Parent.generic_Child;
With generic_Parent_Instantiation;

--++ class generic_Child_Instantiation_1
package generic_Child_Instantiation_1 is new generic_Parent_Instantiation.generic_Child(
    arg3 => 'a'
);

```

Figure 114 Generated code for the instantiation of a derived template class that is a child package of its base class

4.9. Concurrent types

4.9.1. Tasks

Ada tasks are represented in Rhapsody by a class stereotyped as `<<AdaTask>>` or `<<AdaTaskType>>`. The result is the creation of an Ada package containing a task type. The `<<AdaTask>>` stereotype should be used for singleton tasks.

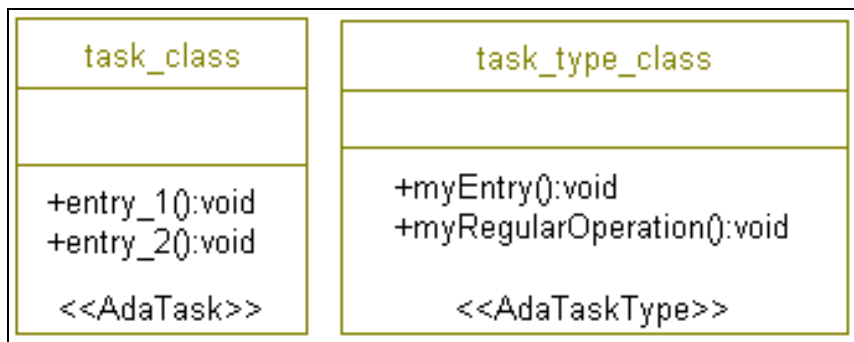


Figure 115: Ada tasks in Rhapsody.

For `<<AdaTask>>` and `<<AdaTaskType>>` classes, the `Ada_CG.Class.Visibility` property has to be set to "Private"

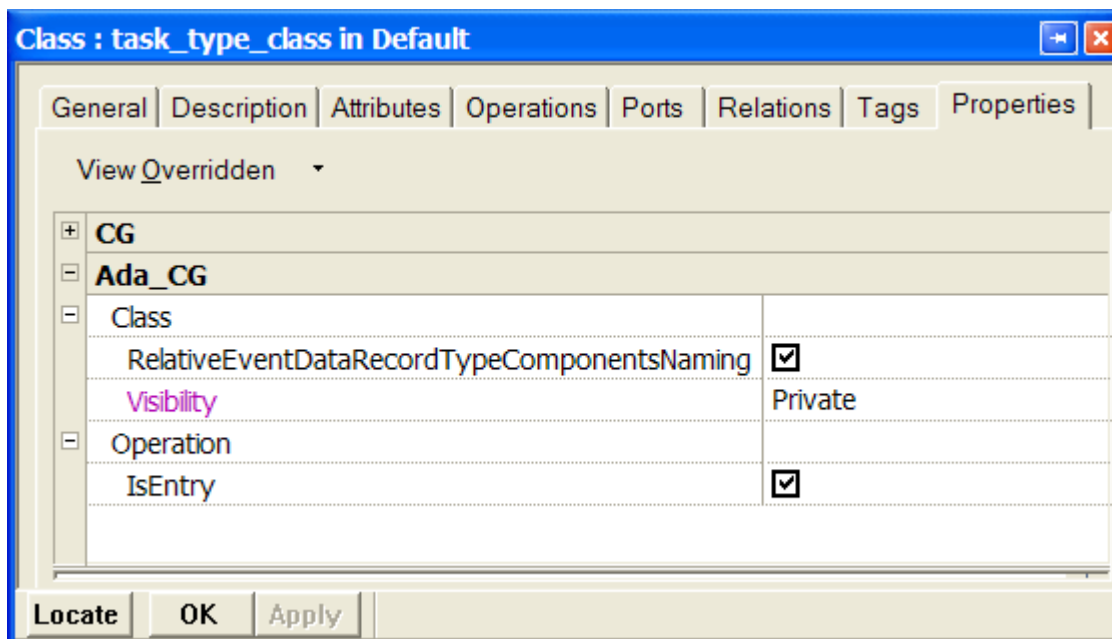


Figure 116: Setting the record type visibility to "Private" for an `<<AdaTaskType>>` class

By default, all the operations in the class represent the entries of the task, and can have either a `<<HSER>>` or `<<LSER>>` stereotype to indicate highly synchronous or loosely synchronous execution requests respectively. In the given example, *myEntry* is HSER while *entry_1* and *entry_2* are LSER. The implementation of the operations is used as the task entry bodies.

For operations such as *myRegularOperation* which do not represent an entry, the `Ada_CG.Operation.isEntry` property has to be overridden to False (it defaults to True on `<<AdaTask>>` and `<<AdaTaskType>>` classes). This will generate a regular operation instead of a task entry for this operation.

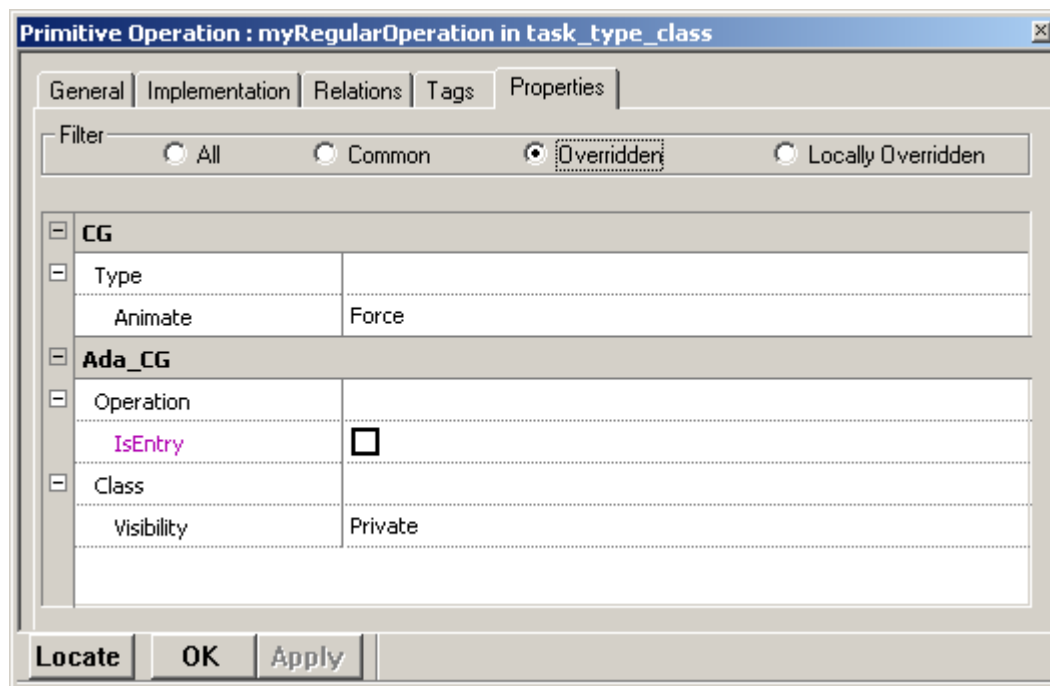


Figure 117: Setting an operation on a <<AdaTaskType>> class to generate as a regular operation.

```
--++ class task_class
package task_class is

    type task_class_t is tagged private;
    type task_class_acc_t is access all task_class_t;

    --Public Functions/Procedures section -----
    procedure entry_1 (this : in out task_class_t);

    procedure entry_2 (this : in out task_class_t);

private

    --task type declaration
    task task_class_task is
        entry entry_1 (this : in out task_class_t);
        entry entry_2 (this : in out task_class_t);
    end task_class_task;

    type task_class_t is tagged null record;

end task_class;
```

Figure 118: Ada task specification.

```
--++ class task class
package body task_class is

    --Functions/Procedures section -----
    procedure entry_1 (this : in out task_class_t) is
    begin
        task_class_task.entry_1 (this);
    end entry_1;

    procedure entry_2 (this : in out task_class_t) is
    begin
        task_class_task.entry_2 (this);
    end entry_2;

    --Tasking Implementation -----
    task body task_class_task is
    begin
        loop
            select
                accept entry_1 (this : in out task_class_t);
                    --implementation of entry_1
            or
                accept entry_2 (this : in out task_class_t);
                    --implementation of entry_2
            end select;
        end loop;
    end task_class_task;
end task_class;
```

Figure 119: Ada task body.

```
--++ class task_type_class
package task_type_class is

    type task_type_class_t is tagged private;
    type task_type_class_acc_t is access all task_type_class_t;

    --Public Functions/Procedures section -----
    procedure myEntry (this : in out task_type_class_t);

    --++ operation myRegularOperation()
    procedure myRegularOperation (this : in out task_type_class_t);

    procedure Initialize(this : in out task_type_class_t);

    procedure Finalize(this : in out task_type_class_t);

private

    --task type declaration
    task type task_type_class_task is
        entry myEntry (this : in out task_type_class_t);
    end task_type_class_task;

    type task_type_class_task_acc is access task_type_class_task;

    type task_type_class_t is tagged

    record
        my_task_type_class_task : task_type_class_task_acc;

    end record;

end task_type_class;
```

Figure 120: Ada task type specification.

```

With UNCHECKED_DEALLOCATION;

--++ class task type class
package body task_type_class is

    --Functions/Procedures section -----
    procedure myEntry (this : in out task_type_class_t) is
    begin
        this.my_task_type_class_task.myEntry (this);
    end myEntry;

    procedure myRegularOperation (this : in out task_type_class_t) is
        --+[ operation myRegularOperation().Variables
        --+]
    begin
        null;
        --+[ operation myRegularOperation()

        --+]
    end myRegularOperation;

    --Tasking Implementation -----
    task body task_type_class_task is
    begin
        loop
            select
                accept myEntry (this : in out task_type_class_t) do
                    --implementation of myEntry
                    null;
                end myEntry;
            end select;
        end loop;
    end task_type_class_task;

    procedure Initialize(this : in out task_type_class_t) is
    begin
        this.my_task_type_class_task := new task_type_class_task;
    end Initialize;

    procedure Finalize(this : in out task_type_class_t) is
        procedure FREE is new UNCHECKED_DEALLOCATION(
            task_type_class_task,
            task_type_class_task_acc
        );
    begin
        FREE(this.my_task_type_class_task);
    end Finalize;

end task_type_class;

```

Figure 121: Ada task type body.

The Ada task type generates the constructor and destructor as well, which create the task instance and destroy it.

It is also possible to define timed or conditional entries on a task. In the next example, we define a timed entry by doing the following:

1. Set the Ada_CG.TaskDefaultScheme to “Timed” on task_with_default_entry class.
2. Set the Ada_CG.TaskDefaultSchemeDelayStatement with a valid delay statement
3. Apply the <<TaskDefaultAction>> to timeOutAction

Note that to define conditional entries, you must set the property to “Conditional” and then apply the <<TaskDefaultAction>> to the default entry.

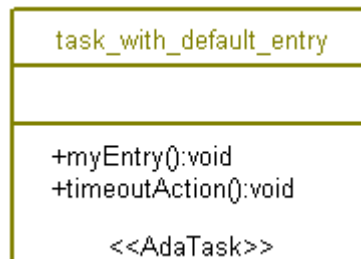


Figure 122 Ada task with default entry

```

--++ class task_with_default_entry
package task_with_default_entry is

  type task_with_default_entry_t;
  type task_with_default_entry_acc_t is access all task_with_default_entry_t;

  --task type declaration
  task task_with_default_entry_task is
    entry myEntry (this : in out task_with_default_entry_t);
  end task_with_default_entry_task;

  type task_with_default_entry_t is tagged null record;

  --Public Functions/Procedures section -----
  procedure myEntry (this : in out task_with_default_entry_t);

private
end task_with_default_entry;
  
```

Figure 123 Specification of Ada Task with default entry

```

--++ class task_with_default_entry
package body task_with_default_entry is

    --Functions/Procedures section -----
    procedure myEntry (this : in out task_with_default_entry_t) is
    begin
        task_with_default_entry_task.myEntry (this);
    end myEntry;

    --Tasking Implementation -----
    task body task_with_default_entry_task is
    begin
        loop
            select
                accept myEntry (this : in out task_with_default_entry_t) do
                    -- regular entry body
                end myEntry;
            or
                delay 10.0;
                -- default behavior starts here
            end select;
        end loop;
    end task_with_default_entry_task;
end task_with_default_entry;

```

Figure 124 Implementation of Ada Task with default entry

4.9.2. Protected Objects

Protected objects are represented in Rhapsody by a class stereotyped as <<AdaProtectedObject>> or <<AdaProtectedType>>. The result is the creation of an Ada package containing a protected type. The <<AdaProtectedObject>> stereotype should be used for singleton tasks.

Protected_Object_Class	Protected_Type_Class
	+attribute_0 : Integer
+entry_default_guard():void +regular_operation():void +regular_function():Integer <<AdaProtectedObject>>	+entry_true_or_false():void <<AdaProtectedType>>

Figure 125: Protected objects in Rhapsody.

For <<AdaProtectedObject>> and <<AdaProtectedType>> classes, the Ada_CG.Class.Visibility property has to be set to "Private"

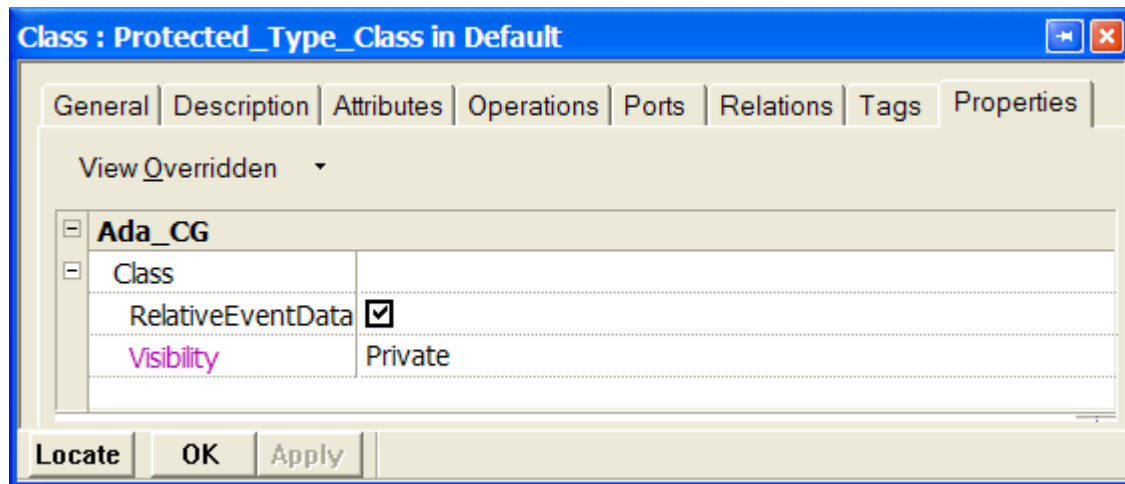


Figure 126: Setting the record type visibility to “Private” for an <<AdaProtectedType>> class

By default, all operations in the class do not represent entries. In order to generate a protected object entry, one has to apply the <<entry>> stereotype to an operation.

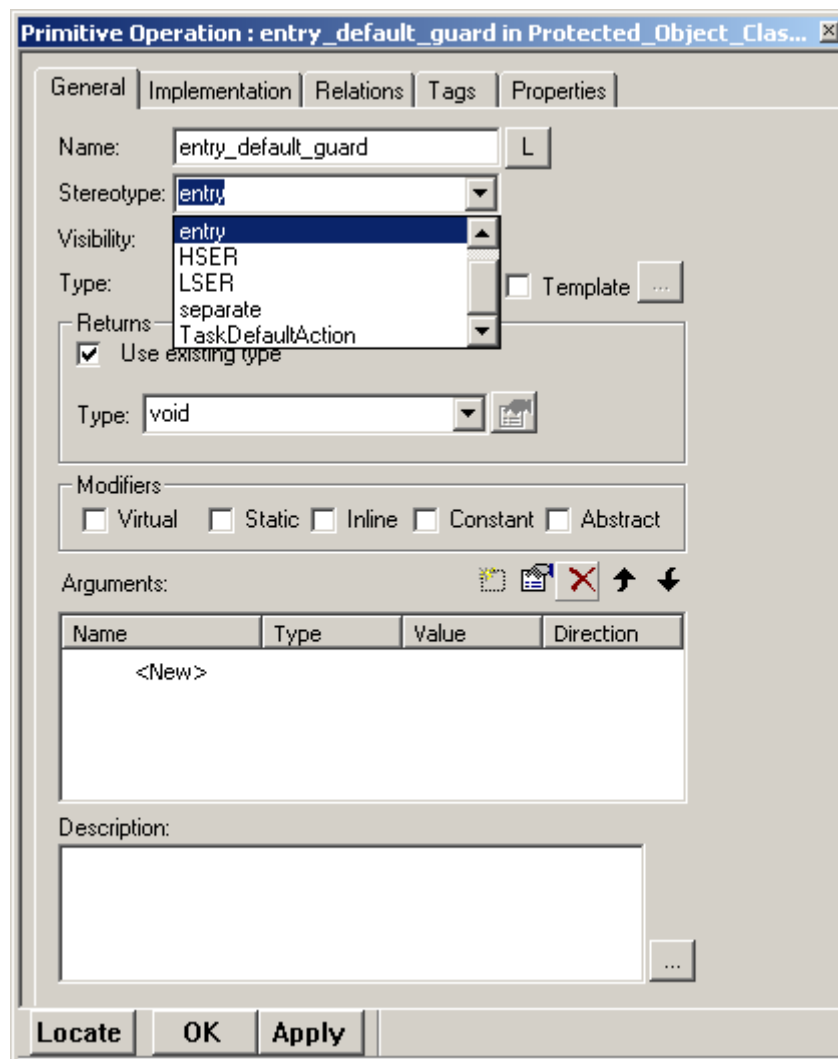


Figure 127: Applying the <<entry>> stereotype to a protected object operation

By default, the guard for a protected entry will be set to true, however one can define its own guard by setting the “Ada.CG.Operation.EntryCondition” property to the appropriate boolean expression.

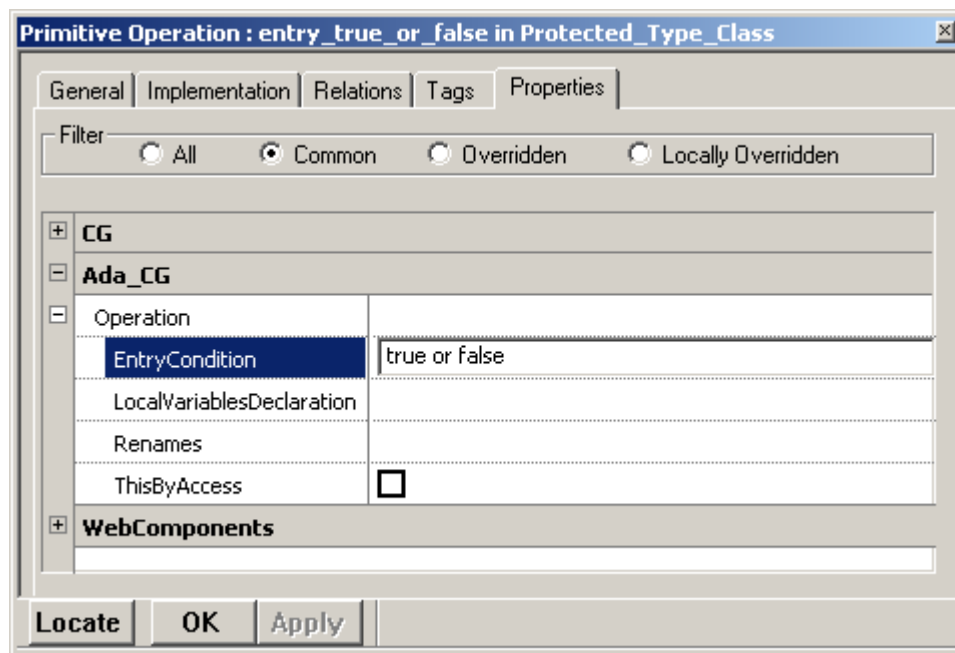


Figure 128: Setting the guard for a protected object/type entry

```

--++ class Protected_Object_Class
package Protected_Object_Class is

    type Protected_Object_Class_t is tagged limited private;
    type Protected_Object_Class_acc_t is access all Protected_Object_Class_t;

    --Public Functions/Procedures section -----
    --++ operation entry_default_guard()
    procedure entry_default_guard (this : in out Protected_Object_Class_t);

    --++ operation regular_function()
    function regular_function (this : in Protected_Object_Class_t) return Integer;

    --++ operation regular_operation()
    procedure regular_operation (this : in out Protected_Object_Class_t);

private

    --protected type declaration
    protected Protected_Object_Class_protected is

        entry entry_default_guard (this : in out Protected_Object_Class_t);
        procedure regular_operation (this : in out Protected_Object_Class_t);
        function regular_function (this : in Protected_Object_Class_t) return Integer;

    private

    end Protected_Object_Class_protected;

    type Protected_Object_Class_t is tagged limited null record;

end Protected_Object_Class;

```

Figure 129: Protected object specification.

```

--++ class Protected Object Class
package body Protected_Object_Class is

  --Functions/Procedures section -----
  procedure entry_default_guard (this : in out Protected_Object_Class_t) is
  begin
    Protected_Object_Class_protected.entry_default_guard (this);
  end entry_default_guard;

  procedure regular_operation (this : in out Protected_Object_Class_t) is
  begin
    Protected_Object_Class_protected.regular_operation (this);
  end regular_operation;

  function regular_function (this : in Protected_Object_Class_t) return Integer is
  begin
    return Protected_Object_Class_protected.regular_function (this);
  end regular_function;

  --Protected Object/Type Implementation
  protected body Protected_Object_Class_protected is
    entry entry_default_guard (this : in out Protected_Object_Class_t) when true is
    begin
      null;
      --+[ operation entry_default_guard()

      --+]
    end entry_default_guard;

    procedure regular_operation (this : in out Protected_Object_Class_t) is
      --+[ operation regular_operation().Variables

      --+]
    begin
      null;
      --+[ operation regular_operation()

      --+]
    end regular_operation;

    function regular_function (this : in Protected_Object_Class_t) return Integer is
      --+[ operation regular_function().Variables

      --+]
    begin
      return 0;
      --+[ operation regular_function()

      --+]
    end regular_function;

  end Protected_Object_Class_protected;
end Protected_Object_Class;

```

Figure 130: Protected object body.

```

--++ class Protected Type Class
package Protected_Type_Class is

    type Protected_Type_Class_t is tagged limited private;
    type Protected_Type_Class_acc_t is access all Protected_Type_Class_t;

    -- Public Variables/Constants -----
    static_attribute : Integer;    --++ attribute static_attribute

    --Public Functions/Procedures section -----
    --++ operation entry_function()
    function entry_function (this : in Protected_Type_Class_t) return Integer;

    --++ operation entry_true_or_false()
    procedure entry_true_or_false (this : in out Protected_Type_Class_t);

    --Public Fields/Variables accessors -----
    function get_attribute_0 (this : in Protected_Type_Class_t) return Integer;
    pragma inline (get_attribute_0);

    procedure set_attribute_0 (this : in out Protected_Type_Class_t; value : in Integer);
    pragma inline (set_attribute_0);

    function get_static_attribute return Integer;
    pragma inline (get_static_attribute);

    procedure set_static_attribute (value : in Integer);
    pragma inline (set_static_attribute);

    procedure Initialize(this : in out Protected_Type_Class_t);

    procedure Finalize(this : in out Protected_Type_Class_t);

private

    --protected type declaration
    protected type Protected_Type_Class_protected is

        entry entry_true_or_false (this : in out Protected_Type_Class_t);
        function entry_function (this : in Protected_Type_Class_t) return Integer;
        function get_attribute_0 (this : in Protected_Type_Class_t) return Integer;
        procedure set_attribute_0 (this : in out Protected_Type_Class_t; value : in Integer);

    private

        -- Fields --
        attribute_0 : Integer;    --++ attribute attribute_0

    end Protected_Type_Class_protected;

    type Protected_Type_Class_protected_acc is access Protected_Type_Class_protected;

    type Protected_Type_Class_t is tagged limited

    record
        my_Protected_Type_Class_protected : Protected_Type_Class_protected_acc;

    end record;

end Protected_Type_Class;

```

Figure 131: Protected type specification.

```

With UNCHECKED_DEALLOCATION;

--++ class Protected_Type_Class
package body Protected_Type_Class is

  --Functions/Procedures section -----
  procedure entry_true_or_false (this : in out Protected_Type_Class_t) is
  begin
    this.my_Protected_Type_Class_protected.entry_true_or_false (this);
  end entry_true_or_false;

  --Fields/Variables accessors -----
  function get_attribute_0(this : in Protected_Type_Class_t) return Integer is
  begin
    return this.my_Protected_Type_Class_protected.get_attribute_0(this);
  end get_attribute_0;

  procedure set_attribute_0 (this : in out Protected_Type_Class_t; value : in Integer) is
  begin
    this.my_Protected_Type_Class_protected.set_attribute_0 (this, value);
  end set_attribute_0;

  --Protected Object/Type Implementation
  protected body Protected_Type_Class_protected is
    entry entry_true_or_false (this : in out Protected_Type_Class_t) when true or false is
    begin
      null;
      --+[ operation entry_true_or_false()
      --+
      --+]
    end entry_true_or_false;

    function get_attribute_0(this : in Protected_Type_Class_t) return Integer is
    begin
      return attribute_0;
    end get_attribute_0;

    procedure set_attribute_0 (this : in out Protected_Type_Class_t; value : in Integer) is
    begin
      attribute_0 := value;
    end set_attribute_0;

  end Protected_Type_Class_protected;

  procedure Initialize(this : in out Protected_Type_Class_t) is
  begin
    this.my_Protected_Type_Class_protected := new Protected_Type_Class_protected;
  end Initialize;

  procedure Finalize(this : in out Protected_Type_Class_t) is
    procedure FREE is new UNCHECKED_DEALLOCATION(
      Protected_Type_Class_protected,
      Protected_Type_Class_protected_acc
    );
  begin
    FREE(this.my_Protected_Type_Class_protected);
  end Finalize;

end Protected_Type_Class;

```

Figure 132: Protected type body.

The protected type generates the constructor and destructor as well, which create the task instance and destroy it.

4.10. Entrypoints

An entrypoint can be created in Rhapsody to represent the starting point of the Ada program. This is done by stereotyping a class as `<<Entrypoint>>`.

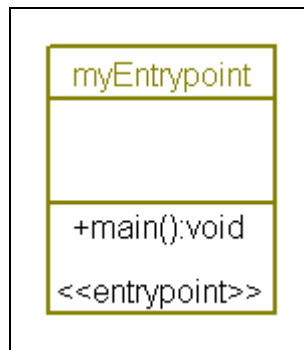


Figure 133: An entrypoint in Rhapsody.

In addition, define an operation on the class, and enter the implementation of the operation to complete the entrypoint. The result is a single package body file generated in `myEntrypoint.adb`.

```
procedure myEntrypoint is
  --+[ operation main().Variables
  --+]
begin
  --+[ operation main()
  --entrypoint implementation
  --+]
end myEntrypoint;
```

Figure 134: The entrypoint definition.

4.11. Singleton Classes

Singleton classes represent classes that have only one instance. This is represented in Rhapsody by stereotyping the class `<<Singleton>>`. A singleton class creates a private variable to contain the singleton instance, and all non-static operations access this instance instead of passing in a *'this'* parameter.

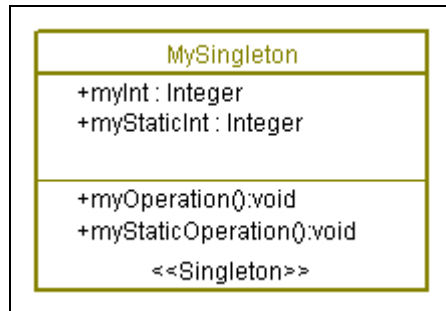


Figure 135: A singleton class in Rhapsody.

4.11.1. Ada 95

When the singleton class is generated using the Ada 95 rules, the non-static attributes are held in a record in the same manner as a normal class.

```

--++ class My_Singleton
package My_Singleton is

  type My_Singleton_t;
  type My_Singleton_acc_t is access all My_Singleton_t;

  type My_Singleton_t is tagged

  record

    -- Fields --
    my_Int : Integer;    --++ attribute my_Int

  end record;

  -- Public Variables/Constants -----
  my_Static_Int : Integer;  --++ attribute my_Static_Int

  --Public Functions/Procedures section -----
  --++ operation my_Operation()
  procedure my_Operation (this : in out My_Singleton_t);

  --++ operation my_Static_Operation()
  procedure my_Static_Operation;

  --Public Fields/Variables accessors -----
  function get_my_Int (this : in My_Singleton_t) return Integer;
    pragma inline (get_my_Int);

  procedure set_my_Int (this : in out My_Singleton_t; value : in Integer);
    pragma inline (set_my_Int);

  function get_my_Static_Int return Integer;
    pragma inline (get_my_Static_Int);

  procedure set_my_Static_Int (value : in Integer);
    pragma inline (set_my_Static_Int);

private

end My_Singleton;

```

Figure 136: The package specification for a singleton class in Ada 95.

```

--++ class My_Singleton
package body My_Singleton is

    --Functions/Procedures section -----
    procedure my_Operation (this : in out My_Singleton_t) is
        --+[ operation my_Operation().Variables

        --+]
    begin
        null;
        --+[ operation my_Operation()

        --+]
    end my_Operation;

    procedure my_Static_Operation is
        --+[ operation my_Static_Operation().Variables

        --+]
    begin
        null;
        --+[ operation my_Static_Operation()

        --+]
    end my_Static_Operation;

    --Fields/Variables accessors -----
    function get_my_Int(this : in My_Singleton_t) return Integer is
    begin
        return this.my_Int;
    end get_my_Int;

    procedure set_my_Int (this : in out My_Singleton_t; value : in Integer) is
    begin
        this.my_Int := value;
    end set_my_Int;

    function get_my_Static_Int return Integer is
    begin
        return my_Static_Int;
    end get_my_Static_Int;

    procedure set_my_Static_Int (value : in Integer) is
    begin
        my_Static_Int := value;
    end set_my_Static_Int;

end My_Singleton;

```

Figure 137: The package body for a singleton class in Ada 95.

4.11.2. Ada 83

Changing the “DefaultComponent” component to be an Ada 83 package changes the generation of the singleton class so that the rules for Ada 83 are followed. In this case, all attributes are considered static attributes and a record type is not created for the class nor is a variable for the singleton instance created in the package body.

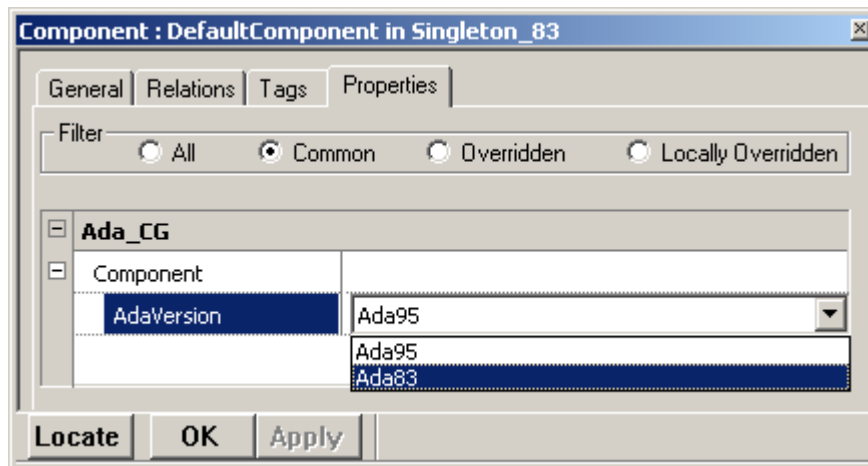


Figure 138: Changing the component to generate Ada 83 code.

```
--++ class My_Singleton
package My_Singleton is

    my_Int : Integer; --++ attribute my_Int

    -- Public Variables/Constants -----
    my_Static_Int : Integer; --++ attribute my_Static_Int

    --Public Functions/Procedures section -----
    --++ operation my_Operation()
    procedure my_Operation;

    --++ operation my_Static_Operation()
    procedure my_Static_Operation;

    --Public Fields/Variables accessors -----
    function get_my_Int return Integer;
    pragma inline (get_my_Int);

    procedure set_my_Int (value : in Integer);
    pragma inline (set_my_Int);

    function get_my_Static_Int return Integer;
    pragma inline (get_my_Static_Int);

    procedure set_my_Static_Int (value : in Integer);
    pragma inline (set_my_Static_Int);

private
end My_Singleton;
```

Figure 139: The package specification for a singleton class in Ada 83.

```

--++ class My_Singleton
package body My_Singleton is

    --Functions/Procedures section -----
    procedure my_Operation is
        --+[ operation my_Operation().Variables
        --+]
    begin
        null;
        --+[ operation my_Operation()

        --+]
    end my_Operation;

    procedure my_Static_Operation is
        --+[ operation my_Static_Operation().Variables
        --+]
    begin
        null;
        --+[ operation my_Static_Operation()

        --+]
    end my_Static_Operation;

    --Fields/Variables accessors -----
    function get_my_Int return Integer is
    begin
        return my_Int;
    end get_my_Int;

    procedure set_my_Int (value : in Integer) is
    begin
        my_Int := value;
    end set_my_Int;

    function get_my_Static_Int return Integer is
    begin
        return my_Static_Int;
    end get_my_Static_Int;

    procedure set_my_Static_Int (value : in Integer) is
    begin
        my_Static_Int := value;
    end set_my_Static_Int;

end My_Singleton;

```

Figure 140: The package body for a singleton class in Ada 83.

4.12. Unidirectional Relations

There are three different options for the implementation of relations. Each implementation creates an Ada record field and accessor(s) and mutator(s) methods and possibly some new types and inner packages to support the implementation. The name of the Ada record field is the name of the role for the relation.

Default implementation: The default implementation uses an access type for the target object. The access type is held in an Ada record field. In addition, a “With” statement is added for the target package so that the access type is visible.

Fixed implementation: The fixed implementation uses a direct reference to the target object. The target object type is held in an Ada record field. As with the “Default” implementation, a “With” statement is added for the target package so that the target type is visible.

Scalar implementation: The scalar implementation is better described as the index implementation. Instead of holding a direct reference to either the target type or an access to the target type, a numerical index is used instead. The intent is that there will be a container object in the system that will hold the instances of the target class, and that the index is used to retrieve the correct instance from the collection. In this case, a “With” statement is not created for the target package because only an index is stored in the class, and not a reference to the target object itself. Instead, a new type is created to represent the valid range for the index. Bi-directional relations are not supported at this time, nor are unbounded multiplicities.

4.12.1. Multiplicity = 1

When the multiplicity = 1, the class will have a reference to only one instance of the target class. The following example demonstrates the generated code for each of the implementation types. Note that no accessors are generated for the “Fixed” implementation.

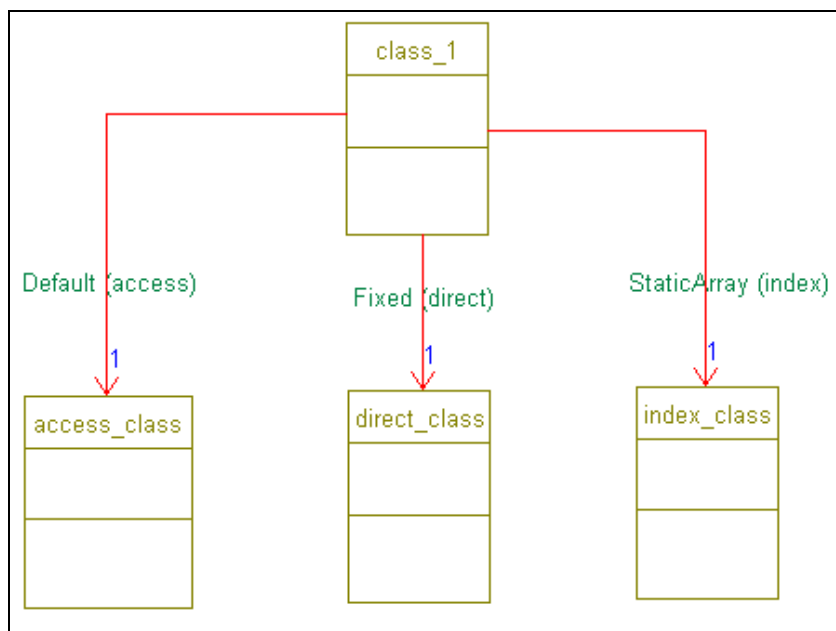


Figure 141: Class relations with multiplicity = 1.

4.12.2. Multiplicity > 1, general notes

When the multiplicity is greater than 1, the same basic concepts are followed for each implementation choice, except some data structures are created to hold the instances. In addition, new types are defined to represent the valid indices into such structures, and iterator subpackages are declared for every relation..

Unbounded relations and qualified relations (both bounded and unbounded) use data structures that rely on the Booch components.

Generated Method	Relation type			
	Bounded	Unbounded	Bounded Qualified	Unbounded Qualified
Get_At_Pos (Procedure)	Y	Y		
Get_At_Pos (Function)	Y	Y		
Contains	Y	Y		
Get_Count	Y	Y	Y	Y
Set_At_Pos	Y	Y		
Add_At_Pos	N	Y		
Remove	N	Y		
Remove_At_Pos	N	Y		
Get(Key) (Procedure)			Y	Y
Get(Key) (Function)				
Contains(Key)			Y	Y
Set(Key)			Y	Y
Remove(Key)			Y	Y

Table 2 Nary relations methods matrix

Method Name	Description
Initialize	Creates the iterator
Get_Next	Gets next element
To_Value	Get value associated to current iterator position
Is_Last	Return true if there's no more elements to iterate over.

Table 3 Nary relations Iterator package method description

4.12.3. Details on the Booch components

IBM® Rational® Rhapsody® Developer for Ada does not install Booch Components files. If needed, user must do it manually by following the procedure. See §2.3 Booch components.

Either the original Ada 83 version of the components can be used, or the Ada 95 version. The choice is made at the component level with the Ada_CG.Component.UseBoochComponents property.

Although only part of it is used. Here is the list of packages that may be with'ed by the generated code when using Ada 83:

1. semaphore

2. storage_manager_concurrent
3. list_single_unbounded_controlled
4. list_utilities_single
5. list_search
6. map_simple_noncached_concurrent_bounded_managed_noniterator
7. map_simple_noncached_concurrent_unbounded_managed_noniterator

	<i>Bounded</i>	<i>Unbounded</i>
<i>Unqualified relations</i>	None	1,2,3,4,5
<i>Qualified relations</i>	1,6	1,2,7

Table 4 Booch 83 Components Package Dependency Matrix

Here is the list of packages that may be with'ed by the generated code when using Ada 95:

1. BC.Support.Standard_Storage
2. BC.Containers.Collections.Unbounded
3. BC.Containers.Maps.Unbounded
4. BC.Containers.Maps.Bounded

	<i>Bounded</i>	<i>Unbounded</i>
<i>Unqualified relations</i>	None	1,2
<i>Qualified relations</i>	4	1,3

Table 5 Booch 95 Components Package Dependency Matrix

4.12.4. Multiplicity > 1, bounded

An array is created to hold the instances

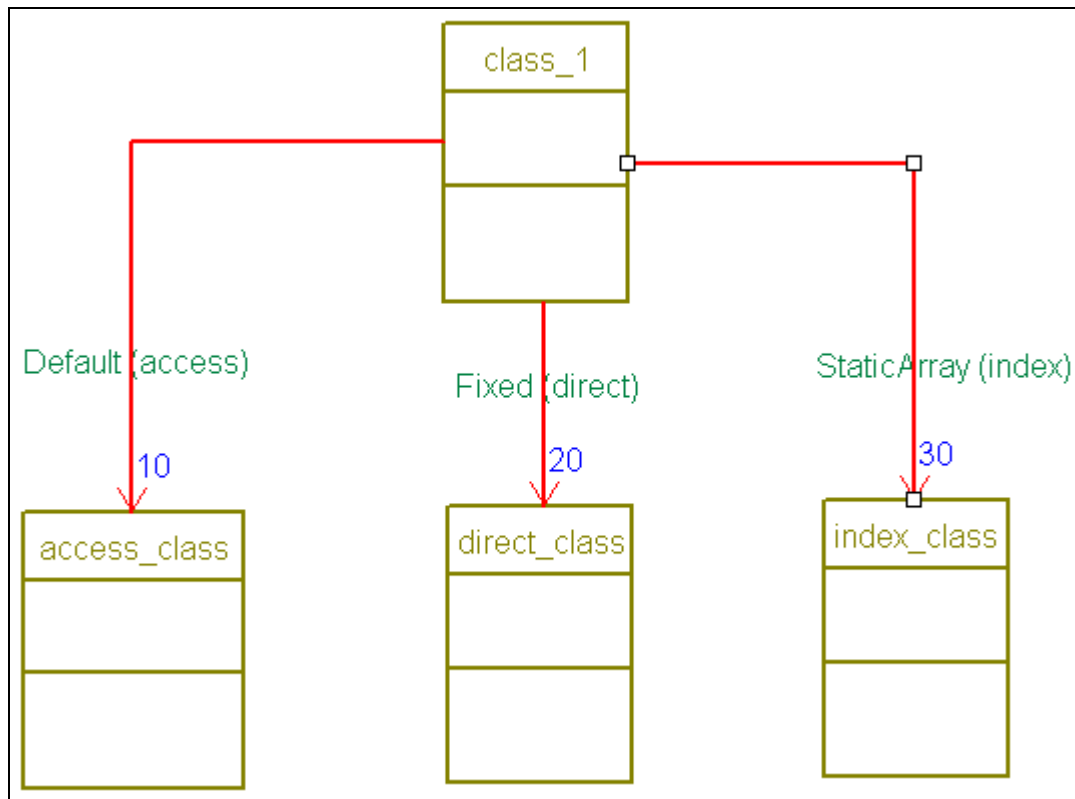


Figure 142: Class relations with multiplicity > 1, bounded.

4.12.5. Multiplicity > 1, unbounded

Unbounded relations are no longer represented as arrays of 100 elements, but as data structures relying on dynamic memory allocation.

Important note: Any legacy code using unbounded relations relying on the fact that the underlying implementation is an array of 100 elements **MUST** be updated, as it will either not compile or potentially lead to run-time errors.

4.12.6. Multiplicity > 1, qualified relations

Qualified relations are represented with maps. The unbounded form relies on dynamic memory allocation.

A qualified relation is a key based association. This means that if there is an association from A to B, where the qualifier is an attribute B.id of type Integer, the relation is key based (in this case, Integer is the key type).

Only one element can be bound for each value of the qualifier domain.

Adding an element with a key that already exists has the effect of replacing the old element with the new one.

Note : only attributes of a type that is a subtype of Integer can be used as Qualifiers.

Note : with booch 95 components, if the type of the key is a subtype of integer (or any standard type) which is defined in an other package, then user must define a new "=" function in this other package.

For example, a new subtype of integer is defined into User_Type package.

User must define a new "=" function into this package.

```
package User_Type is
  subtype My_Subtype_Integer is Integer range 1..1000;
  function "=" (A : in My_Subtype_Integer; B : in My_Subtype_Integer)
    return Boolean;
private
end User_Type;

package body User_Type is
  function "=" ( A : in My_Subtype_Integer; B : in My_Subtype_Integer )
    return Boolean is
  begin
    return standard."="(A, B);
  end "=";
end User_Type;
```

4.13. Bidirectional relations

IBM® Rational® Rhapsody® Developer for Ada provides two implementations for bidirectional relations. By setting the Ada_CG.Relation.BidirectionalRelationsScheme to SubtypingAndRenaming (selected by default) or IntermediateParentClasses, it is possible to select which one is to be used.

4.13.1. SubtypingAndRenaming scheme

4.13.1.1 Implementation principles

This implementation supports bidirectional relations via the following mechanisms :

- the actual class members for classes participating in bidirectional relations are all generated in the same package so as to get reciprocal visibility.
- The classes are “emulated” in packages made up of subtyping and renaming of the class members.

4.13.2.1 Limitations

Note that there are limitations applicable to classes participating in bidirectional relations using that scheme which are described hereafter :

- They shall not contain elements with the same name (for types, static attributes or association ends role names) or signatures (for operations).
- They shall not be template classes
- Deferred initialization of public constants is not supported.
- They shall not contain statechart code
- They shall not have triggered operations
- Roundtrip is not supported
- Ports are not supported

4.13.2. IntermediateParentClasses scheme

4.13.2.1 Implementation principles

This scheme supports bidirectional relations via the following mechanisms :

- For each class participating in a bidirectional relation, an intermediate parent class is generated and inserted into the inheritance hierarchy.
- The bidirectional relations pointing to this class are redirected to point to its intermediate parent.

4.13.2.2 Using

Example :

Classes class_1 and class_2 have bidirectional relation.

If class_2 wants to get instance of “its **class_1**”, then the following accessor should be used :

get_downcast_itsClass_1 (this : in class_2_t)

4.13.3.2 Limitations

Most of the limitations of the SubtypingAndRenaming scheme are no longer relevant with this implementation. However there are still a few remaining limitations :

- This implementation is not compatible with Ada83 only configurations.

4.14. Ports

4.14.1. Limitations

Note that there are limitations applicable to usage of ports which are described hereafter :

- Ports contracts have to be implicit
- Multiplicities in links between ports have to be balanced
 - The number of source instances has to match the number of target instances
 - And the number of source ports has to match the number of target ports
- A port can have multiple contracts if the model is built for Ada 2005
- Fast ports are available only if the model is built for Ada 2005

4.14.2. Using ports

RiA generates code for ports in classes and for linking instances via ports.

When a class has ports, the code generator will create an additional Ada package called `<class>_port`. This Ada package contains all the material to declare the class's ports. The class contains a part of this `<class>_port` type.

Some functions are added in order to send some messages through ports.

- *Get_<Port_Name>(this : class_type) : return port_type*

This function gets the instance of the port we need.

- *<message>(this: class_type, port : port_type)*

This function sends the message "message" through the port "port" of the class "this". One function is created for each message defined in the port's interface. The usual way to send a message is to write :

```
<message>(this, Get_<Port_Name>(this));
```

If some parameters need to be passed with the message, then they are added after the port's instance :

```
<message>(this, Get_<Port_Name>(this), param : param_type);
```

The procedure is the same when you want to send an event through a port. Use the gen event function with port's name :

```
Gen_<event_name>(this, Get_<Port_Name>(this));
```

When an event is received from a port, it is possible to know which port received it. It is done with the function `Is_Port` of class `Oxf.I_Event`. Its signature is :

```
oxf.I_Event.is_port(event : oxf.I_event_t, port_ID : System.address)
```

The `port_ID` is given by the address of the port. You must get it like that :

```
<port_name>.Get_Inbound(Get_<port_name>(this)).Port_ID
```

4.14.3. Example 1 : behavioral port

Create a new model with 2 classes Class_0 and Class_1 which are parts of a class Build.

Create an interface interface_1 with one operation “message_0”.

Create port “Port_0” on Class_0.

Open features window of this port. Check behavioral check button, and add a provided interface in contract tab.

Copy this port in Class_1.

Open features window of this new port. Check reversed check button

Create a link between ports of the two classes.

Add an operation message_0() in class_0. Its implementation must be :

```
put_line("class_0 : message_0()");
```

Add an operation test() in class_1. Its implementation must be :

```
put_line("Class_1 : test()");
message_0(this,get_port_0(this));
```

Add an operation test() in class Build. Its implementation must be :

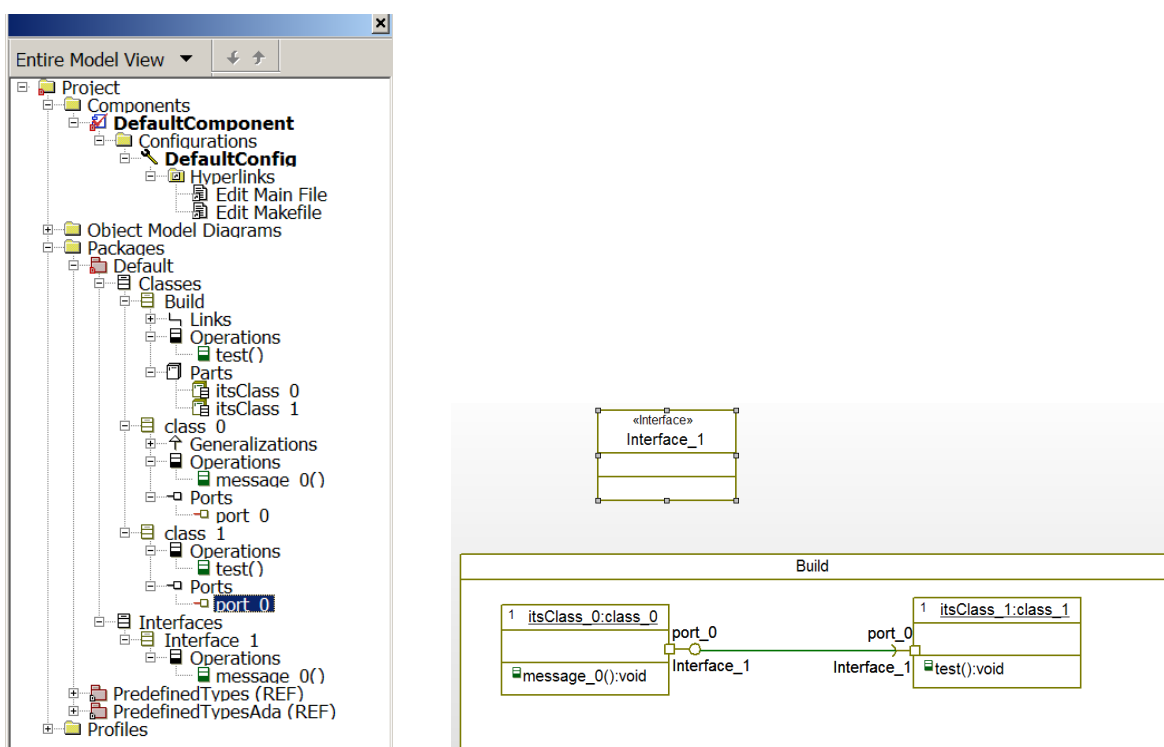
```
put_line("Build : test()");
```

Add a with and use clause for Ada.text_IO in all classes.

In configuration features, initialize the Build class in Initialization tab, and implement initialization code with :

```
Build.test(p_Build.all);
```

You should get a model like this one :



In this model test() function of class_1 will send message “message_0” through its port port_0.

4.14.4. Example 2 : fast ports

Take the model created below.

Set **Ada_CG:Component:AdaVersion** Property to Ada05 in order to build the model for Ada 2005.

In Class_0, Add a statechart with 2 states and one event “a” between the 2 states

In Class_1, Add a statechart with 2 states and one event “b” between the 2 states

Remove the contract of the 2 ports

Change the implementation of class_1.test() :

```
gen_event(this, Default.get_a, get_port_0(this));
```

To send an event through a fast port you must use the function Gen_Event. Its signature is:

```
gen_event(this: class type, event : Oxf.Event.Event_acc_t, port : port type);
```

The event must be created with the function defined in the event's package. Its name is :

```
<event package>.get_<event_name>
```

4.14.5. Multicast ports

This feature allows sending a message through one port to several ports in a single operation. It uses Booch components 95 in order to create an unbounded list of interfaces. Booch components are not provided with Rhapsody install. They must be installed manually if needed. (See 2.3 Booch components for more information.)

Multicast generation is controlled by the property

```
ADA_CG::Port::Support Multicasting
```

This property has the following values

Never : multicast instrumentation is never used

Smart : multicast instrumentation is used only if needed (see algorithm below)

Always : multicast instrumentation is always generated

Sending a multicast message

Sending a message to a port will automatically send it to all connected interfaces. The syntax is the same than with a single interface.

Example

The port Port_0 of an instance of class Class_0 is linked to several ports of instances of various other classes(which have the same interface as port_0)

To send a message “message” through Port_0 of class_0 you must write :

```
message(this, get_port_0(this));
```

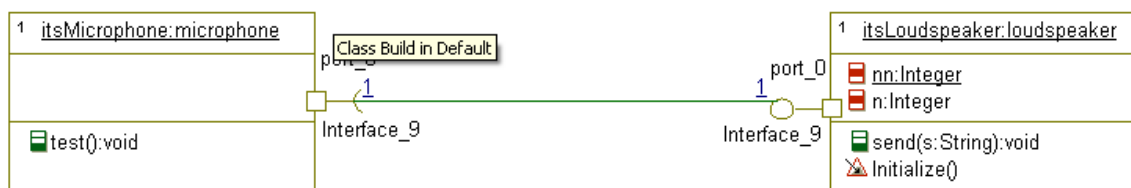
A message can be a procedure, an event or a triggered operation. Functions cannot be sent through multicast ports.

Fast ports also support multicast.

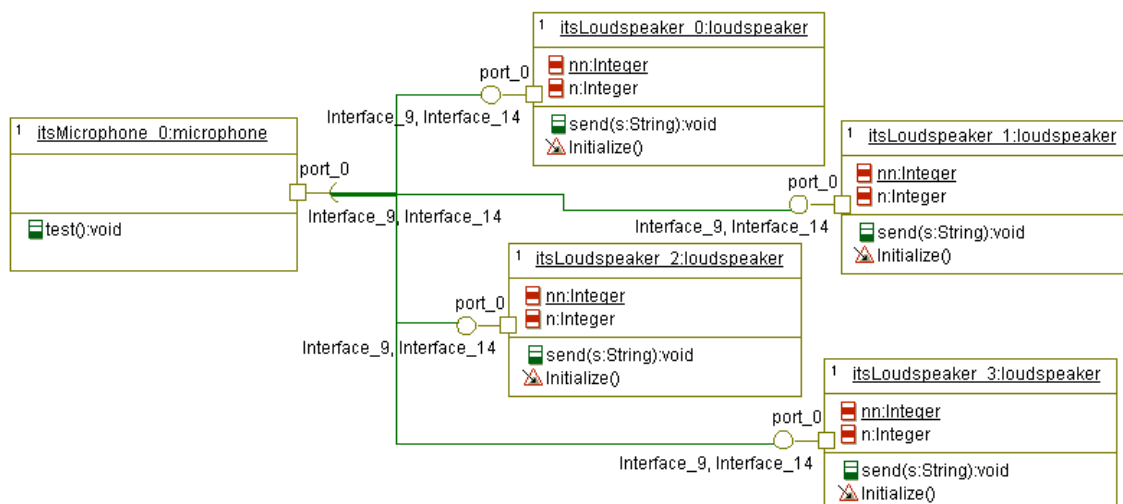
Link initialization with multiplicity

Multicast can be automatically initialized only if required and provided interface of the link belongs to class instance of multiplicity 1 and if its port has also multiplicity 1. In this case there is no ambiguity for initializing the links. Multiplicity of both ends of the link must be equal to 1.

Multiplicity equals 1 in both ends of the link (in this case multicast is not useful)



There are several instances of a class with provided interface.



Other operations on multicast ports

Some other operations can be done on a multicast Port in order to control the links between ports.

- Add a new provided interface
- Remove an existing provided interface
- Send a message to only one provided interface.

A provided interface can be disconnected from required interface. The following procedure does this.

```
Remove_<interface_name>(this : port_type, interface : interface_type);
```

Example

```
declare
    currentSourcePort    : microphone_port.port_0.port_type;
    currentTargetPort    : loudspeaker_port.port_0.port_type;
begin
    currentSourcePort := microphone.get_port_0(this.itsMicrophone_0.all);
    currentTargetPort := loudspeaker.get_port_0(this.itsLoudspeaker_0.all);

    microphone_port.port_0.remove_Interface_9(
        currentSourcePort,
        loudspeaker_port.port_0.get_Interface_9(currentTargetPort)
    );
end;
```

To add a new link to port, you just need to set port interface as usual.

Example

```
declare
    currentSourcePort    : microphone_port.port_0.port_type;
    currentTargetPort    : loudspeaker_port.port_0.port_type;
begin
    currentSourcePort := microphone.get_port_0(this.itsMicrophone_0.all);
    currentTargetPort := loudspeaker.get_port_0(this.itsLoudspeaker_0.all);
```

```
microphone_port.port_0.set_Interface_9(  
    currentSourcePort,  
    loudspeaker_port.port_0.get_Interface_9(currentTargetPort)  
);  
  
end;
```

It is possible to send a message to only one link.

A message “Message” can be sent to only one provided interface.

```
Message(this : class_type, port : port_type, interface : interface_type);
```

4.15. Ada Libraries

4.15.1. Creating an Ada Library

An Ada library can be created from a project by setting the “Library” option on the component.

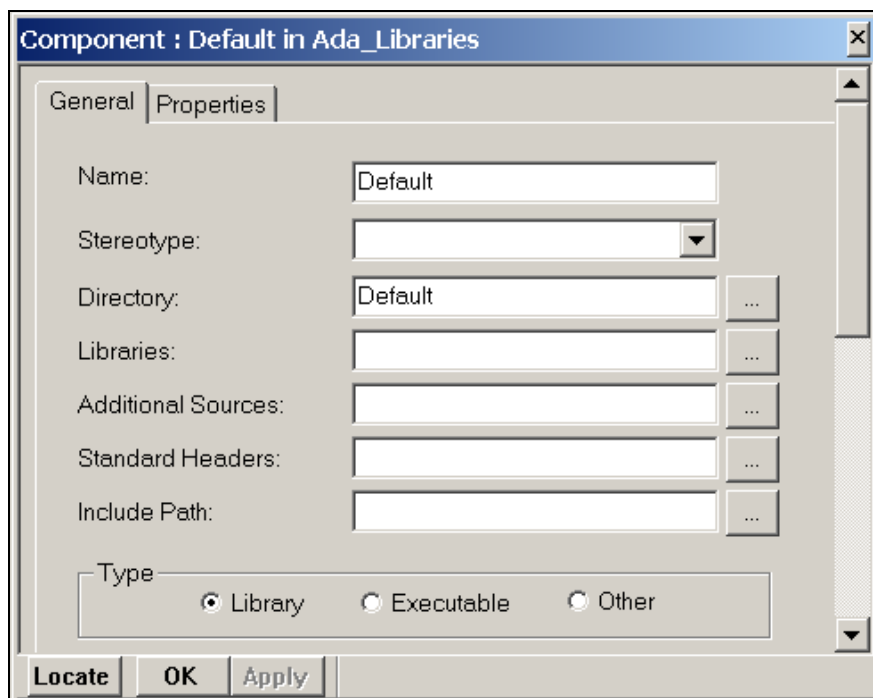


Figure 143: Setting the Component to Create a Library.

When the project is built, a library will be created in the directory specified by the configuration using the naming conventions described in the table below.

Compiler(s)	Library naming convention
GNAT	Lib<ComponentName>.a
GreenHills AdaMulti / Win32	<ComponentName>.lib
GreenHills AdaMulti / Integrity	Lib<ComponentName>.a
ObjectAda / Win32	<ComponentName>.lib
ObjectAda / Raven	N/A

Table 6 Compilers library naming conventions

4.15.2. Linking an Ada Library

To use an Ada library from another project, two pieces of information are required in the component properties. The first is the name of the library to use. This name depends on the compiler you are using, the syntax is described in the previous table. This name is put in the “Libraries” field. If there is more than one library to list, place a carriage return between the names.

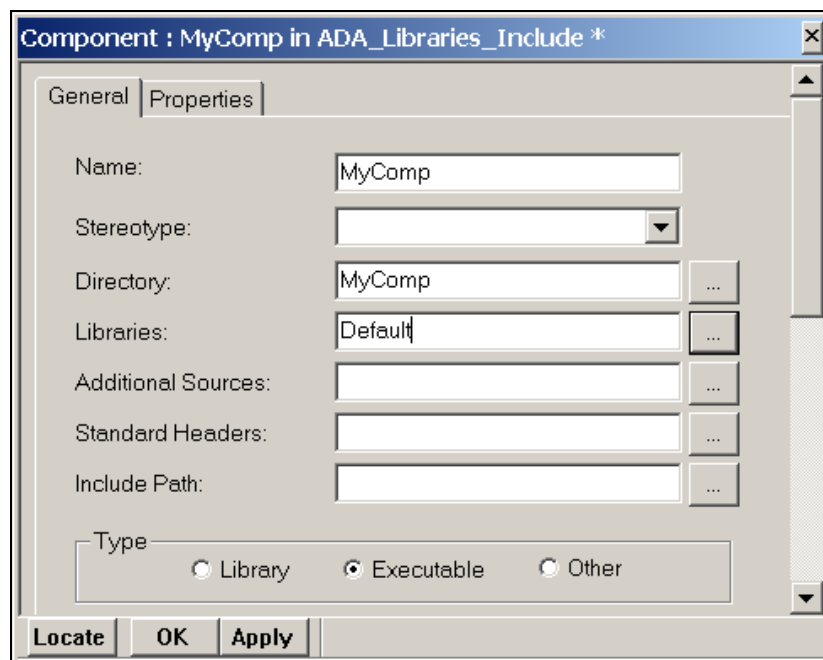


Figure 144: Using an Ada Library.

The location of the libraries also needs to be specified. The “Include Path” field is used to capture this information. The location of the library as well as the location of the sources for the library must be included. If there is more than one path to enter a carriage return should be used as a separator.

4.16. Configuration of Main File Generation

If the property “CG.Configuration.MainGenerationScheme” is set to “Full” on the configuration being generated, an entrypoint will automatically be created. The entrypoint will be named `main<Component Name>.adb`, and will produce an executable called `<Component Name>.exe`. This entrypoint will overwrite the output from any user-created entrypoint in the model.

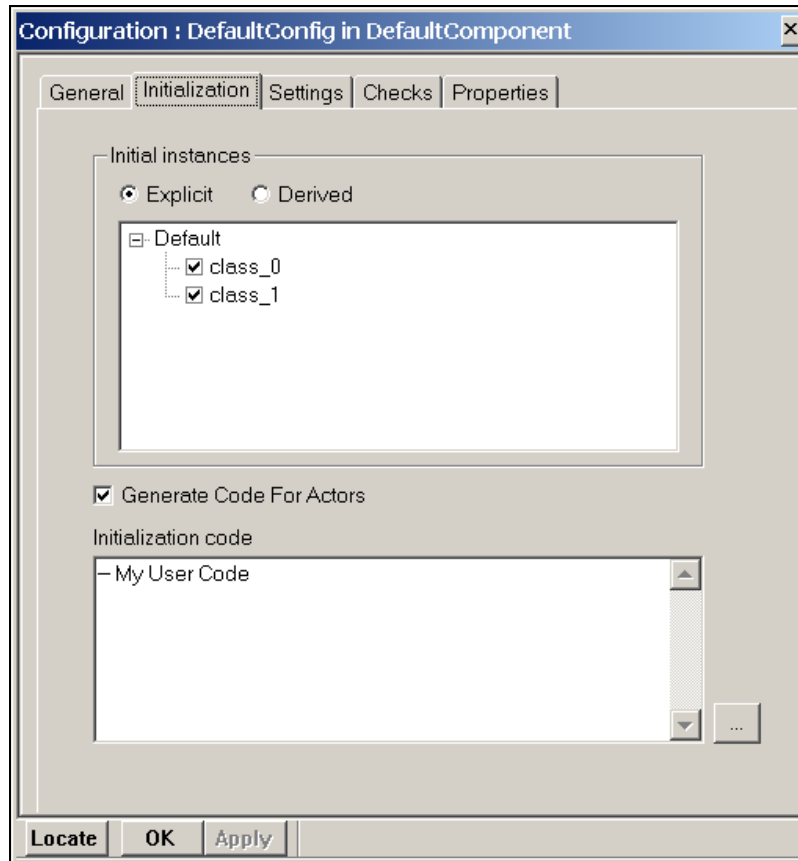


Figure 145: Configuration Instances.

4.16.1. With Clauses

A “With” Clause will be created for every class selected.

4.16.2. Configuration Prolog

The contents of the “Ada_CG.Configuration.ImplementationProlog” property on the configuration will appear just after the “With” clauses. It can be used to “With” other classes or packages as needed.

4.16.3. Instance Creation

If the selected class is not a singleton, a variable will be created to hold an access to the type of the class, and initialized with a new instance. If the class implements an Initialize procedure, the new instance will be initialized as well. If the class is a singleton and implements the Initialize procedure, the procedure will be called on the class.

4.16.4. RiADefaultActive Initialization

If there is a reactive class in the model that requires the RiADefaultActive class, the RiADefaultActive class will be initialized.

4.16.5. Reactive Instance Hookup

If there is both an instance of a reactive class, and an instance of the active context for this reactive class, the reactive instance will be registered on the active instance. If the reactive instance uses the RiADefaultActive class, this registration will be done as well.

4.16.6. Start Behavior

The RiADefaultActive class will be started if needed

As for the configuration initial instances, the Ada_CG.Relation.ObjectInitialization (Creation, Full, None) configuration property controls their initial behavior. By default it is set to “Full”, which means instances will be initialized and their behavior will be started. If the user would like the behavior not to be started, “Creation” should be selected.

4.16.7. User defined local variables

Variables declared in the “Ada_CG.Configuration.LocalVariablesDeclaration” property will appear in the declaration of the entrypoint.

4.16.8. User Initialization Code

Any code entered in the “Initialization Code” field on the configuration will be inserted into the entrypoint.

4.16.9. Configuration Epilog

The contents of the “Ada_CG.Configuration.ImplementationEpilog” property on the configuration will appear just after the “end MainDefaultComponent;” line.

```
With RiA_Default_Active;
With class_0;
With class_1;

procedure MainDefaultComponent is
  start_behavior_status : Boolean;
  p_class_0 : class_0.class_0_acc_t;
  p_class_1 : class_1.class_1_acc_t;
begin
  -- Instance Initialization
  RiA_Default_Active.Initialize;
  p_class_0 := new class_0.class_0_t;
  class_0.Initialize(p_class_0.all);
  p_class_1 := new class_1.class_1_t;

  -- Register Reactive Classes
  RiA_Default_Active.register_context_class_0(p_class_0.all);

  -- Start Behavior
  RiA_Default_Active.start;
  class_0.start_behavior(p_class_0.all, start_behavior_status);

  -- Initialize Package Instances

  -- User Initialization Code
  -- Your Configuration Initialization Code

end MainDefaultComponent;
```

Figure 146: Auto-generated Entrypoint.

The generated entrypoint can be viewed by selecting “Edit Configuration Main File” from the Configuration.

If animation is turned on, these instances will NOT be animated.

4.17. Instances Defined on a Package

4.17.1. Package Modifications

A child package will be created to handle the creation and initialization of any instances defined in a package. The name of the package will be <<Ada Package Name>>.RiA_Instances, where <<Ada Package Name >> is the name of the package where the instances are defined. When generating Ada83, the package name will be <<Ada Package Name>>_Instances. Each instance defined in the UML package will create global variables in this generated Ada package.

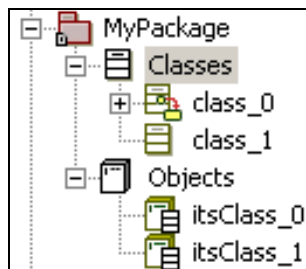


Figure 147: Global Instances on a Package.

Object: itsClass_0 in MyPackage

General | Attributes | Operations | Relations | Tags | Properties

Name: itsClass_0 L

Stereotype: [dropdown]

Main Diagram: [dropdown]

Concurrency: [dropdown]

Type: class_0 in MyPackage [dropdown] [icon]

Multiplicity: 1 [dropdown]

Initialization: [text] ...

Relation to whole

☐ Knows its whole as: [text]

Description: [text area] ...

Locate OK Apply

Figure 148: Global Instance with Multiplicity = 1.

Object : itsClass_1 in MyPackage

General | Attributes | Operations | Relations | Tags | Properties

Name: L

Stereotype:

Main Diagram:

Concurrency:

Type:

Multiplicity:

Initialization: ...

Relation to whole

☐ Knows its whole as:

Description:

Locate OK Apply

Figure 149: Global Instance with Multiplicity > 1.

Variables will be created in the public part of the package specification for each instance defined. The type of variable will depend on the setting of the “CG.Relation.Implementation” property for each of the relations. If the instance is a singleton, no variable will be created. If the multiplicity is greater than 1 but not *, an array will be used of the given size. If the multiplicity is given as *, an array will be generated of size 100. The elements in this array will not be initialized.

The appropriate “With” statements will be added to the package specification for each Class instantiated.

These instances will be created in the procedure *Initialize_Relations*, and they will be initialized if an Initialize operation exists for their class.

Likewise, these instances will be finalized in the procedure *Finalize_Relations* if a Finalize operation exists for their class.

If the instances have a statechart, the *start_behavior* procedure will be called to start the behaviors of the instances in the *Initialize_Relations* procedure. The instances will be hooked up to their active context if needed as well. If the instances are active, they will be started by calling the *start* procedure.

```

With MyPackage.class_0;
With MyPackage.class_1;

package MyPackage.RiA_Instances is

    -- Instance Declarations
    itsClass_0 : MyPackage.class_0.class_0_acc_t;

    type itsClass_1_card_t is new Positive range 1..10;
    type itsClass_1_acc_lst_t is array(itsClass_1_card_t) of MyPackage.class_1.class_1_acc_t;
    itsClass_1 : itsClass_1_acc_lst_t;

    procedure Initialize_Relations;

    procedure Finalize_Relations;

end MyPackage.RiA_Instances;

```

Figure 150: The Instances Package Specification.

```

With RiA_Default_Active;

package body MyPackage.RiA_Instances is

    procedure Initialize_Relations is
        start_behavior_status : Boolean;
    begin
        -- Create the global instances
        itsClass_0 := new MyPackage.class_0.class_0_t;
        MyPackage.class_0.Initialize(itsClass_0.all);
        for i in itsClass_1_card_t loop
            itsClass_1(i) := new MyPackage.class_1.class_1_t;
        end loop;

        -- Register the contexts
        RiA_Default_Active.register_context_class_0(itsClass_0.all);

        -- Start behavior
        MyPackage.class_0.start_behavior(itsClass_0.all, start_behavior_status);

        -- Hookup instances

        -- Ensure that there is at least one statement
        null;
    end Initialize_Relations;

    procedure Finalize_Relations is
    begin
        MyPackage.class_0.Finalize(itsClass_0.all);

    end Finalize_Relations;

end MyPackage.RiA_Instances;

```

Figure 151: The Instances Package Body.

If a link is created between the instances, the relation will be initialized as well in the *Initialize_Relations* procedure.

Only the *Initialize_Relations* procedure will be called from the auto-generated entrypoint.

4.18. User-Defined header and footers

4.18.1. Available properties

By modifying the following properties on the project, component, configuration, package or class level the user can get Rhapsody to use custom headers and footers instead of the default ones for generated files.

- Ada_CG.File.ImplementationFooter
- Ada_CG.File.ImplementationHeader
- Ada_CG.File.SpecificationFooter
- Ada_CG.File.SpecificationHeader

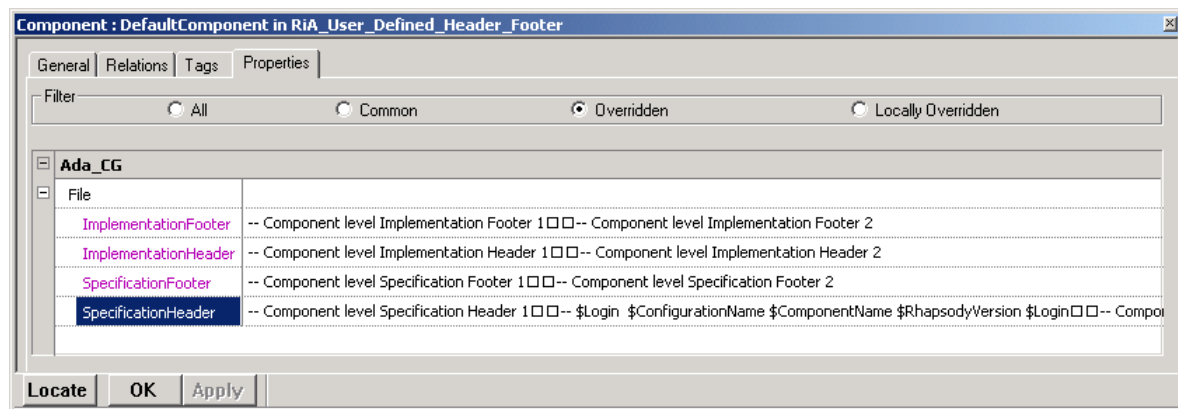


Figure 152 Defining custom header and footer at the component level

The usual Rhapsody inheritance rules apply for these properties, which means that you can refine your settings from the project level all the way down to the class.

These four properties are independent, which means that you can use a single project level setting for say the specification header and have different configuration level settings for the implementation header.

Those properties can also be updated at class level or at operation level (for separate operation). This can be useful to set change history log for example.

If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the ADA_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the ADA_CG::Configuration::DescriptionEndLine property.

4.18.2. Keyword substitution

Keyword based substitution is supported inside of these headers and footers.

The following keywords are supported:

- * \$ProjectName - The project name.
- * \$ComponentName - The component name.
- * \$ConfigurationName - The configuration name.
- * \$ModelElementName - The name of the element mapped to the file. If there is more than one, this is the name of the first element.
- * \$FullModelElementName - The name of the element mapped to the file, including the full path. If there is more than one, this is the name of the first element.

- * \$CodeGeneratedDate - The generation date.
- * \$CodeGeneratedTime - The generation time.
- * \$RhapsodyVersion - The version of Rational Rhapsody that generated the file.
- * \$Login - The user who generated the file.
- * \$CodeGeneratedFileName - The name of the generated file.
- * \$FullCodeGeneratedFileName - The full file name.
- * \$Description – the description of the class or package

Note the following:

- * Keyword names can be written in parentheses. For example:

\$(Name)

- * If the value of a keyword is a MultiLine, each new line (except the first one) starts with the value of the ADA_CG::Configuration::DescriptionBeginLine property; each line ends with the value of the ADA_CG::Configuration::DescriptionEndLine property.

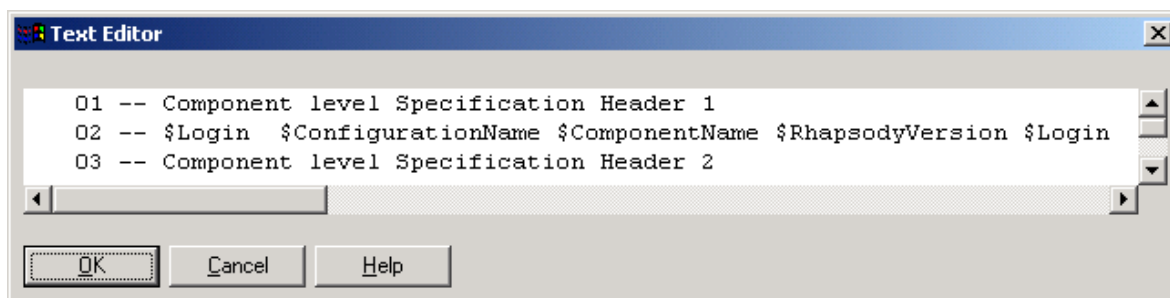


Figure 153 Inserting keywords inside user-defined header and footer

```
--++ package Default
-- Component level Specification Header 1
-- sodius InheritedSettings DefaultComponent 6.0 sodius
-- Component level Specification Header 2

--++ class class_0
package class_0 is

    type class_0_t;
    type class_0_acc_t is access all class_0_t;

    type class_0_t is tagged null record;

private

end class_0;
-- Component level Specification Footer 1
-- Component level Specification Footer 2
```

Figure 154 Example of generated code using user-defined header and footer

4.18.3. Script Evaluation

It is also possible to put script names inside of headers and footers so that they get evaluated at code generation time.

The name has to be framed following this convention `[[scriptName]]`.

The script has to be applicable for the model element for which the header is being evaluated, otherwise an error message will be displayed.

4.19. Custom makefiles

4.19.1. Introduction

Makefiles are usually generated by the Ada code generator. However they can also be created manually. This document describes all the features of custom makefiles, and gives some examples of makefile creation.

4.19.2. Features

4.19.1.2 Entry point

A makefile is built from 2 entry point properties

Ada_CG.<ENV>.MakeFileNameForExe

This property sets the name of the makefile. The extension must be inserted in this property.

Ada_CG.<ENV>. MakeFileContentForExe

This property is the file template. It can contain some text and some keywords. Keywords will be interpreted by CG.

There are some entry point properties to generate makefiles for executable project and for library projects. Different entry points will be used depending on the component property.

Executable project

MakeFileNameForExe

MakeFileContentForExe

Library project

MakeFileNameForLib

MakeFileContentForLib

If several files must be generated, then property names must be followed by a number from 1 to N. The CG will automatically scan all entry point properties.

MakeFileNameForExe1

MakeFileContentForExe1

MakeFileNameForExe2

MakeFileContentForExe2

2 additional entry points must be added for Green Hills compilers in order to generate entry point build files.

FilenameEntrypointBuildFileContent

EntrypointBuildFileContent

4.19.2.2 Keywords

Keywords are replaced by CG with some text. This text can also contain other keywords, which will be interpreted recursively.

Keywords are preceded by the character “\$”

There are 2 kinds of keywords: property keyword and macro.

4.19.3.2 Property keyword

Property keywords can be any of the properties of the current environment. This keyword will be replaced by the content of the property. Its syntax is

\$<property_name>

Example

Property MakeExtension String ".bat"

Property MakeFileNameForExe String "makefile\$MakeExtension"

In the second property, CG will replace \$MakeExtension by the content of property Ada.CG.<ENV>.MakeExtension. The result will be :

“makefile.bat.”

4.19.4.2 Macro

A Macro is a keyword which will be interpreted by CG to execute a script. Macros are recognized because they start with “\$AdaCG”.

Some macros don't begin by this prefix :

- \$ComponentName
- \$ProjectName
- \$OMROOT

4.19.5.2 **Creating new macro**

If needed, users can add some new macros which will call Code Generator rule. The user must have knowledge of the RiA Code Generator rules in order to do this.

A property file must be created, which will make the mapping between the name of the macro and the script to call.

The file name must be :

<Rhp_install_dir>\share\properties\MakeFileCommand.ini

Syntax to fill this file is

<Macro_Name>=<script_name> for a script defined in configuration level

<Macro_Name>=Project.<script_name> for a script defined in project level

4.19.3. **Standard Macros and property Keywords**

This list of macros already known by the code generator:

AdaCGAnimationInclude	If animation is enabled, this macro will get property AnimationLibraries83Path, AnimationLibraries95Path or AnimationLibrariesNew95Path, depending on used FWK
AdaCGAnimLib	If animation is enabled, this macro will get property AnimationLibraries
AdaCGBehavioralInclude	If animation is enabled, or if model needs Behavioral FWK, this macro will get property BehavioralLibraries83Path, BehavioralLibraries95Path or BehavioralLibrariesNew95Path
AdaCGBehavioralLib	This macro gets properties BehavioralLibraries83Lib, BehavioralLibraries95Lib or BehavioralLibrariesNew95Lib depending on used FWK
AdaCGBoochPath	If relations are used, this macro will get property Booch83Path or Booch95Path depending on used Booch component.
AdaCGBoochFiles	<p>This macro gets some property depending on the following condition :</p> <pre> if Uses_Relations_Include{ if Use_Booch_95_Components{ if Needs_Relations_Include_Bounded_Qualified{ get property Booch95RelationsIncludeBoundedQualified } if Needs_Relations_Include_Unbounded{ </pre>

	<pre> get property Booch95RelationsIncludeUnbounded} if Needs_Relations_Include_Unbounded_Not_Qualified{ get property Booch95RelationsIncludeUnboundedNotQualified} if Needs_Relations_Include_Unbounded_Qualified{ get property Booch95RelationsIncludeUnboundedQualified} } if Use_Booch_83_Components { if Needs_Relations_Include_Bounded_Qualified{ get property Booch83RelationsIncludeBoundedQualified} if Needs_Relations_Include_Unbounded{ get property Booch83RelationsIncludeUnbounded} if Needs_Relations_Include_Unbounded_Not_Qualified{ get property Booch83RelationsIncludeUnboundedNotQualified} if Needs_Relations_Include_Unbounded_Qualified{ get property Booch83RelationsIncludeUnboundedQualified} } } </pre>
AdaCGAdaPath	<p>This macro gets all generated folders. All folders are separated by “\n”. This macro uses the property AdaPathContent in order to format this list. See AdaPathContent description for more details.</p>
AdaCGAdaVersionSwitch	<p>This macro will get property Ada83Switch, Ada95Switch or Ada2005Switch, depending on Ada version used. If a model in Ada83 is animated, version switch will be forced to Ada95.</p>
AdaCGDebugSwitch	<p>This macro generates the text of property CompileDebug if build set of configuration setting is set to debug.</p>
AdaCGAdditionalSources	<p>This macro gets additional sources from configuration settings. It uses property AdditionalSourcesTemplate in order to format this text. See AdditionalSourcesTemplate description for more details.</p>
AdaCGUserIncludPath	<p>This macro gets user include path from configuration settings. It uses property IncludePathTemplate in order to format this text. See IncludePathTemplate description for more details.</p>
AdaCGLibraries	<p>This macro gets library from configuration settings. It uses property LibrariesTemplate in order to format this text. See LibrariesTemplate description for more details.</p>
AdaCGCompileSwitches	<p>This macro gets compile switches from configuration settings</p>
AdaCGLinkSwitches	<p>This macro gets link switches from configuration settings</p>
AdaCGFileSpecList	<p>This Macro makes the list of all generated spec files. The spec file is added to the make file using the SpecTemplate property.</p> <p>This macro uses properties</p>

	<ul style="list-style-type: none"> - SpecTemplate<index of makefile> - ProtectedStartTagFormat1<index of makefile> - ProtectedEndTagFormat1<index of makefile> <p><index of makefile> is the index of makefile entry point</p>
AdaCGFileBodyList	<p>This Macro makes the list of all generated body files This macro uses properties</p> <ul style="list-style-type: none"> - BodyTemplate<index of makefile> - ProtectedStartTagFormat1<index of makefile> - ProtectedEndTagFormat1<index of makefile> <p><index of makefile> is the index of makefile entry point</p>
AdaCGObjectAdaMakefile	<p>This macro generates makefile for OBJECTADA compiler. "Compiler" property should be set to OBJECTADA.</p>
AdaCGGnatMakefile	<p>This macro generates makefile for GNAT or GNATVxWorks compiler. "Compiler" property should be set to GNAT or GNATVxWorks.</p>
AdaCGGnatAdc	<p>This macro generates gnat.adc file for GNAT or GNATVxWorks compiler. "Compiler" property should be set to GNAT or GNATVxWorks.</p>
AdaCGOptionalAdaPath	<p>Used to split AdaCGGnatMakeFile macro</p>
AdaCGGnatchopCommands	<p>Used to split AdaCGGnatMakeFile macro</p>
AdaCGCommands	<p>Used to split AdaCGGnatMakeFile macro</p>
AdaCGArchiveCommand	<p>Used to split AdaCGGnatMakeFile macro</p>
AdaCGIDENAME	<p>Get value of IDENAME tag of current configuration (for RiA in eclipse)</p>
AdaCGIDEProject	<p>Get value of IDEProject tag of current configuration (for RiA in eclipse)</p>
AdaCGIDEWorkspace	<p>Get value of IDEWorkspace tag of current configuration (for RiA in eclipse)</p>
AdaCGFilenameMULTIEntrypointBuildFile	<p>This macro generates makefile name for user entry point for GHS tools.</p>
AdaCGMULTIEntrypointBuildFile	<p>This macro generates makefile content for user entry point for INTEGRITY or MultiWin32 environment</p>
AdaCGMULTI4EntrypointBuildFile	<p>This macro generates makefile content for user entry point for INTEGRITY5 or Multi4Win32 environment</p>
AdaCGMultiMakeFile	<p>This macro generates makefile for INTEGRITY or MultiWin32 environment. "Compiler" property should be set to INTEGRITY or MultiWin32.</p>
AdaCGMultiEntryPoint	<p>This macro generates entry point files for INTEGRITY or MultiWin32 environment. "Compiler" property should be set to INTEGRITY or MultiWin32.</p>

AdaCGMultiSources	This macro generates sources files for INTEGRITY or MultiWin32 environment. "Compiler" property should be set to INTEGRITY or MultiWin32.
AdaCGMulti4MakeFile	This macro generates makefile for INTEGRITY5 or Multi4Win32 environment. "Compiler" property should be set to INTEGRITY5 or Multi4Win32.
AdaCGMulti4EntryPoint	This macro generates entry point file for INTEGRITY5 or Multi4Win32 environment. "Compiler" property should be set to INTEGRITY5 or Multi4Win32.
AdaCGMulti4Sources	This macro generates sources file for INTEGRITY5 or Multi4Win32 environment. "Compiler" property should be set to INTEGRITY5 or Multi4Win32.
AdaCGDefaultActiveClass	If default active class is need for FWK83, then this macro will get the property ActiveClassInclude.
AdaCGFileList	This macro generates the list of generated spec files. It uses property FileTemplate in order to format this list.
AdaCGOMROOTSingleSlashes	Generates OMROOT with back slashes.
AdaCGOMROOTDoubleSlashes	Generates OMROOT with double back slashes
AdaCGOMROOTForwardSlashes	Generates OMROOT with forward slashes.
OMROOT	Generates OMROOT with back slashes. This string is quoted if the property QuoteOMROOT is set to "True".

This list of additional properties shows properties which are used by macro listed below.

CompileDebug	This property contains debug switches. Is used by AdaCGDebugSwitch macro.
Ada83Switch Ada95Switch Ada2005Switch	Those properties contain Ada version switches.
AnimationLibraries83Path AnimationLibraries95Path AnimationLibrariesNew95Path	Those properties contain Animation libraries path for each FWK
BehavioralLibraries83Path BehavioralLibraries95Path BehavioralLibrariesNew95Path	Those properties contain Behavioral libraries path for each FWK
AnimationLibraries	This property give the animation library located at <Rhp_Install_Dir> Share\LangC\Lib
AdaPathContent	This property contains patterns to be replaced. The format

	<p>of this property is <code><"The_String_To_Replace"><"Is_Replaced_With"></code></p> <p>For example <code><"\n"><";"></code></p> <p>This will replace \n by ;</p>
Compiler	<p>This property sets the name of compiler. It is used to be able to reuse already existing rules with other environment.</p> <p>For example, if you create a new Environment called GNAT_1, and if you use the Macro AdaCGGnatMakefile, then some part of rules won't work as expected because this new environment variable is unknown. So a new property is created to replace this environment name with compiler name.</p> <p>If a new compiler is created, this property is not useful, because it is unknown by CG.</p> <p>The values for this property should be :</p> <ul style="list-style-type: none"> GNAT GNATVxWorks OBJECTADA MultiWin32 Multi4Win32 INTEGRITY INTEGRITY5 RAVEN_PPC SPARK
Booch83Path Booch95Path	Those properties contain booch components path
Booch95RelationsIncludeBoundedQualified Booch95RelationsIncludeUnbounded Booch95RelationsIncludeUnboundedNotQualified Booch95RelationsIncludeUnboundedQualified Booch83RelationsIncludeBoundedQualified Booch83RelationsIncludeUnbounded Booch83RelationsIncludeUnboundedNotQualified Booch83RelationsIncludeUnboundedQualified	Those properties contain the list of booch component files needed for each kind of relation.
AdditionalSourcesTemplate IncludePathTemplate LibrariesTemplate	<p>Those 3 properties allow replacing some string of configuration fields by some other string. Syntax is :</p> <p>[optional_string] <code><"The_String_To_Replace"><"Is_Replaced_With"></code></p>

	<p>For example</p> <p>Property LibrariesTemplate Multiline " -largs <\"\\\"><\" -\\\"><\"\\n\"><\" -\\\"><\"\\r\\n\"><\" -\\\"><\", \"><\" -\\\">"</p> <p>If user update configuration libraries with the string "lib1,lib2", then it will be generated like this :</p> <p>-largs -llib1 -llib2</p>
<p>BodyTemplate</p> <p>SpecTemplate</p>	<p>Those properties are use to format the file list generated by Macros AdaCGFileBodyList and AdaCGFileBodyList.</p> <p>The following keyword can be used in this template :</p> <p>ConfigurationRelativeFilename ConfigurationRelativeBodyFilename ConfigurationRelativeBodyFilename SpecRelativeFilename BodyRelativeFilename FileName GNATCommandFileName AdaCGRiAFullName</p> <p>For example for a class class_0 defined in default package those keywords will produce :</p> <p>ConfigurationRelativeFilename : .\Default\class_0 ConfigurationRelativeBodyFilename : .\Default\class_0.adb ConfigurationRelativeSpecFilename : .\Default\class_0.ads SpecRelativeFilename : class_0.ads BodyRelativeFilename : class_0.adb FileName : class_0 FullNameDashes : class_0 ("." Are replaced by "-") AdaCGRiAFullName : Default::class_0</p>
<p>ProtectedStartTagFormat</p> <p>ProtectedEndTagFormat</p>	<p>Those properties are used to format the file list generated by Macros AdaCGFileBodyList and AdaCGFileBodyList.</p> <p>They are used to add some tags in the list in order to help CG to add only new text in the file.</p> <p>The same keywords than properties BodyTemplate and SpecTemplate can be used.</p>

4.19.4. New environment creation

Create a new file SiteAda.prp

Add the following properties

```
Subject Ada_CG

    Metaclass Configuration

        Property Environment Enum
"GNAT, INTEGRITY, INTEGRITY5, MultiWin32, Multi4Win32, OBJECTADA, RAVEN_PPC, SPARK, GNATV
xWorks, New_Env" "GNAT"

        end

    end
```

Copy an environment which is as close as possible to your new one, from sodius.prp to siteAda.prp. It must be copied just before the last “end” of the file.

Then modify entry points and add some new properties which will describe your new files.

4.19.5. Use cases

Custom makefiles can be created for several purposes. For example user needs to make a small modification of a current makefile. Or user wants to use a new compiler which is not supported by code generator. This chapter will describe how this can be done.

4.19.1.5 Create a new makefile for an unknown compiler.

Fast solution

Just add the text of the make file in the entry point property. This will generate always the same makefile. This solution can be used to make a quick test, but it cannot take into account all possible configurations.

Configurable solution

A new environment could be used for different kind of configuration. User can use different frameworks or Ada versions, or he can set animation or not. In order to take into account automatically those configurations, custom makefiles can be written with some macros which will automatically select the desired property depending on configuration properties.

Makefiles depend also on the model structure. A list of generated files or folders can be added to makefiles with the possibility to format the list.

Take into account Framework

If your model has events or statecharts, then a Framework must be used. Three different Frameworks can be used (FWK83, FWK95, NewFWK95), depending on property **Ada_CG:Component:UseAdaFramework**. Those Frameworks have been first generated and compiled for your compiler and your environment. You should know what their locations are. The path of your different Framework should be set in properties BehavioralLibraries83Path, BehavioralLibraries95Path and BehavioralLibrariesNew95Path, and you should invoke them by using macro AdaCGBehavioralInclude, which will select the correct property, depending on used Framework.

Take into account animation

If model is animated, you should add some libraries

C Animation libraries which are :

```
%OMROOT%\LangC\lib\AdaWinaomanim.lib
%OMROOT%\LangC\lib\AdaWinoxfirst.lib
%OMROOT%\LangC\lib\AdaWinomcomappl.lib
```

C libraries should be compiled as explained in Ada_User_Guide.pdf.

Ada animation libraries path which are :

```
%OMROOT%\LangAda\aoom for FWK 83.
%OMROOT%\LangAda\aoom_95 for FWK95 and newFWK95.
```

You should use AdaCGAnimationInclude and AdaCGAnimationLib macros with there associated properties, in order to add animation facilities into makefiles.

Take into account relations

If your model contains relations which use booch components, then some booch components files must be added to your makefile.

Booch components are located in :

```
%OMROOT%\LangAda95\booch_ada_83\src\
%OMROOT%\LangAda95\booch_ada_95\src\
```

For each kind of relation, the Code generator uses a different set of Booch components.

Booch components 83

Relations_Include_Bounded_Qualified

```
map_simple_noncached_concurrent_bounded_managed_noniterator.ads
map_simple_noncached_concurrent_bounded_managed_noniterator.adb
```

Relations_Include_Unbounded

```
Storage_Manager_Concurrent.ads
Storage_Manager_Concurrent.adb
```

Relations_Include_Unbounded_Not_Qualified

```
list_single_unbounded_controlled.ads
```

list_single_unbounded_controlled.adb
list_utilities_single.ads
list_utilities_single.adb
list_search.ads
list_search.adb

Relations_Include_Unbounded_Qualified
Storage_Manager_Concurrent.ads
Storage_Manager_Concurrent.adb
map_simple_noncached_concurrent_unbounded_managed_noniterator.ads
map_simple_noncached_concurrent_unbounded_managed_noniterator.adb

Booch components 95

Relations_Include_Bounded_Qualified
bc.ads
bc-containers.ads
bc-containers.adb
bc-containers-maps.ads
bc-containers-maps.adb
bc-containers-maps-bounded.ads
bc-containers-maps-bounded.adb
bc-support.ads
bc-support-hash_tables.ads
bc-support-hash_tables.adb
bc-support-bounded_hash_tables.ads
bc-support-bounded_hash_tables.adb

Relations_Include_Unbounded
bc.ads
bc-support.ads
bc-support-standard_storage.ads
bc-support-unbounded.ads
bc-support-unbounded.adb

Relations_Include_Unbounded_Not_Qualified
bc.ads
bc-containers.ads
bc-containers.adb
bc-containers-collections.ads
bc-containers-collections.adb
bc-containers-collections-unbounded.ads
bc-containers-collections-unbounded.adb

Relations_Include_Unbounded_Qualified
bc.ads
bc-containers.ads
bc-containers.adb
bc-containers-maps.ads
bc-containers-maps.adb
bc-containers-maps-unbounded.ads
bc-containers-maps-unbounded.adb

You should use AdaCGBoochPath and AdaCGBoochFiles macro with their associated properties to add Booch component facilities into makefiles.

4.19.2.5

Modify the generated code of a current environment.

For example, you need to change the primaryTarget of your project for INTEGRITY5 environment. But this is hardcoded by the code generator. Instead of calling the macro AdaCGMulti4MakeFile, you will call a set of macros which are called by it. To do this you must use the Rules composer and generate code in debug mode. Open the debug hierarchy and find the script which is called by the macro.

Copy this script in a new property and change the script names into some custom names.

The script (get from Ada code generator rules) called by AdaCGMulti4MakeFile does this :

```
[#script]

#!gbuild

# Generated by Rhapsody

${self.MULTI_4_Top_Settings}${self.MULTI_4_Get_Library_Options}${self.MULTI_4_Get_Executable_Options}

${self.MULTI_4_Debug_Switches}${self.MULTI_4_Additional_Options}${self.Makefile_Compile_Switches}${self.Makefile_Link_Switches}

# Generation directories settings

    -object_dir=obj

    --ada_info_dir info

    --ada_xref_dir xref

${self.MULTI_4_Ada_Path}

${self.MULTI_4_User_Include_Path}${self.MULTI_4_User_Libraries}${self.MULTI_4_RiA_Anim_Libs_Link_Options}

${self.MULTI_4_Anim_And_Behavioral_Includes}

Sources.gpj          ${self.MULTI_4_Component_Type}

${self.MULTI_4_Entrypoints}[/#script]
```

The Macro AdaCGMulti4MakeFile should be replaced by the following text in property MakeFileContentForExe3

```
#!gbuild

# Generated by Custom template

primaryTarget=NEW_PRIMARY_TARGET_integrity.tgt

[INTEGRITY Application]

    -o $ComponentName$ExeExtension

# Target definition

    -bsp $BLDTarget
```

```

        -os_dir=$IntegrityRoot

$BLDMainExecutableOptions

$MULTI4DebugSwitches$MULTI4AdditionalOptions$MakefileCompileSwitches$Make
fileLinkSwitches

# Generation directories settings

        -object_dir=obj

        --ada_info_dir info

        --ada_xref_dir xref

$MULTI4AdaPath

$MULTI4UserIncludePath$MULTI4UserLibraries$MULTI4RiAAnimLibsLinkerOptions

$MULTI4AnimAndBehavioralIncludes

Sources.gpj          [Project]

Main$ComponentName$MakeExtension[program]

```

This new text will do the same job than the RulesComposer script. The names of the rules have been replaced by new Macros. The rule `MULTI_4_Top_Settings` has been replaced directly by some text, because this is the part of generation that we want to modify.

In order to enable the code generator to understand those new macros, a new initialization file must be updated, to create mappings between macros and rules.

Create the file :

<Rhp_install_dir>\share\properties\MakeFileCommand.ini

In this file, you will map custom names to a script.

```

MULTI4DebugSwitches          =Project.MULTI_4_Debug_Switches
MULTI4AdditionalOptions      =Project.MULTI_4_Additional_Options
MakefileCompileSwitches     =Project.Makefile_Compile_Switches
MakefileLinkSwitches        =Project.Makefile_Link_Switches
MULTI4AdaPath                =Project.MULTI_4_Ada_Path
MULTI4UserIncludePath        =Project.MULTI_4_User_Include_Path
MULTI4UserLibraries          =Project.MULTI_4_User_Libraries
MULTI4RiAAnimLibsLinkerOptions=Project.MULTI_4_RiA_Anim_Libs_Linkers
MULTI4AnimAndBehavioralIncludes=Project.MULTI_4_Anim_And_Behavioral_Includes

```

The prefix “Project.” means that this script is defined at project level. If this prefix is omitted, then the script must be defined at configuration level.

You should also take care of Framework location. You may have generated a new Framework for your new environment. So Framework location has changed. To do this, replace the Macro `MULTI4AnimAndBehavioralIncludes` by `AdaCGBehavioralInclude`. This macro will get information from properties `BehavioralLibraries83Path`, `BehavioralLibraries95Path` or `BehavioralLibrariesNew95Path` depending on Framework version.

Property `BehavioralLibraries83Path` should be for example :

```
--ada_elab_dirs '\'  
$AdaCGOMROOTForwardSlashes/LangAda83/oxf/NewEnv_sim800'  
  
-L'$AdaCGOMROOTDoubleSlashes\\\\LangAda83\\\\oxf\\\\ NewEnv  
_sim800'
```

A new property “Compiler” should also be added and set to “INTEGRITY5” in order to be sure that the code generator will interpret the macro for the correct compiler.

5. SPARK code generation

IBM® Rational® Rhapsody® Developer for Ada enables the generation of SPARK annotations from UML models.

In order to analyse the generated annotations, you need the SPARK Examiner, available from Praxis High Integrity Systems, much like you need an Ada compiler to compile the code generated by Rhapsody.

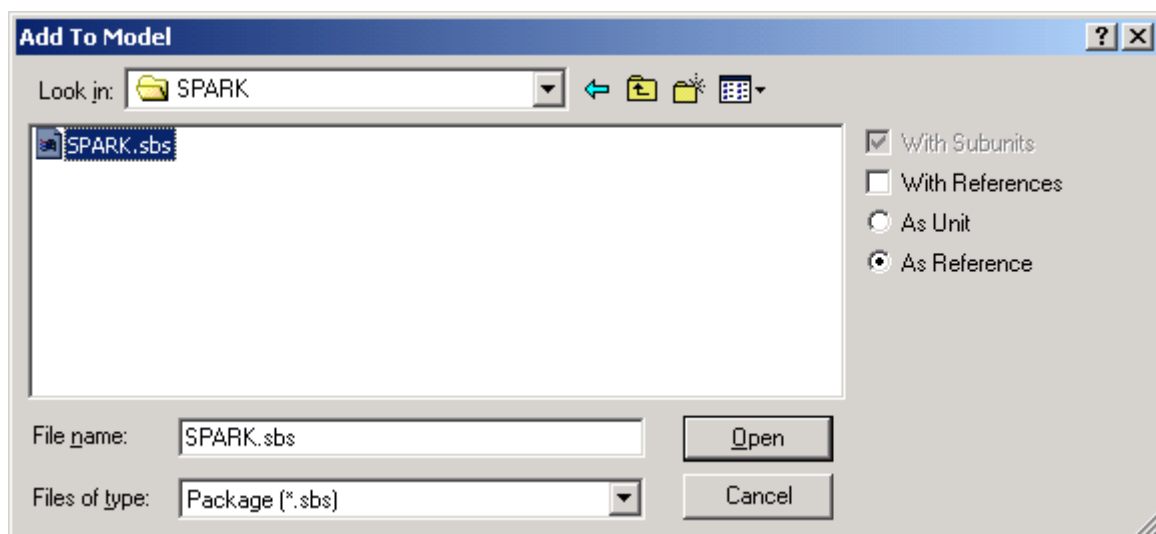
IBM® Rational® Rhapsody® Developer for Ada supports the SPARK toolkit version 7.2 and above. Contact Praxis High Integrity Systems to get the appropriate updates if you have a previous version.

5.1. Enabling SPARK code generation

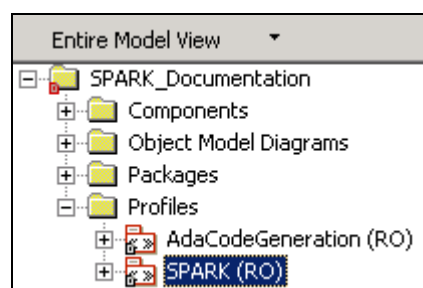
5.1.1. Adding the SPARK profile to the model

A SPARK profile is provided with IBM® Rational® Rhapsody® Developer for Ada that allows modeling of SPARK annotations. To use this profile on a new or existing Rhapsody model, from the model :

- select “File, Add to model”...
- Navigate to the <Rhapsody>\Share\Profiles\SPARK directory
- Select the file type “Package (*.sbs)”
- Select the “As Reference” radio button
- Click “Open”



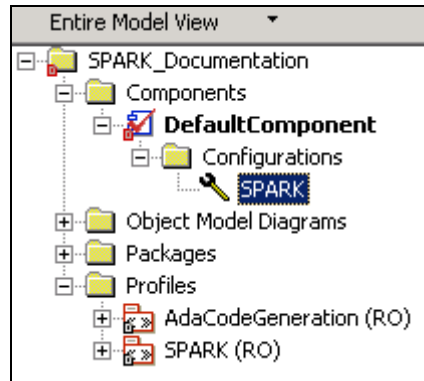
The SPARK Profile is now added to your model.



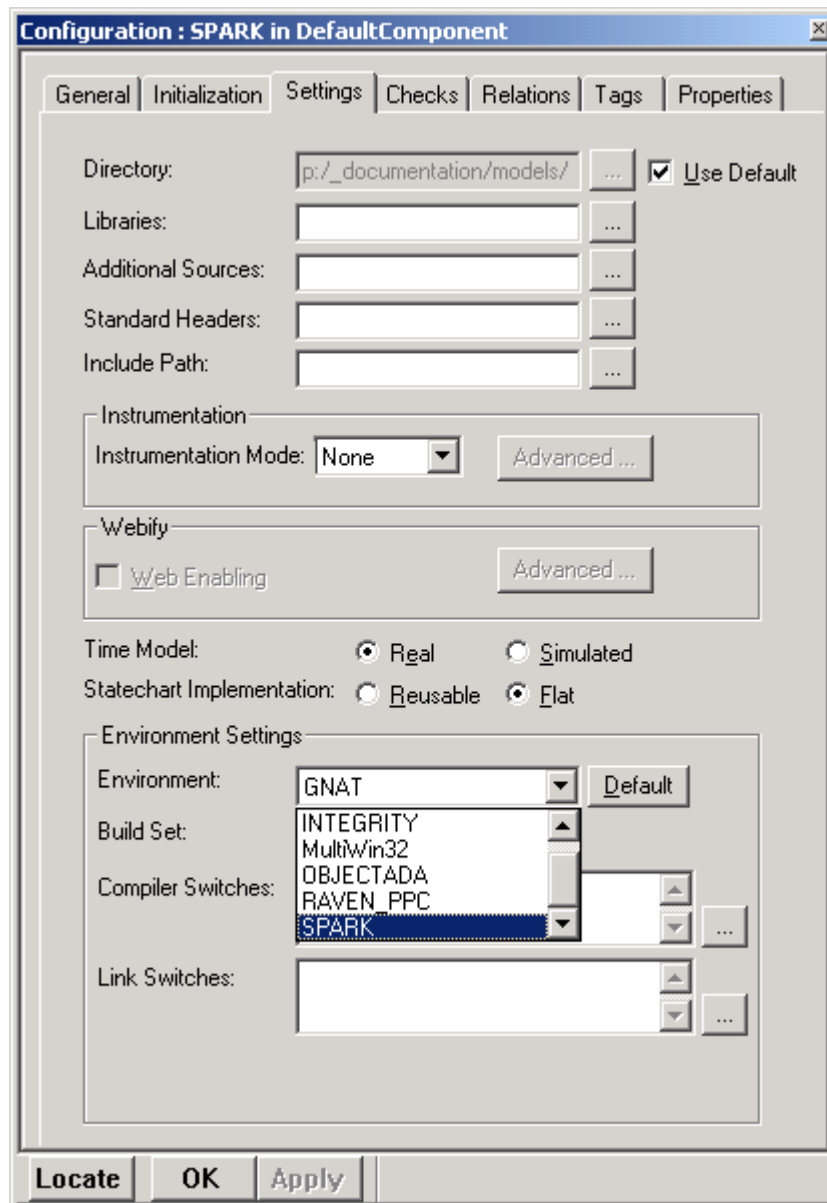
5.1.2. Setting the SPARK environment

The motivation in generating SPARK annotations is to have them analyzed by the SPARK Examiner. This is why Rhapsody generates commands to pass on to the SPARK Examiner which will then analyse the generated SPARK code. To enable the generation of these commands on a configuration, you have to go through the following steps :

- Create or add a SPARK configuration



- On the features window of the configuration, go to the settings tab, and in the environment settings frame, select SPARK as the environment.



5.1.3. Examination level

You can select the required examination level for a class or package specification property by setting the following properties to the appropriate values:

- Class.SPARK.Class.ExaminerLevelBody
- Class.SPARK.Class.ExaminerLevelSpec
- Package.SPARK.Class.ExaminerLevelBody
- Package.SPARK.Class.ExaminerLevelSpec

The available values for these properties are described below :

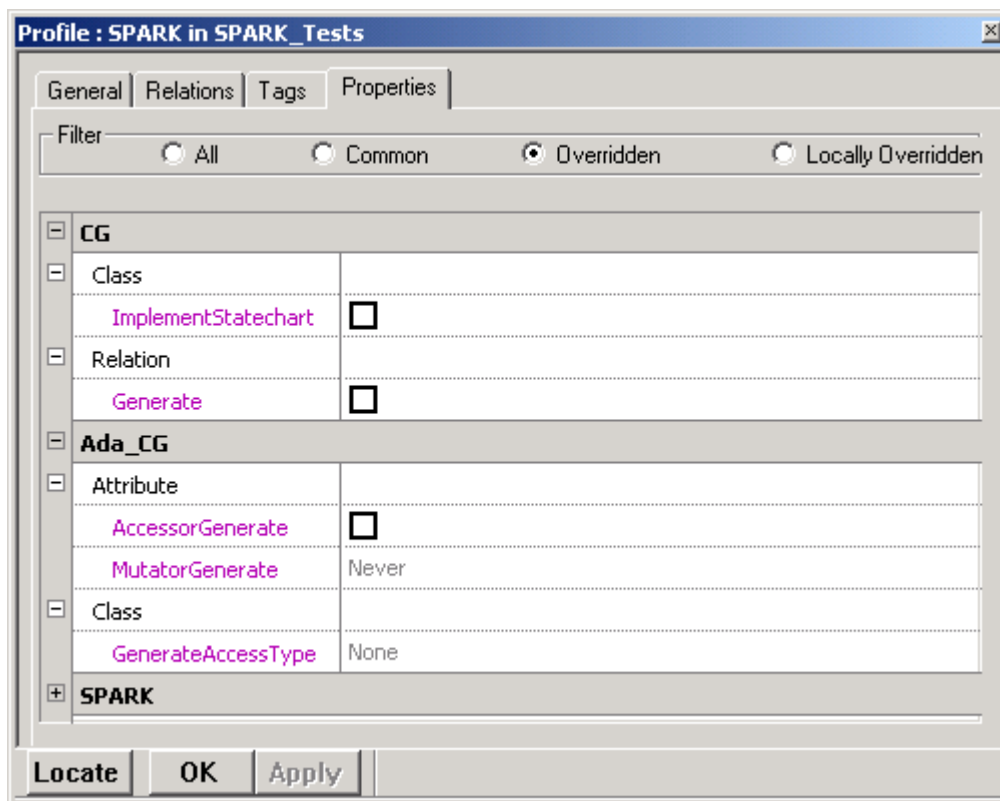
- None : the file will not be examined
- Data : data-flow analysis will be performed on the file
- Information : information-flow analysis will be performed on the file

5.2. Differences between code generated with and without the SPARK profile

Apart from the fact that the profile allows to model and generate SPARK annotations, there are a few differences between code generated using the profile and code generated without using the profile :

- Attribute accessors are not generated
- Relations accessors are not generated
- Statecharts code is not generated

This is achieved by overriding the appropriate properties in the profile.



5.3. General usage notes on SPARK profile tags

5.3.1. Capturing annotations with string tags

The SPARK profile is mostly based on the use of tags. Some of these tags are strings in which you can type annotations, which should be valid SPARK annotations, except that they shall not contain "--#" character sequences at the beginning of each line, as they will be generated by Rhapsody for you for each line in the tag.

5.3.2. Annotations often come in pairs

Very often, an annotation to be generated in the package specification may have a counterpart to be generated in the package body. Such annotations pairs are modeled by :

- tags which names end by a "Spec" and a "Body" suffix (such as DerivesBody and DerivesSpec tags for an operation).
- Dependencies from a client that have the same stereotype, but a different CG.Dependency.UsageType property value (either Implementation or Specification). An examples of such a dependency pair is the <<SPARK_Global>> dependency from an operation to an attribute.

5.3.3. Multiple modeling approaches

For most annotations, there are several ways to model and/or capture them using Rhapsody :

- The most common one is to use a string tag in which you type in the annotation content
- You might also use some stereotyped dependencies for most annotations
- More rarely, depending on the annotation kind, there might be some other ways yet (one illustration of this is described in the “Initializes annotations” section)

Depending on your preferences, you might favor one style over the other. These different approaches are generally not exclusive, meaning that part of an annotation might be modeled via the use of a string tag while another part is modeled through dependencies. However, we recommend avoiding mixing styles to facilitate model maintenance.

5.4. Inherit clauses

This section describes the various ways to model inherit annotations

5.4.1. Using inheritance

When a class A inherits from a class B, a with clause to B is generated in the specification of A. When generating code for SPARK, an inherit clause to B is also generated

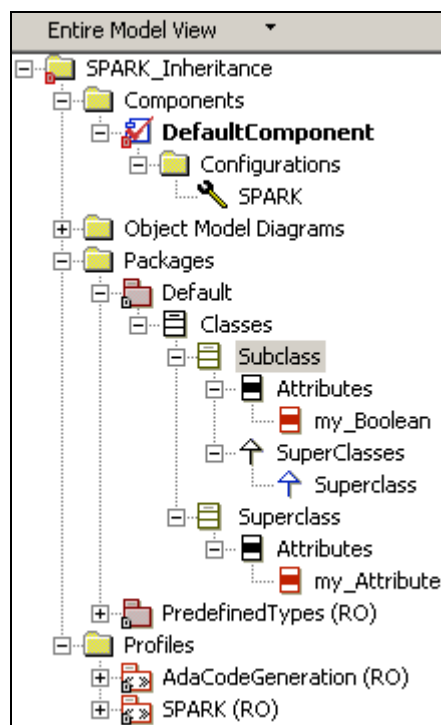


Figure 155: Modeling inherit clauses via inheritance

```

With Superclass;
--# inherit Superclass;

--++ class Subclass
package Subclass is

    type Subclass_t is new Superclass.Superclass_t with private;

    --Public Fields/Variables accessors -----

private

    type Subclass_t is new Superclass.Superclass_t with

    record

        -- Fields --
        my_Boolean : Boolean;    --++ attribute my_Boolean

    end record;

end Subclass;

```

Figure 156: Generated code for derived class using the SPARK profile

5.4.2. Using <<Usage>> dependencies

By default, every usage dependency will generate both a with clause (either in the client specification or body) and an inherit clause (always in the specification of the client) to the supplier of the dependency.

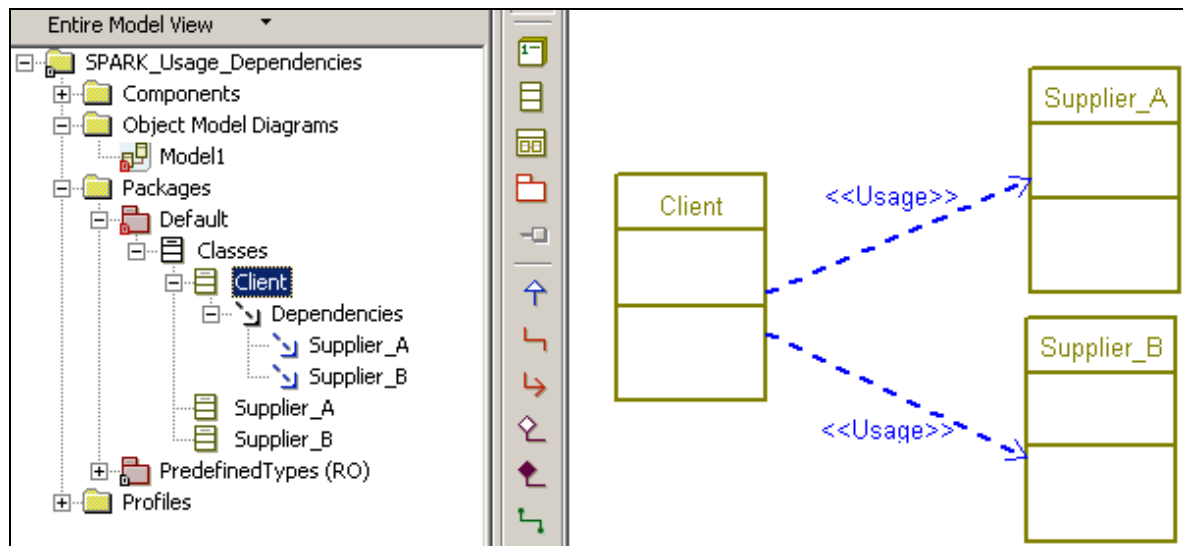


Figure 157: Modeling inherit clauses via <<Usage>> dependencies

```

With Supplier_A;
With Supplier_B;
--# inherit Supplier_A,
--#           Supplier_B;

--++ class Client
package Client is

    type Client_t is tagged private;

private

    type Client_t is tagged null record;

end Client;

```

Figure 158: Generated code for dependency client class using the SPARK profile

To turn off the generation of the inherit clause for usage dependency, uncheck the inherits tag on the dependency

There are cases where an inherit clause is required but not a with clause. In such cases, you can turn off the generation of the with clause for <<Usage>> dependencies by setting the Ada_CG.Dependency.GenerateWithClause property to false.

5.4.3. Using the inherits tag on a class or a package

You also have the option to type in the list of packages to be inherited using the inherits tag available for packages and classes.

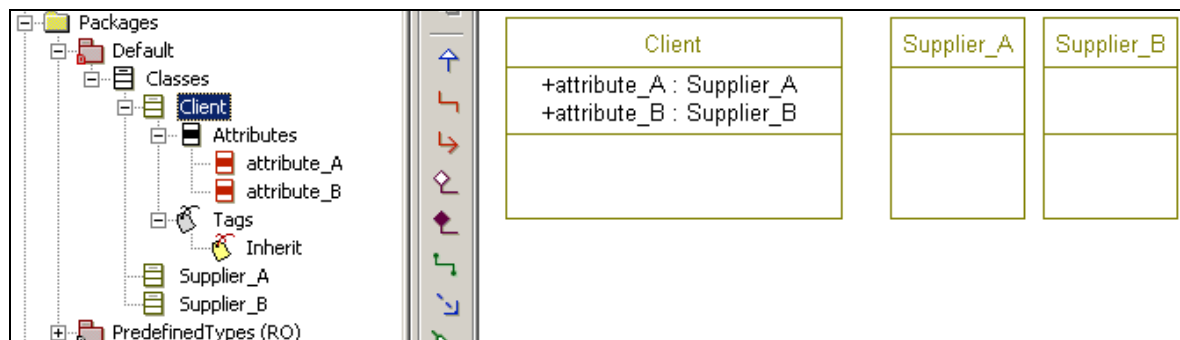


Figure 159: Modeling inherit clauses via Inherit tag

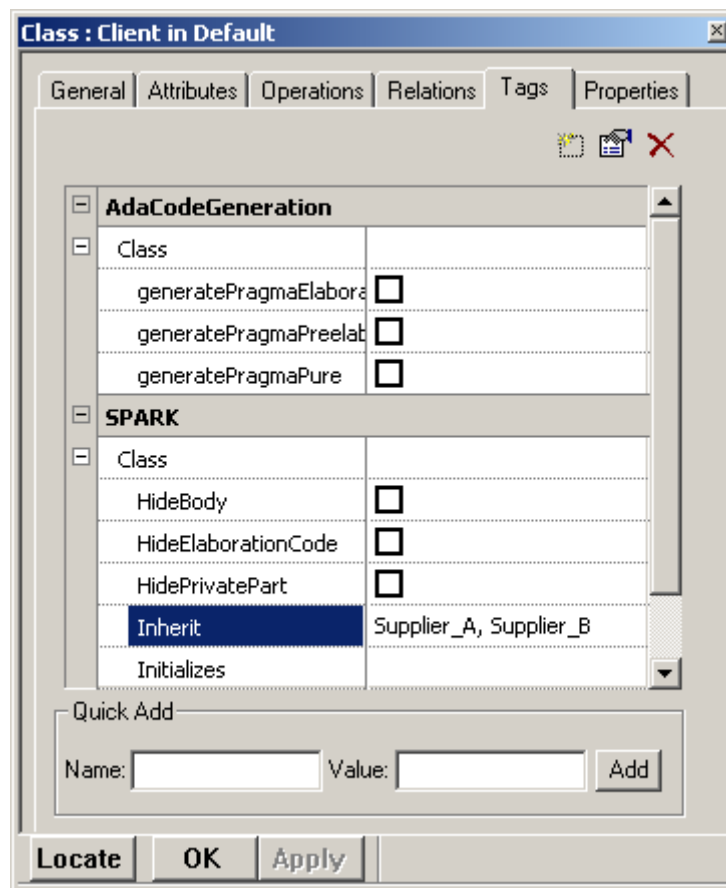


Figure 160: Setting the inherit tag on a class

```

With Supplier_A;
With Supplier_B;
--# inherit Supplier_A, Supplier_B;

--++ class Client
package Client is

    type Client_t is tagged

    record

        -- Fields --
        attribute_A : Supplier_A.Supplier_A_t;  --++ attribute attribute_A
        attribute_B : Supplier_B.Supplier_B_t;  --++ attribute attribute_B

    end record;

private

end Client;

```

Figure 161: Generated code for inherit tag on a class using the SPARK profile

5.5. Own variables

This section describes the various ways to model own annotations.

5.5.1. Modeling through tags on attributes

The table below summarizes the roles of the various SPARK profile tags related to own variables.

Note that these tags do not affect non-static attributes of classes.

Tag name	Description
IsAbstract	<p>SPARK own variables may designate actual Ada variables, or may have no concrete Ada counterpart. When they fall in this last category, they are called abstract own variables.</p> <p>Use this tag to prevent the generation of an Ada variable for an abstract SPARK own variable.</p>
IsInitialized	<p>Setting this tag to true will add the name of this own variable to the list of initialized variables in the "initializes" SPARK annotation for the class or package it is defined in.</p>
OwnKind	<p>In SPARK, a static attribute falls into one of these 3 categories :</p> <ul style="list-style-type: none"> • Own: the attribute is in the own annotation of the specification for this package or class • TypelessOwn: same as Own, but there is no type associated to it in the annotation • RefinementConstituent: the attribute is not in the own annotation for the specification of this package or class, but it should be part of a refinement definition in the own annotation for the body • None: the attribute is not in the own annotation for the specification of this package or class, and it should not be part of a refinement definition
OwnMode	<p>Use this tag to control the mode for an own variable :</p> <ul style="list-style-type: none"> • None : the own variable will not have an associated mode • In : the own variable will have a mode set to "in" • Out : the own variable will have a mode set to "out"
OwnRefinement	<p>The purpose of this tag is to hold the refinement definition for this own variable.</p> <p>This refinement is generated in the package body.</p>

Table 7 Own variables related tags

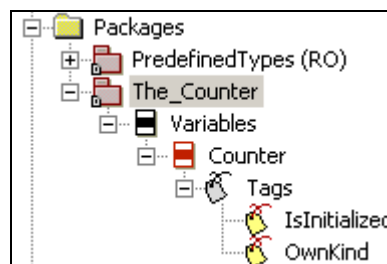


Figure 162: Modeling an own annotation on a package via tags on an attribute

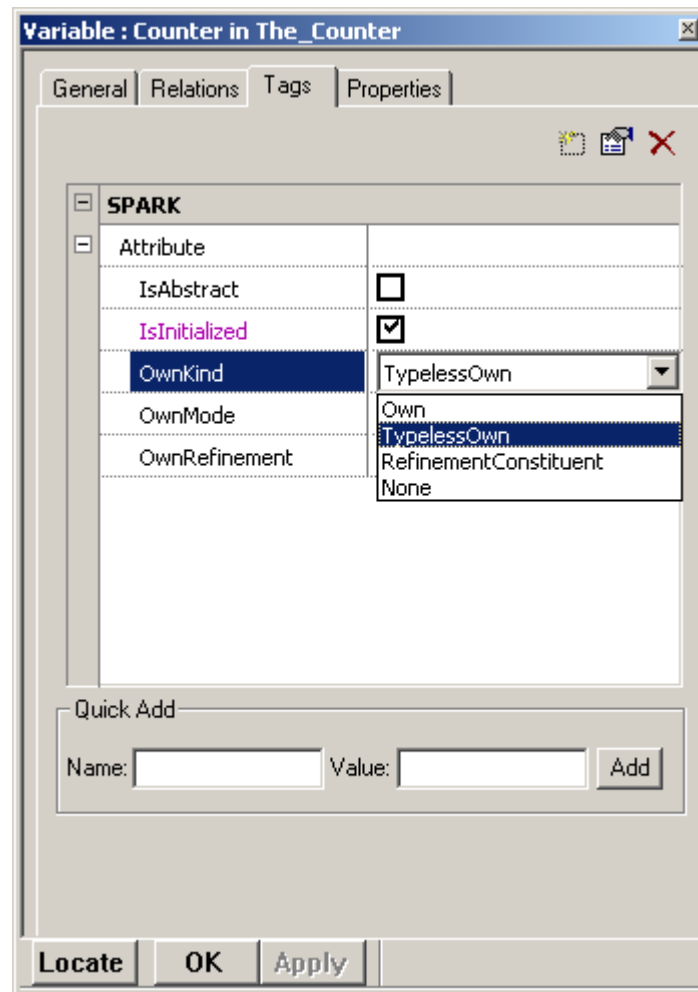


Figure 163: Setting some of the tags related to own variables on an attribute

Variable : Counter in The_Counter

General Relations Tags Properties

Name: Counter L

Stereotype: [dropdown]

Attribute type

☒ Use existing type

Type: Natural [dropdown] [icon]

Visibility

☒ Public ☐ Private

Multiplicity: 1 [dropdown] ☐ Ordered

☐ Constant ☐ Reference ☐ Static

Initial Value: 0 [input] [button]

Description: [text area] [button]

Locate OK Apply

Figure 164: Setting a default value on an initialized own variable attribute

```

--++ package The_Counter
package The_Counter
--# own Counter;
--# initializes Counter;
is

    -- Public Variables/Constants -----
    Counter : Natural := 0;  --++ attribute Counter

    --Public Fields/Variables accessors -----

private
end The_Counter;

```

Figure 165: Generated annotations for an initialized own variable

5.5.2. Using the OwnSpec and OwnBody tags

You also have the option to type in the list of own variables in the OwnSpec tag.

The OwnBody tag can be used to specify the own variables refinements.

Both tags are available for packages and classes.

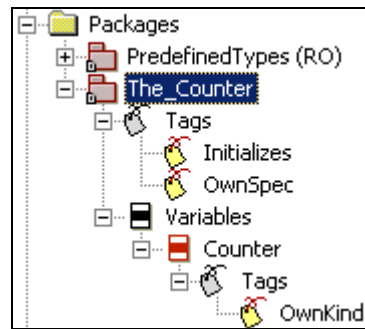


Figure 166: Modeling an own annotation on a package via tags on packages

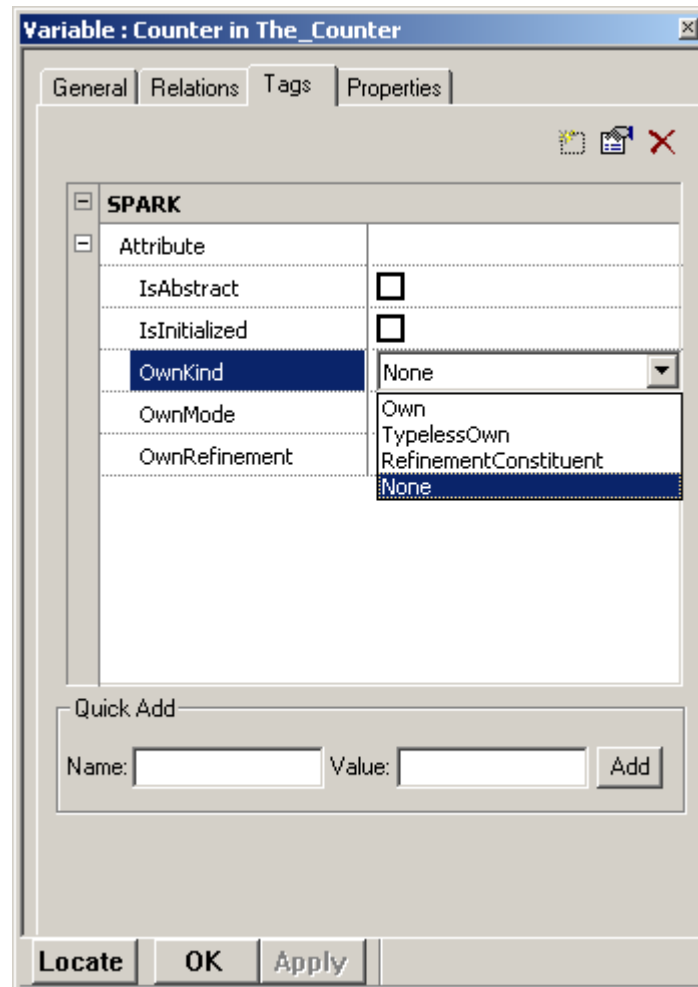


Figure 167: Disabling the generation of an own annotation at the attribute level

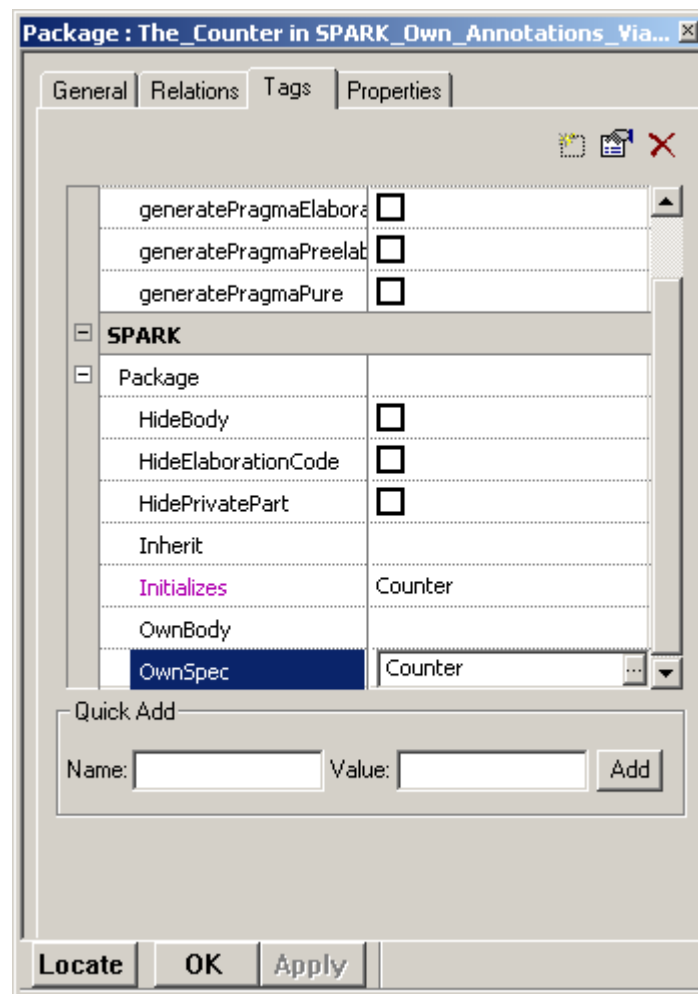


Figure 168: Setting some of the tags related to own variables on a package

Note that this example would generate the same code as the one shown in the previous section.

5.6. Initializes annotations

This section describes the various ways to model initializes annotations.

5.6.1. Using tags on attributes

As described in the section on own variables, one possible way to indicate that an own variable is initialized by a class or package is to use the `IsInitialized` tag on the own variable.

5.6.2. Using tags on class and package

As an alternative, and also described in the section on own variables, you might want to use the `initializes` tag available for classes and packages to type in the list of owned variables initialized by this class or package.

5.6.3. Using `<<SPARK_Initializes>>` dependencies

Yet another approach is to draw `<<SPARK_Initializes>>` dependencies from the class or package to the attributes to be initialized.

5.7. Proof types and Proof functions annotations

This section describes how to model proof types and proof functions annotations.

To indicate that a given type or function is to be generated as a proof type or as a proof function, apply the `<<SPARK_Proof>>` stereotype to it.

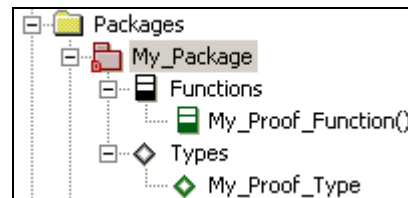


Figure 169: Modeling a package with a proof type and a proof function

The dialog box is titled "Primitive Operation : My_Proof_Function in My_Package". It has five tabs: "General", "Implementation", "Relations", "Tags", and "Properties". The "General" tab is selected.

Name: My_Proof_Function

Stereotype: SPARK_Proof

Visibility: Public

Returns:

- ☒ Use existing type
- Type:** void

Modifiers:

- ☐ Static
- ☐ Template

Arguments:

Name	Type	Value	Direction
<New>			

Description:

Locate OK Apply

Figure 170: Setting the stereotype of a function to <<SPARK_Proof>>

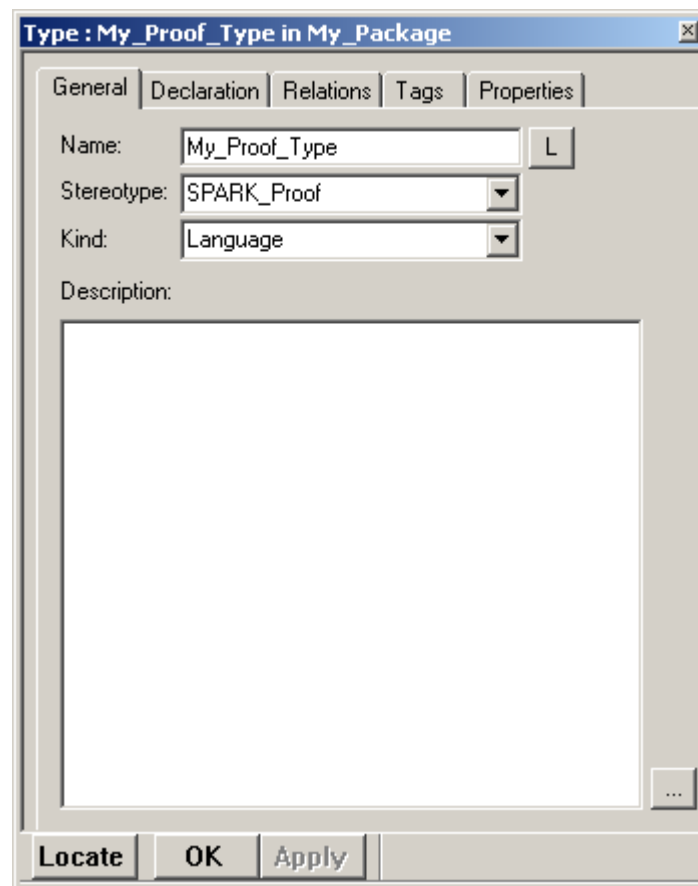


Figure 171: Setting the stereotype of a type to <<SPARK_Proof>>

```
--++ package My_Package
package My_Package is

    --Public types -----
    --# type My_Proof_Type is abstract;

    --Public Functions/Procedures section -----
    --++ operation My_Proof_Function()
    --# procedure My_Proof_Function;

private

end My_Package;
```

Figure 172: Generated code for package with proof type and proof function

5.8. Global annotations

This section describes the various ways to model global annotations.

5.8.1. Using <<SPARK_Global>> dependencies

You can model the global annotations for an operation using <<SPARK_Global>> dependencies from the operation to the global variables it is using.

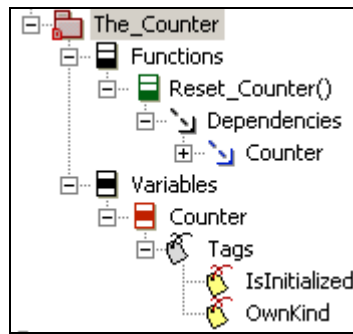


Figure 173: Modeling global annotations via dependencies from operation to attribute

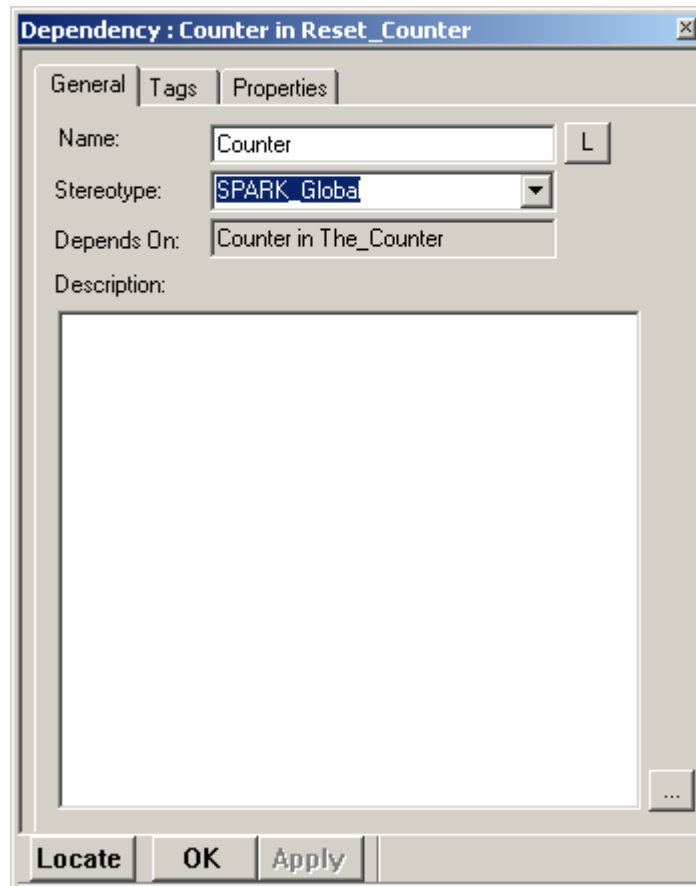


Figure 174: Setting the stereotype of a dependency to <<SPARK_Global>>

On such dependencies, a GlobalMode tag is available so that you can specify in which mode the variable is being used by the operation. The possible values are :

- In
- Out
- In_Out

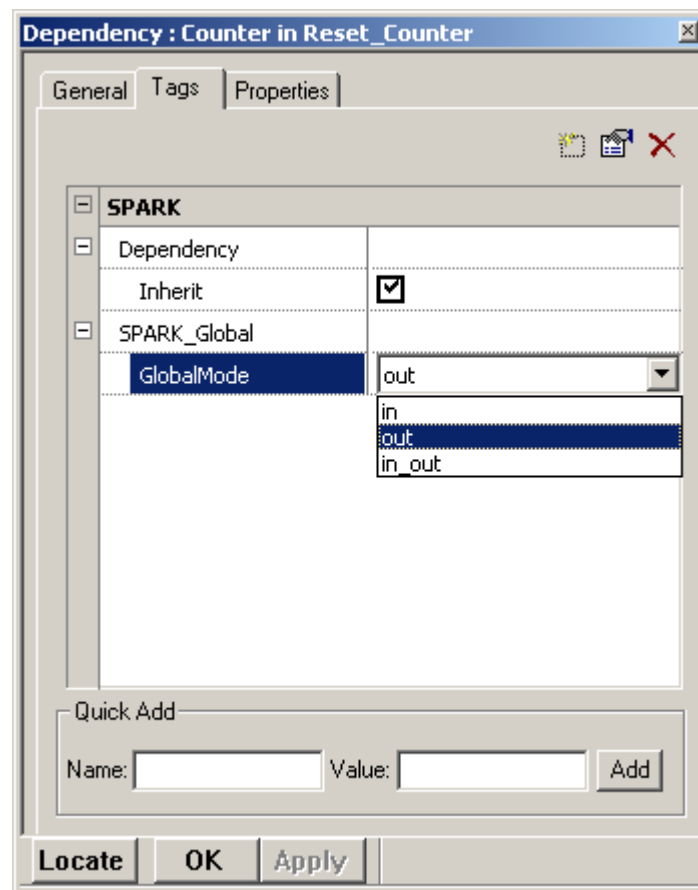


Figure 175: Setting the mode of a <<SPARK_Global>> dependency

The CG.Dependency.UsageType property on the dependencies determines whether the global variable will be added to the global annotation in the specification or in the body.

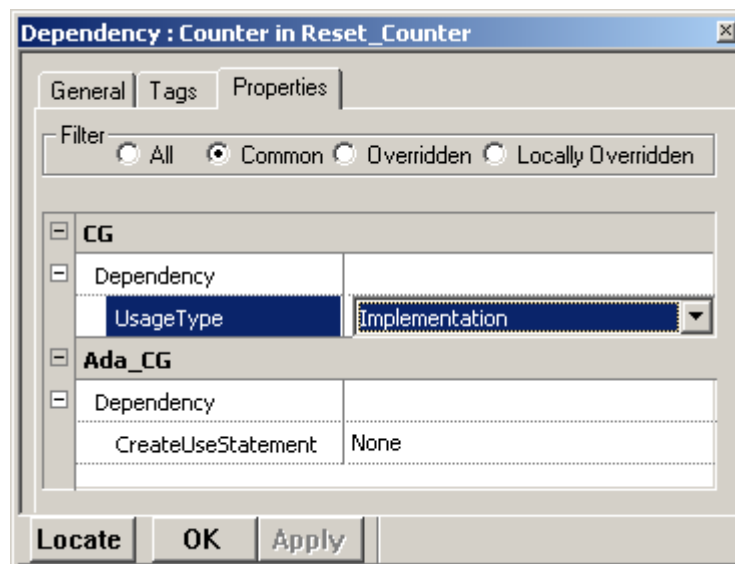


Figure 176: Controlling where the annotation is generated for a <<SPARK_Global>> dependency

```

--++ package The_Counter
package The_Counter
--# own Counter;
--# initializes Counter;
is

    -- Public Variables/Constants -----
    Counter : Natural := 0;    --++ attribute Counter

    --Public Functions/Procedures section -----
    --++ operation Reset_Counter()
    procedure Reset_Counter;

    --Public Fields/Variables accessors -----

private

end The_Counter;

```

Figure 177: Specification for a package with a <<SPARK_Global>> dependency from an operation to a package

```

--++ package The_Counter
package body The_Counter is

    --Functions/Procedures section -----
    procedure Reset_Counter
    --# global    out Counter;
    is
    begin
        --+[ operation Reset_Counter()
        Counter := 0;
        --+]
    end Reset_Counter;

    --Fields/Variables accessors -----

end The_Counter;

```

Figure 178: Implementation for a package with a <<SPARK_Global>> dependency from an operation to a package

5.8.2. Using tags on operations

An alternative is to use the GlobalSpec and GlobalBody tags on operations to type in the list of global variables and their modes.

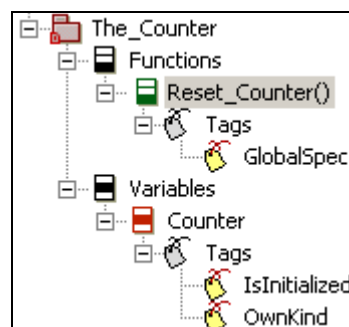


Figure 179: Modeling global annotations via GlobalSpec tag

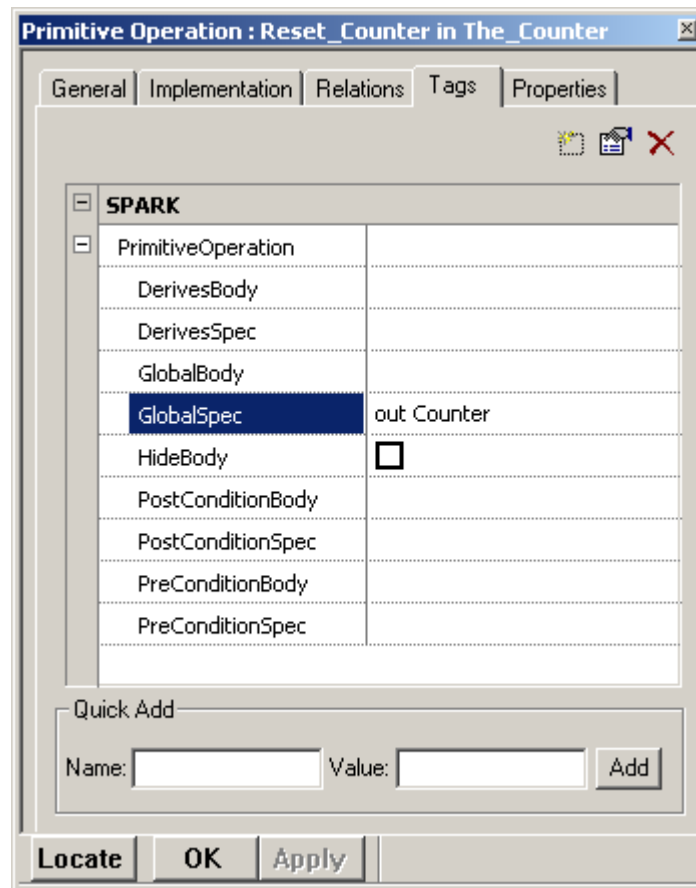


Figure 180: Setting the GlobalSpec tag on an operation

```

--++ package The_Counter
package The_Counter
--# own Counter;
--# initializes Counter;
is

    -- Public Variables/Constants -----
    Counter : Natural := 0;    --++ attribute Counter

    --Public Functions/Procedures section -----
    --++ operation Reset_Counter()
    procedure Reset_Counter;
    --# global out Counter;

    --Public Fields/Variables accessors -----

private
end The_Counter;

```

Figure 181: Specification for a package with an operation with a GlobalSpec tag

```

--++ package The_Counter
package body The_Counter is

  --Functions/Procedures section -----
  procedure Reset_Counter is
  begin
    --+[ operation Reset_Counter()
    Counter := 0;
    --+}
  end Reset_Counter;

  --Fields/Variables accessors -----

end The_Counter;

```

Figure 182: Implementation for a package with an operation with a GlobalSpec tag

Note that except for the location of the global annotation, which is here set to be generated in the specification, the generated code is similar to the one shown in the previous section.

5.9. Derives annotation

This section describes how to model derives annotations

You can model the derives annotation using the DerivesSpec and DerivesBody tags on an operation.

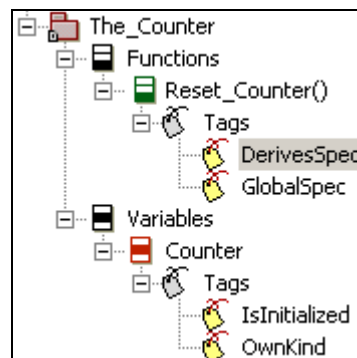


Figure 183: Modeling derives annotations via DerivesSpec tag

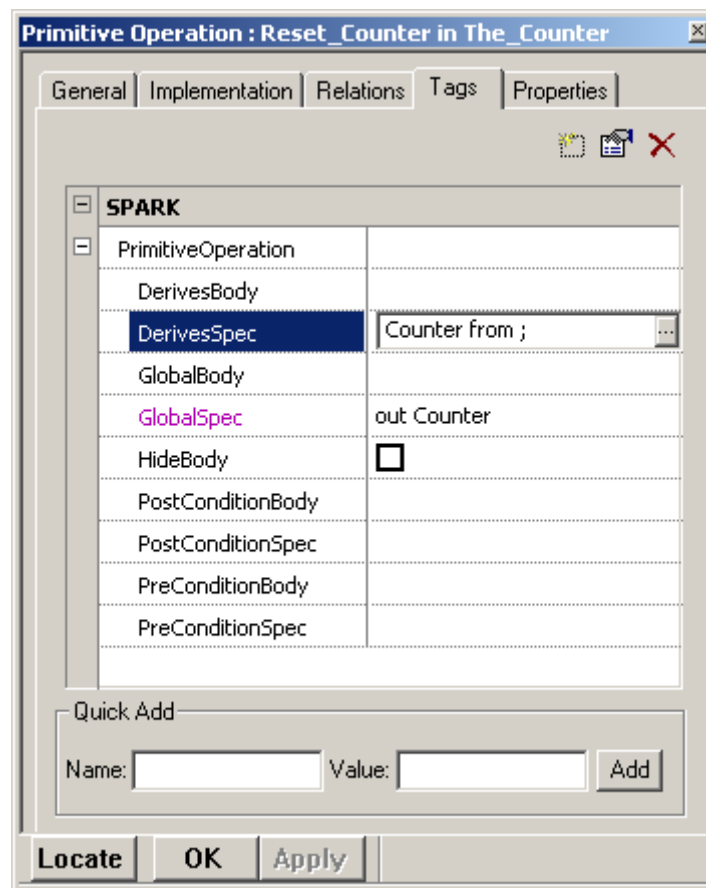


Figure 184: Setting the DerivesSpec tag on an operation

```

--++ package The_Counter
package The_Counter
--# own Counter;
--# initializes Counter;
is

    --Public Functions/Procedures section -----
    --++ operation Reset_Counter()
    procedure Reset_Counter;
    --# global out Counter;
    --# derives Counter from ;

    --Public Fields/Variables accessors -----

private

end The_Counter;

```

Figure 185: Specification for a package with an operation with a DerivesSpec tag

5.10. Precondition, postcondition and return Annotations

This section describes how to model precondition, postcondition and return annotations

You can model the precondition, postcondition and return annotations using the following tags on operations

- PreConditionSpec
- PreConditionBody

- PostConditionSpec
- PostConditionBody

Note that return annotations are modeled using the PostConditionSpec and PostConditionBody tags.

The example below, which is an extension of the counter example used in some of the other sections on SPARK, illustrates how a postcondition tag can be used.

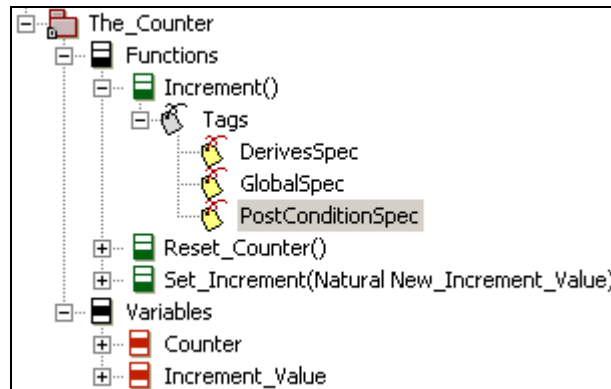


Figure 186: Modeling post conditions annotations via PostConditionSpec tag

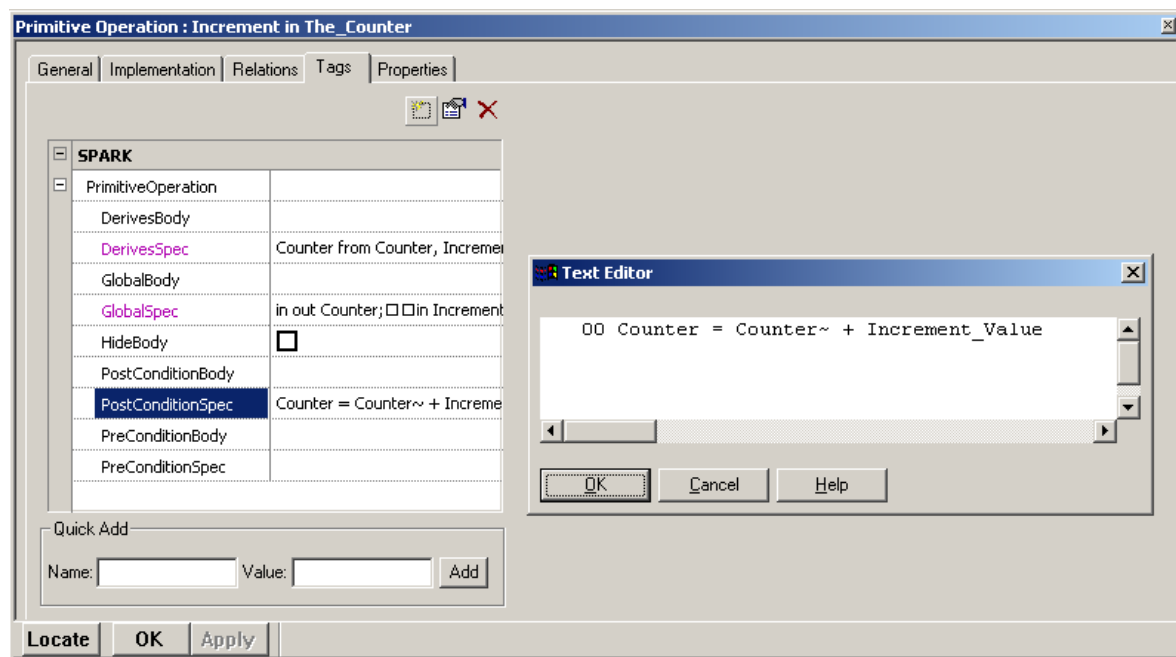


Figure 187: Setting the PostConditionSpec tag on an operation

```

--++ package The_Counter
package The_Counter
--# own Counter : Natural,
--#   Increment_Value : Natural;
--# initializes Counter,
--#   Increment_Value;
is

    --Public Functions/Procedures section -----
    --++ operation Increment()
    procedure Increment;
    --# global in out Counter;
    --#   in Increment_Value;
    --# derives Counter from Counter, Increment_Value;
    --# post Counter = Counter~ + Increment_Value;

    --++ operation Reset_Counter()
    procedure Reset_Counter;
    --# global out Counter;
    --# derives Counter from ;

    --++ operation Set_Increment(Natural)
    procedure Set_Increment (
        New_Increment_Value : in Natural
    );
    --# global out Increment_Value;
    --# derives Increment_Value from New_Increment_Value;

    --Public Fields/Variables accessors -----

private

end The_Counter;

```

Figure 188: Specification for a package with an operation with a PostConditionSpec tag

```

--++ package The_Counter
package body The_Counter is

    -- Body Variables/Constants -----
    Counter : Natural := 0;    --++ attribute Counter

    Increment_Value : Natural := 0;    --++ attribute Increment_Value

    --Functions/Procedures section -----
    procedure Increment is
    begin
        --+[ operation Increment()
        Counter := Counter + Increment_Value;
        --+]
    end Increment;

    procedure Reset_Counter is
    begin
        --+[ operation Reset_Counter()
        Counter := 0;
        --+]
    end Reset_Counter;

    procedure Set_Increment (
        New_Increment_Value : in Natural
    ) is
    begin
        --+[ operation Set_Increment(Natural)
        Increment_Value := New_Increment_Value;
        --+]
    end Set_Increment;

    --Fields/Variables accessors -----

end The_Counter;

```

Figure 189: Implementation for a package with an operation with a PostConditionSpec tag

5.11. Hide annotation

This section describes how to model hide annotations

5.11.1. On a class or a package

There are 3 tags controlling the various hide annotations that can appear in the generated code for a class or a package

- HideBody : controls the generation of a hide annotation in the body of the generated Ada package
- HideElaborationCode : controls the generation of a hide annotation in the initialization code of the generated Ada package
- HidePrivatePart : controls the generation of a hide annotation in the private part of the generated Ada package

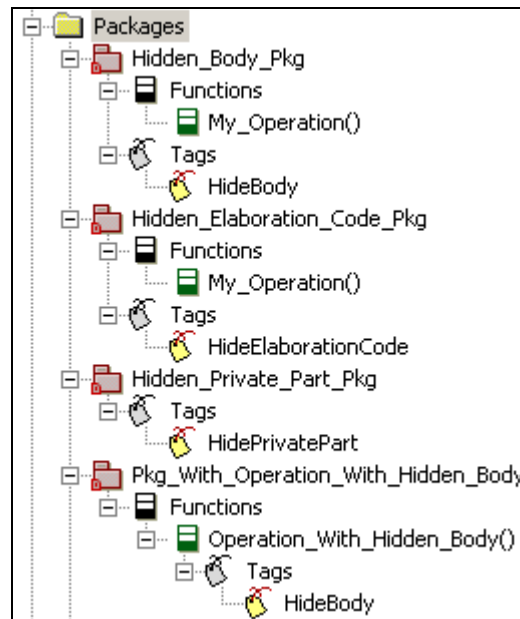


Figure 190: Modeling hide annotations on packages and operations

In the figure above, every tag is a Boolean and has its value set to True.

```
--++ package Hidden_Body_Pkg
package body Hidden_Body_Pkg is
--# hide Hidden_Body_Pkg

--Functions/Procedures section -----
procedure My_Operation is
begin
  null;
  --+[ operation My_Operation()

  --+]
end My_Operation;
end Hidden_Body_Pkg;
```

Figure 191: Generated body code for a package body with its HideBody tag set to true

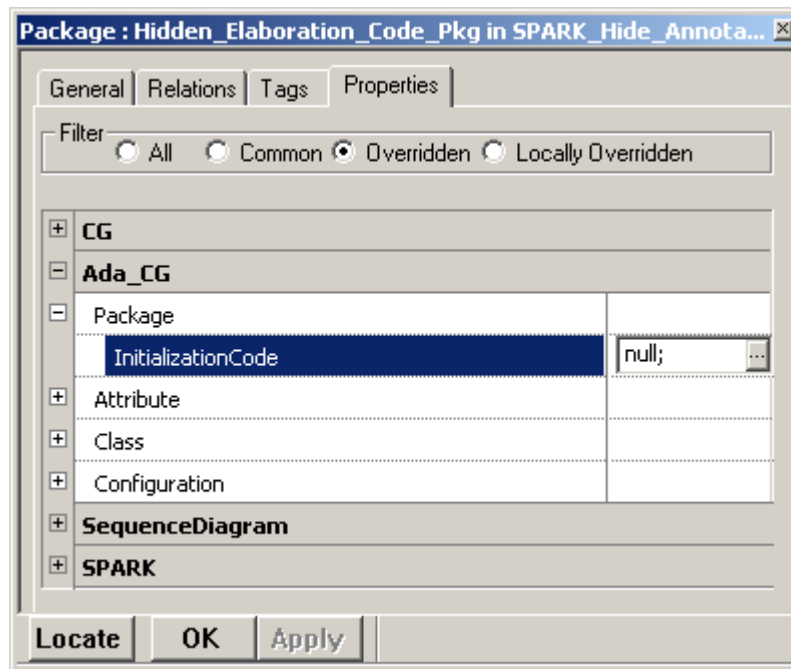


Figure 192: Setting the elaboration code on a package

```
--++ package Hidden_Elaboration_Code_Pkg
package body Hidden_Elaboration_Code_Pkg is

  --Functions/Procedures section -----
  procedure My_Operation is
  begin
    null;
    --+[ operation My_Operation()

    --+]
  end My_Operation;

begin
  --# hide
  null;
end Hidden_Elaboration_Code_Pkg;
```

Figure 193: Generated code for a package body with elaboration code and its HideElaborationCode tag set to true

```
--++ package Hidden_Private_Part_Pkg
package Hidden_Private_Part_Pkg is

  private
  --# hide Hidden_Private_Part_Pkg

end Hidden_Private_Part_Pkg;
```

Figure 194: Generated code for a package specification with its HidePrivatePart tag set to true

```
--++ package Pkg_With_Operation_With_Hidden_Body
package body Pkg_With_Operation_With_Hidden_Body is

    --Functions/Procedures section -----
    procedure Operation_With_Hidden_Body is
    begin
        --# hide Operation_With_Hidden_Body
        null;
        --+[] operation Operation_With_Hidden_Body()

        --+[]
    end Operation_With_Hidden_Body;

end Pkg_With_Operation_With_Hidden_Body;
```

Figure 195: Generated code for an operation body with its HideBody tag set to true

5.11.2. On an operation

The HideBody tag on operations controls the generation of a hide annotation in the body of an operation.

5.12. Main program annotation

This section describes how to model the main program annotation.

The main_program annotation is automatically generated for <<entrypoint>> classes.

6. Behavioral Code Generation

This section uses the following definitions:

Reactive class	A class that consumes messages, typically defined by statecharts or activity diagrams
Reactive instance	An instance of a reactive class.
Active class	A class that dispatches events (e.g. manages an event loop) on its own OS task.
Active instance	An instance of an active class.
Event	An asynchronous message, with or without data.
Triggered Operation	A synchronous message, with or without data.

6.1. Overview of the behavioral frameworks

6.1.1. Selecting the behavioral framework implementation

It is possible to select between three versions of the behavioral framework, one relies on Ada83 constructs exclusively, and the others exploit Ada95 constructs.

The new Ada 95 based framework is the default for new models. The component property "Ada_CG.Component.UseAdaFramework" is used to switch between the frameworks. The value "NewFWK95" is used to select the ravenstar compatible framework. "FWK95" is used for backward compatibility for models before version 7.3. And, "FWK83" is set to use the Ada 83 Framework instead.

Note that it is not possible to mix the three frameworks in a same component.

6.1.2. Differences between the Ada 83 and the Ada 95 implementations

- Features available exclusively with the two 95 Frameworks
 - Tasks are only used for active classes and not for locking resources (protected objects are used for this).
 - Dynamic memory allocation (DMA) is only used for asynchronous events (with the Ada 83 Framework, triggered operations are using DMA by default, unless the "Ada_CG.Class.OptimizeStatechartsWithoutEventsMemoryAllocation" property is set).
 - Statechart Inheritance
 - Generation of code for SendAction states
 - Deep History Connectors
 - Active classes no longer need <<active_context>> dependencies,
 - Indirect reactive parts (that is a reactive composite class having direct parts that are not reactive but have reactive parts themselves)
 - Out transitions with same trigger but different guards on a same state (with the Ada 83 Framework, this requires using a condition connector)

- Features available exclusively with the 83 Framework
 - Generic reactive classes

6.1.3. Common features of both frameworks

Behavior is modeled in IBM® Rational® Rhapsody® Developer for Ada by creating a statechart for a class. This class is then said to be a *reactive* class. Within the family of reactive classes, there are 2 distinct types: classes with asynchronous messaging using events, and those with synchronous messaging that use only triggered operations.

6.1.1.3 Reactive classes

Four operations exist to start or stop the reactive class statechart, and to monitor its status. They have the same interface regardless of the framework version.

```

-- Activate the state machine, and take the default transitions.
procedure start_behavior (this : in out reactive_class_t; success : out Boolean);

-- Set the ria_behavior_terminated attribute to true
procedure terminate_behavior (this : in out reactive_class_t);

-- Returns the ria_behavior_started attribute
function is_behavior_started (this : in reactive_class_t) return Boolean;

-- Returns the ria_behavior_terminated attribute
function is_behavior_terminated (this : in reactive_class_t) return Boolean;

```

Figure 196: Operations to Control the Reactive Class Statechart.

6.1.2.3 Active classes

For a class to be considered an active class, the concurrency of the class must be set to “active”.

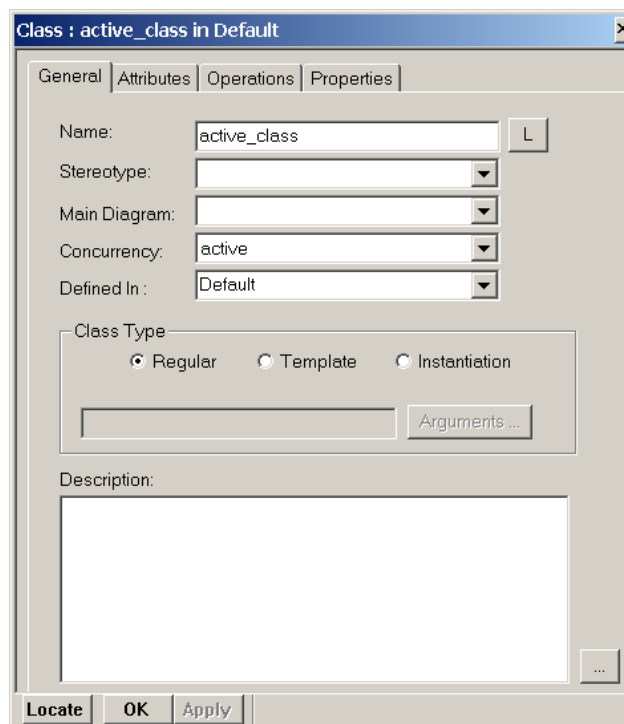


Figure 197: Definition of an Active Class.

An active class has a start operation which begins the event loop for the process of events.

```

procedure Start (this : in out Active_Class_t);

```

Figure 198: Operations to Control the Active Class

6.2. Using the Ada 83 Behavioral framework

6.2.1. Limitations

- Indirect reactive parts are not supported (that is a reactive composite class having direct parts that are not reactive but have reactive parts themselves)
- Statechart inheritance is not supported
- Deep history connectors are not supported
- SendAction states are not supported

- Out transitions with same trigger but different guards on a same state are not supported, a workaround is to use an intermediate condition connector instead.

6.2.2. Event-based reactive classes

Each event-based reactive class must have an *active* class that performs the event loop and delivers events to it. Each active class creates its own task to handle its event loop, and is linked to its reactive classes by a dependency stereotyped “Active Context”.

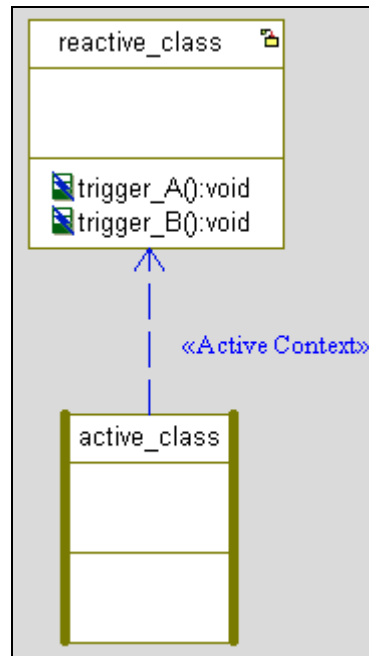


Figure 199: A Reactive Class and its Active Class.

6.2.3. Reactive Class Generation

Each reactive class automatically generates:

- Its own event type which contains a variant record for various event reception data.
- Its own event queue type.
- gen_<event>() method for each type of event it may consume.
- Event creation/deletion methods.
- Event consumption methods.
- Specific statechart implementation methods.

Every reactive class will use the Ada packages found in the Rhapsody behavioral code for Ada 83 library. This library uses only Ada 83 constructs and is used whether generating Ada 95 or 83. A future release will include a library for Ada 95 which will incorporate some optimizations possible when using Ada 95.

The set of “With” statements for an event-based reactive class is shown below.

```

With RiA_Mutex_Ada83;      -- Provides the event queue guard
With RiA_Event_Flag_Ada83; -- Provides the event flag
With RiA_Guarded_Queue_Ada83; -- Provides the queue for the events to process
With RiA_Types;           -- Provides the types used in the behavior code library
  
```

Figure 200: The "With" Clauses for a Reactive Class.

6.2.1.3 Reactive Class Variables

A reactive class communicates with its active class using shared variables between the two instances. These variables are generated in the class record.

```

type reactive_class_t is tagged
record
  ria_behavior_started : Boolean := false;
  ria_behavior_terminated : Boolean := false;
  ria_null_transitions_count : Integer := 0;
  ria_state_machine_busy : Boolean := false;
  ria_current_event : Event;
  ria_events_available_signal : RiA_Event_Flag_Ada83.RiA_Event_Flag_Ada83_acc_t := null;
  ria_queue_guard : RiA_Mutex_Ada83.RiA_Mutex_Ada83_acc_t := null;
  ria_event_queue : Event_Queue_t;
  ria_context_ready : RiA_Types.Boolean_acc := null;
  ria_context_queue : Reactive_Instances_Queue_acc := null;
  root_state_active : Integer := RiA_Non_State;
  root_state_sub_state : Integer := RiA_Non_State;
end record;

```

Figure 201: The Reactive Class Record.

The variables are used in the following manner:

ria_behavior_started: A boolean flag that indicates that the reactive class is ready to consume events.

ria_behavior_terminated: A boolean flag that indicate that the reactive class reached a terminate connector.

ria_null_transitions_count: Count the number of null-transitions that are yet to be taken after an event is consumed.

ria_state_machine_busy: An indicator that the state machine is currently consuming an event, used to prevent self calls of triggered operations while the state machine is in an undefined state. This flag is not used for mutual exclusion.

ria_current_event: A pointer to the currently consumed event.

ria_events_available_signal: A pointer to the active instance event flag.

ria_queue_guard: The event queue guard, specified by the active class.

ria_event_queue: The reactive instance event queue.

ria_context_ready: A pointer to a boolean flag, used for event consumption optimization.

ria_context_queue: A pointer to a Reactive_Instances_Queue.

root_state_active: An integer that holds the current active state, of the root state.

root_state_sub_state: An integer that holds the current active sub-state of the root state.

There is also one static variable that is generated:

ria_maximum_allowed_null_steps: A static integer that indicated the maximum allowed null steps (used to detect infinite null-transitions loop), can be disabled by setting to 0.

6.2.2.3 Reactive Class Public Operations

In addition, there are several public operations generated for the Ada package that represents a reactive class.

There are 2 operations to retrieve information about the current event being processed.

```

-- Returns the current event data record
function current_event_data (this : in reactive_class_t) return Event_Data_acc;

-- Returns the current event id
function current_event_id (this : in reactive_class_t) return Integer;

```

Figure 202: Operations for the Current Event Information.

These operations use the public types that are defined for a reactive class and will be discussed in the next section.

As well as defining the new operations discussed above, reactive classes also require some initialization and finalization. These methods will be created if they do not already exist, or the necessary code will be added to existing implementations.

```

-- Initialization of the instance
procedure Initialize(this : in out reactive_class_t);

-- Finalization of the instance
procedure Finalize(this : in out reactive_class_t);

```

Figure 203: Initialization and Finalization of the Reactive Class.

6.2.3.3 Statechart-Specific Reactive Class Operations

In addition to the generic public operations discussed above for every reactive class, there are also specific operations and types created based on the contents of the statechart.

For this discussion, an example project will be used which contains a reactive class called “reactive_class” defined in the “Default” package. The reactive class has a simple statechart with 2 states (“state_1” and “state_2”) with 2 events being used to trigger the state transitions (“trigger_A”, and “trigger_B”). The event “trigger_B” has an Integer for its event data.

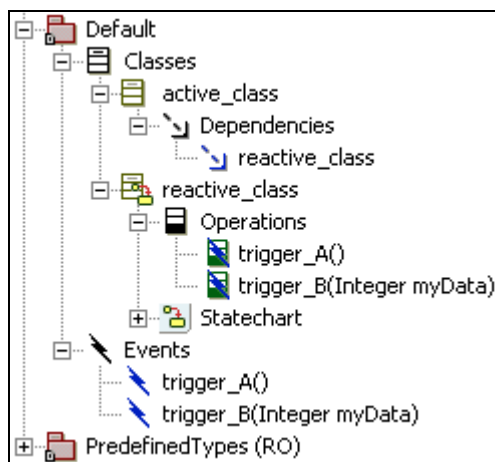


Figure 204: Example Reactive Class Project.

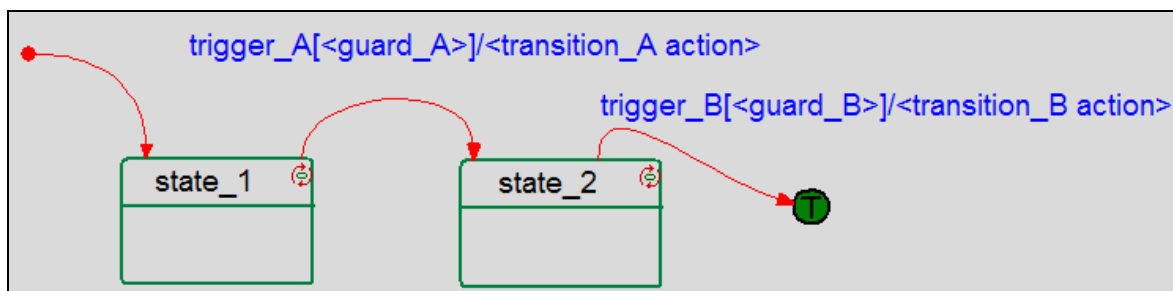


Figure 205: Statechart for the Reactive Class.

Given the above example project, the event types will be generated as follows:

```
-- A variant record that holds a pointer to the events consumed
-- by the class.
type Event_Data (event_id : integer) is
record
  case event_id is
    when Default.trigger_a_id =>
      null;
    when Default.trigger_b_id =>
      Default_trigger_B_myData : Integer;
    when others =>
      null;
  end case;
end record;

type Event_Data_acc is access Event_Data;

-- The reactive class event representation
type Event is
record
  id : Integer := 0;
  data : Event_Data_acc := null;
end record;

type Event_acc is access Event;
```

Figure 206: The Event Types for the Reactive Class.

The Event_Data variant record will either have no elements in the case of the event being “trigger_A”, or will have an Integer element for the “trigger_B” event.

To allow different events to share argument names, the record component corresponding to an event argument has a name based on the full namespace of the argument.

Shall the relative name of the argument be preferred instead, setting the Ada.CG.Class.RelativeEventDataRecordTypeComponentsNaming to true will disable this behavior. Note that if this property is set to true, there shall be no events or triggered operations sharing an argument name, as they would generate variant record components with the same name, which is uncompileable.

In case of triggered operations, the property shall be set on the reactive class whose statechart uses the triggered operations, and in case of events it shall be set on the events themselves (and not on the receptions).

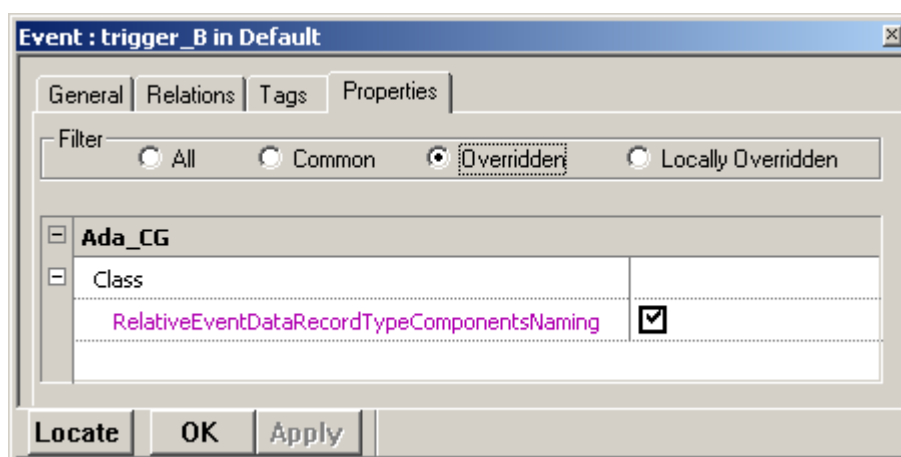


Figure 207: Using relative naming for current event data

```

type Event_Data (event_id : integer) is
record
  case event_id is
    when Default.trigger_a_id =>
      null;
    when Default.trigger_b_id =>
      myData : Integer;
    when others =>
      null;
  end case;
end record;

```

Figure 208: Event data record type using relative naming

As shown in the sample code above, the event is identified by an Integer defined on the containing package, in this case the “Default” package.

```

package Default is

  -- Identifier for trigger_A event
  trigger_a_id : constant Integer := 1;

  -- Identifier for trigger_B event
  trigger_b_id : constant Integer := 2;

end Default;

```

Figure 209: The Parent Package of the Reactive Class.

Beside the event types, there are public operations added to enable events to be sent to the reactive class. For each event, a “get_<event_name>” operation is created.

```

-- Generate the trigger_A event
procedure gen_trigger_a (this : in out reactive_class_t) ;

-- Generate the trigger_B event
procedure gen_trigger_b (this : in out reactive_class_t; myData : in Integer);

```

Figure 210: Operations to Generate Events for a Reactive Class.

The following statecharts show how one can access the current event data in a statechart so as to decide which transition shall be taken. Note that depending on the value of the Ada_CG.Class.RelativeEventDataRecordTypeComponentsNaming property, the guards on some transitions may have to be modified

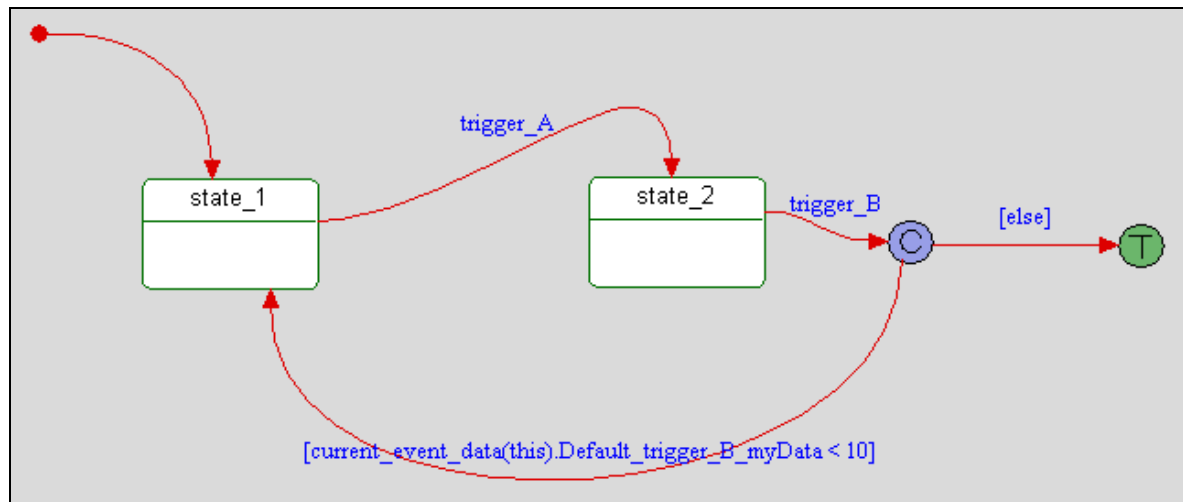


Figure 211: Accessing a trigger parameter value (using full namespace based naming)

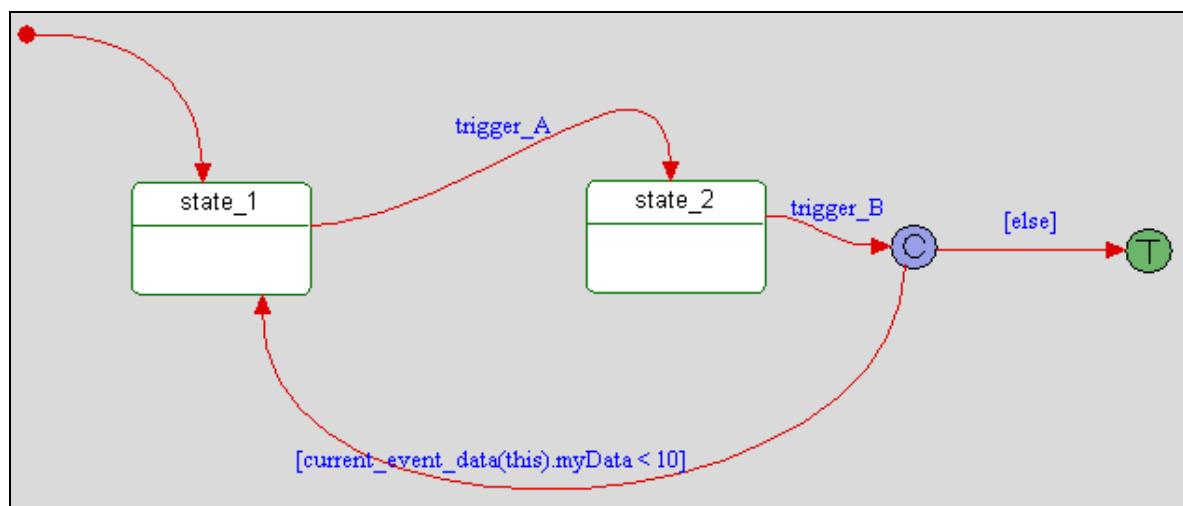


Figure 212: Accessing a trigger parameter value (using relative naming)

6.2.4. Active Class Generation

An active class must have a dependency to at least one reactive class stereotyped “Active Context”.

For the code to generate correctly, the active class must set its record type to be private. This is done by setting the “Ada_CG.Class.Visibility” property to “Private” for the active class.

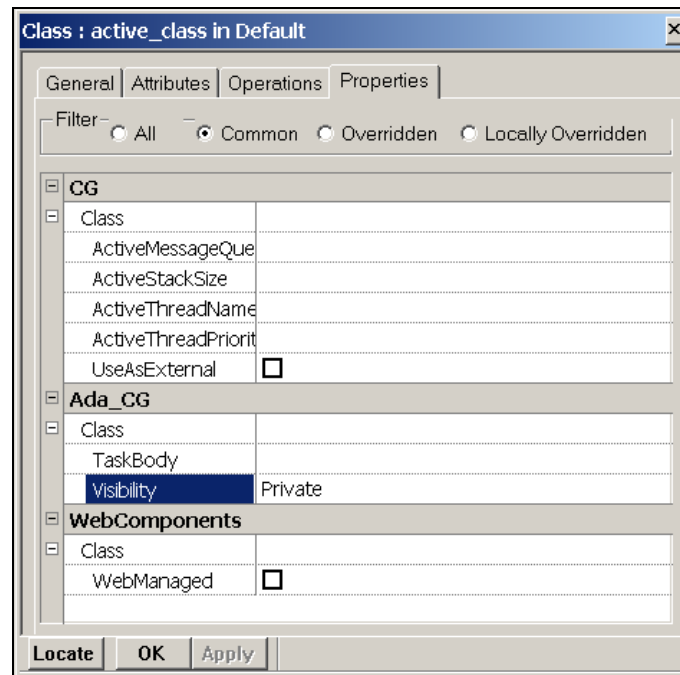


Figure 213: Setting the Record Type Visibility for an Active Class.

An active class generates an Ada task type to execute the event loop for the handling of events. There is one entry to the task type called “process_events”.

```
-- The task to handle the processing of events
task type RiA_Active_Task is
  entry process_events (this : in out active_class_t);
end RiA_Active_Task;

type RiA_Active_Task_acc is access RiA_Active_Task;
```

Figure 214: The Task Generated for an Active Class.

6.2.1.4 Active Class Variables

In addition, several elements are added to the record for the active class type.

```
record
  <reactive class name>_context_queue :
    <reactive class name>.Reactive_Instances_Queue_acc := null;
  <reactive class name>_context_ready : RiA_Types.Boolean_acc := null;
  ria_task : RiA_Active_Task_acc := null;
  ria_events_available_signal :
    RiA_Event_Flag_Ada83.RiA_Event_Flag_Ada83_acc_t := null;
  ria_queue_guard : RiA_Mutex_Ada83.RiA_Mutex_Ada83_acc_t := null;
end record;
```

Figure 215: The Record Definition for the Active Class.

ria_task: The instance task.

ria_events_available_signal: The task event flag, used to signal the active instance that there are events ready to be consumed on at least one of the reactive instances in its context.

ria_queue_guard: A mutex to protect the reactive instances queues.

For every reactive class that the active class has an <<Active Context>> dependency to, the following attributes are generated:

<reactive class name>_context_queue: A queue of reactive instances that is used for event dispatching.

<reactive class name>_context_ready: A boolean flag used for event dispatching optimizations.

The following “With” statements are needed in the active class as well to use the Rhapsody behavioral code for Ada 83 library.

```
With RiA_Mutex_Ada83;
With RiA_Event_Flag_Ada83;
With RiA_Types;
```

Figure 216: "With" Statements for an Active Class.

In addition, a “With” statement will be added for each of its reactive classes.

6.2.2.4 Public Operations for an Active Class

There are 2 kind of public operations of interest generated for an active class. One is the “start” operation, as mentioned earlier.

The other is the “register_context_<reactive class name>” series of procedures, which make the active instance the context for the event dispatching of the reactive instance. One of these methods will be created for each reactive class for which this class serves as its active context.

```
-- Register an instance of the equivalent reactive type on the active instance
procedure register_context_reactive_class (this : in out Active_Class_t;
    reactive : in out reactive_class.reactive_class_t);
```

Figure 217: The Public Operations of an Active Class.

An active class also requires some initialization and finalization. The Initialize and Finalize methods will be created if they do not already exist, or the necessary code will be inserted into existing implementations.

```
-- Initialization of the instance
procedure Initialize(this : in out reactive_class_t);

-- Finalization of the instance
procedure Finalize(this : in out reactive_class_t);
```

Figure 218: Initialization and Finalization of the Active Instance.

6.2.5. Working with Active and Reactive Classes

There are four distinct steps when using active and reactive classes: *Initializing, Starting the Behavior, Sending Events, and Finalizing.*

6.2.1.5 Initializing

Both the active and reactive instances need to be initialized before using them. This is accomplished by calling the “Initialize” procedure on each of them.

6.2.2.5 Starting the Behavior

To activate the state machine for a reactive class, the “start_behavior” procedure is called for the reactive instance. This will cause the default transitions to be taken, and will allow the reactive class to receive events.

The active class can be started by calling the “start” procedure which will activate the active instance event loop.

6.2.3.5 Sending Events

Once the active and reactive classes have been started, it is possible to send events to the reactive class. This is done by the calling the “gen_<event name>” procedure on the reactive instance.

6.2.4.5 Finalizing

To cleanup the resources of the reactive and active instances, the “Initialize” and “Finalize” procedures should be called.

The following procedure definition demonstrates these four phases for the example model given above.

```

procedure main is
  active_instance : active_class.active_class_acc_t;
  reactive_instance : reactive_class.reactive_class_acc_t;
  start_result : Boolean;
begin
  -- Create the new instances
  active_instance := new active_class.active_class_t;
  reactive_instance := new reactive_class.reactive_class_t;

  -- Initialize the instances
  active_class.Initialize(active_instance.all);
  reactive_class.Initialize(reactive_instance.all);

  -- Start the behavior
  reactive_class.start_behavior(reactive_instance.all, start_result);
  active_class.start(active_instance.all);

  -- Send events
  reactive_class.gen_trigger_A(reactive_instance.all);
  reactive_class.gen_trigger_B(reactive_instance.all, 1);

  -- Cleanup the instances
  active_class.Finalize(active_instance.all);
  reactive_class.Finalize(reactive_instance.all);
end main;

```

Figure 219: Using an Active and Reactive Class.

6.2.6. Active Reactive Class

A reactive instance can act as its own active context as well by setting its concurrency to “active” - this class is then called an *Active-Reactive* class. Instances of this type cannot have another active class act as its active context, and they cannot be the active context for any other reactive instances. When using an active-reactive class, an <<Active Context>> dependency to itself is not needed.

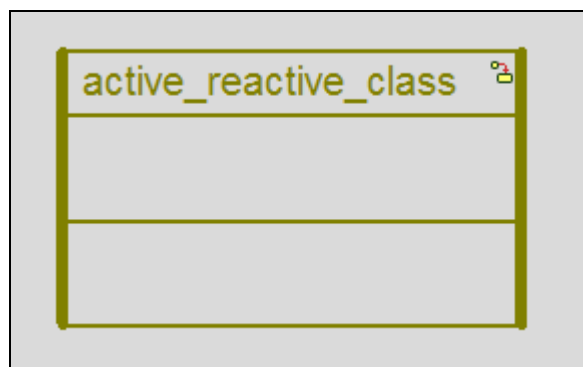


Figure 220: An Active-Reactive Class.

Using the active-reactive class is the same as using a separate active class. The instance needs to be initialized, the statechart needs to be started, the event loop needs to be started, events are sent, and the instance is finalized.

```

procedure main is
    active_reactive_instance : active_reactive_class.active_reactive_class_acc_t;
    start_result : Boolean;
begin
    -- Create the new instance
    active_reactive_instance := new active_reactive_class.active_reactive_class_t;

    -- Initialize the instance
    active_reactive_class.Initialize(active_reactive_instance.all);

    -- Start the behavior
    active_reactive_class.start_behavior(active_reactive_instance.all, start_result);
    active_reactive_class.start(active_reactive_instance.all);

    -- Send events
    active_reactive_class.gen_trigger_A(active_reactive_instance.all);
    active_reactive_class.gen_trigger_B(active_reactive_instance.all, 1);

    -- Cleanup the instances
    active_reactive_class.Finalize(active_reactive_instance.all);

end main;

```

Figure 221: Using an Active-Reactive Class.

6.2.7. Default Active Class

Instead of explicitly creating an active class for his reactive class, a user has the option of using the automatically generated Default Active Class singleton. The default active class is called “RiA_Default_Active”, and a “With” statement is automatically generated in all <<entrypoint>> packages when needed.

The user can disable the creation of the default active class, as well as control which classes it can act as the active context for, with the *Ada_CG.Configuration.DefaultActiveGeneration* property. The settings are as follows:

Disable: The default active singleton is not created.

ReactiveWithoutContext: This is the default setting. The default active singleton is created if there are reactive classes which consume events and which do not have an active context explicitly specified. The default active singleton can handle only these classes.

All: The default active singleton is generated if there is at least one event-consuming reactive class, and the active singleton can handle all reactive classes that consume events – even those reactive classes that specify another active class as their active context.

```

procedure main is
    reactive_instance : reactive_class.reactive_class_acc_t;
    start_result : Boolean;
begin
    -- Create the new instance
    reactive_instance := new reactive_class.reactive_class_t;

    -- Initialize the instances
    RiA_Default_Active.Initialize;
    reactive_class.Initialize(reactive_instance.all);

    -- Start the behavior
    reactive_class.start_behavior(reactive_instance.all, start_result);
    RiA_Default_Active.start;

    -- Send events
    reactive_class.gen_trigger_A(reactive_instance.all);
    reactive_class.gen_trigger_B(reactive_instance.all, 1);

    -- Cleanup the instances
    RiA_Default_Active.Finalize;
    reactive_class.Finalize(reactive_instance.all);

end main;

```

Figure 222: Using the Default Active Class.

6.2.8. Triggered Operations

A reactive class can also execute synchronously by using triggered operations. A reactive class that uses only triggered operations does not need an active context, and therefore does not produce an Ada task.

NOTE: By default, memory is allocated for each triggered operation that has parameters.

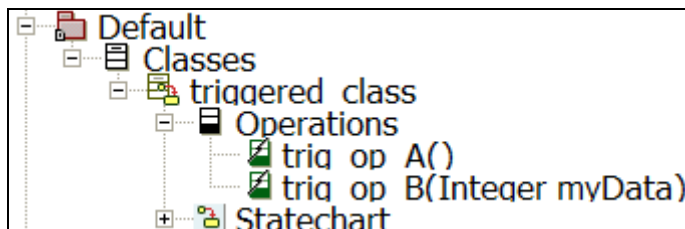


Figure 223: A Sample Model of a Synchronous Reactive Class.

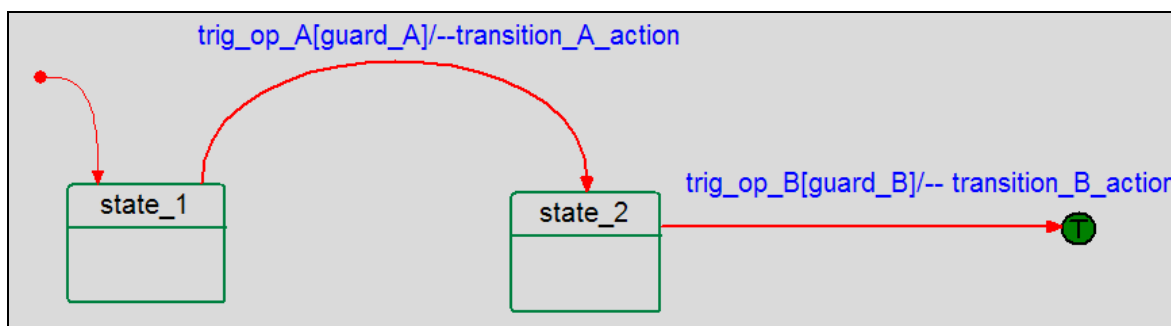


Figure 224: A Statechart Using Triggered Operations.

6.2.1.8 Reactive Class Variables

For each triggered operation, a static attribute is created in the reactive class to contain its unique id.

```
-- identifier for trig_op_A
trig_op_a_id : constant Integer := 1;

-- identifier for trig_op_B
trig_op_b_id : constant Integer := 2;
```

Figure 225: Triggered Operation Unique Identifiers.

6.2.2.8 Reactive Class Public Operations

A synchronous reactive class has the same operations as the asynchronous version, with the exception of the Initialize and Finalize procedures. These procedures are not necessary for a reactive class with only triggered operations.

6.2.3.8 Statechart-Specific Reactive Class Operations

Triggered operations produce the same “gen_<event name>” procedures as an asynchronous reactive class, but in addition, another operation is provided which adds an “in out” parameter to return the event consumption status for this trigger. The type of this parameter is `RiA_Types.Consume_event_status`, and is equal to one of the following values: *event_consumed*, *event_not_consumed*, *instance_in_destruction*, or *reached_terminate*.

Another difference occurs if the triggered operation has a return value. This value becomes an “out” parameter of the given return type.

For the example given above, the following procedures are generated.

```
-- Send the trig_op_A trigger
procedure trig_op_A (this : in out triggered_class_t);

-- Send the trig_op_A trigger and return the consume status
procedure trig_op_A (this : in out triggered_class_t;
  ria_consume_result : out RiA_Types.Consume_event_status);

-- Send the trig_op_B trigger
procedure trig_op_B (this : in out triggered_class_t;
  myData : in Integer; ria_result : out Boolean);

-- Send the trig_op_B trigger and return the consume status
procedure trig_op_B (this : in out triggered_class_t;
  myData : in Integer; ria_result : out Boolean;
  ria_consume_result : out RiA_Types.Consume_event_status);
```

Figure 226: Generated Procedures for Triggered Operations.

6.2.4.8 Using a Synchronous Reactive Class

To use a synchronous reactive class, only two steps are needed: starting the behavior, and calling the triggered operations. The following procedure demonstrates these steps.

```

procedure main (this : in out Entrypoint_t) is
  triggered_instance : triggered_class.triggered_class_acc_t;
  start_result : Boolean;
  operation_result : Boolean;
begin
  -- Create the new instance
  triggered_instance := new triggered_class.triggered_class_t;

  -- Start the behavior
  triggered_class.start_behavior(triggered_instance.all, start_result);

  -- Send events
  triggered_class.trig_op_A(triggered_instance.all);
  triggered_class.trig_op_B(triggered_instance.all, 1, operation_result);

end main;

```

Figure 227: Using a Synchronous Reactive Class.

6.2.5.8 *Avoiding memory allocation on statecharts with only triggered operation*

It is now possible to generate code for statecharts that use only triggered operations which does not allocate memory for triggered operations.

The Ada_CG.Class.OptimizeStatechartsWithoutEventsMemoryAllocation class property can be used to enable this generation scheme.

Note however that this slightly modifies the generated code :

- The Event_data type is now generated as a mutant record
- The Event type does no longer hold an instance of an access type to Event_Data but a direct instance of the Event_Data type.
- The current_event_data operation no longer returns an instance of an access type to Event_Data but a direct instance of the Event_Data type.
- The initialize_event procedure does no longer take a parameter that is an instance of an access type to Event_Data, but a direct instance of the Event_Data type

Should some model using the default implementation be converted to use this new implementation, any calls to the current_event_data and initialize_event operations should take into account the fact that the event_data is no longer represented by an instance of an access to Event_Data type but by an instance of Event_Data type.

6.3. Using the Ada 95 Behavioral frameworks

6.3.1. Limitations

- Generic reactive classes are not supported
- Inherited statecharts for singleton reactive classes are not supported

6.3.2. New Ada 95 Framework changes

Starting with Rhapsody 7.3, the new Ada 95 framework makes several changes in the generation of the reactive and active classes. Reactive classes are given a reactive part that inherits from the Oxf.Reactive type of the OXF framework. The reactive part just delegates the processing of events to its parent user-defined Reactive class. The same concept is used for Active types. They are given an active part which does the handling of events.

When using the default active class, or classes with only triggered operations, the framework is ravenstar compliant. If a user active class is created, some properties must be set in order to be ravenstar compliant (see §6.3.11).

Some changes may occur in user code, if parameterized events are used. The function `get_parameter()` does not exist anymore in new Ada 95 framework (see §6.3.7). Therefore the *params* variable needs to be used directly.

6.3.3. Reactive classes

A `State_Type` enumeration type is declared for each reactive class, it will hold two predefined literals and the qualified names (relatively to the statemachine) of the states as the other literals

```
type State_Type is (
    NON_STATE,
    root_state,
    state_1,
    state_2
);
```

Figure 228: The `State_Type` enumeration type for a reactive class

6.3.4. Event-based reactive classes

Each event-based reactive class must be part of an *active* class that performs the event loop and delivers events to it. Each active class creates its own task to handle its event loop.

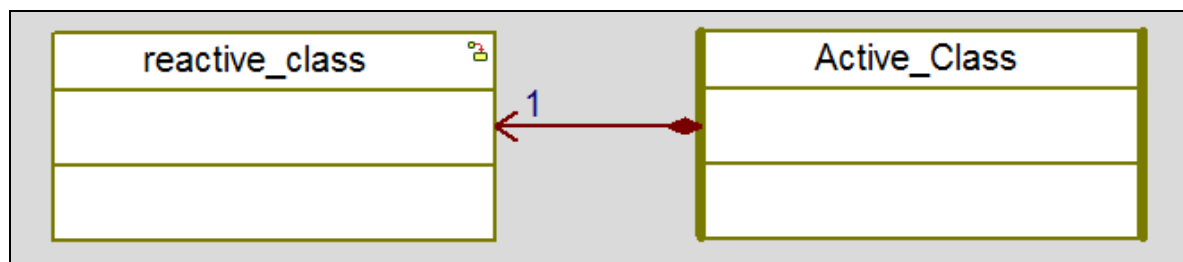


Figure 229: A Reactive Class and its Active Class.

6.3.5. Sending events

Once the active and reactive classes have been started, it is possible to send events to the reactive class. This is done by the calling the “`Gen_<qualified_event_name>`” procedure on the reactive instance. The `<qualified_event_name>` token designates the full path to the event from the root of the model, using underscores as namespace separators.

```
procedure Gen_Default_trigger_a (this : in out reactive_class_t);
procedure Gen_Default_trigger_b (this : in out reactive_class_t;
    myData : in Integer
);
```

Figure 230: Operations to Generate Events for a Reactive Class.

6.3.6. Using triggered operations

Triggered operations are invoked the same way as with the Ada_83 framework.

6.3.7. Accessing the current event parameters

A reference to the `current_event` is available on transitions that have triggers with parameters. To access these parameters, two methods are available :

- use “get_parameters(current_event) .<parameter name>” (only in the previous Ada 95 framework).
- use “params.<parameter name>”. (available on both Ada 95 frameworks).

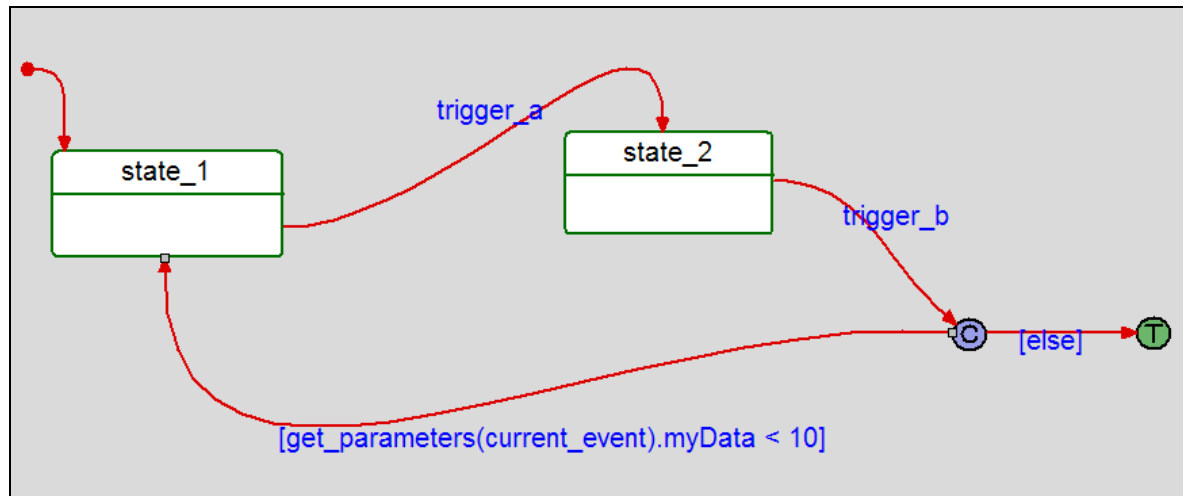


Figure 231: Accessing a trigger parameter value

6.3.8. Testing if a state is active

Use the “Is_In(<qualified_state_name>)” function. The <qualified_state_name> token designates the full path to the state from the root of the statechart, using underscores as namespace separators.

6.3.9. Working with Active and Reactive Classes

The usage is similar to the one of the Ada 83 framework except that reactive classes do not need any initialization or finalization (unless some non statechart related operations, relations, attributes or user-defined code require them).

6.3.10. Default Active Class

The mechanism for default active class is similar to the one available in the Ada 83 framework except that the class is called “Ria_Default_Active_Class” instead of “Ria_Default_Active”

6.3.11. User Active class for ravenscar

To have a ravenscar compliant model with user active classes, those classes must be created statically. To do this, the property **Ada_CG:Class:BaseNumberOfInstances** must be filled with the number of instance that the model needs.

7. *Code order respect tool*

7.1. Introduction

Although the Ada code is generated fully automatically, we allow you to organize the content of the source modules in order to respect your own code writing rules.

So this tool respects:

- Location of declarations for operations, attributes, types, etc, in the specification file and the body file.
- Order of declarations in structures (members), etc...
- Newlines count between declarations.

This tool can be used with basic round-tripping in operation's body.

This tool doesn't reverse code to rhapsody model, also doesn't allow renaming arguments of methods (spec & body), changing method names, and types or attributes names.

All changes of the elements order in the source file are kept in it, and they are not sent back to the model. The mechanism of code order respect consists in merging a generated file with a source file. The order of elements in the source file is preserved, and new elements are added in the source file.

7.2. Activation and usage

To activate this tool:

- Open **IBM Rational Rhapsody Developer for Ada** application.
- Open a project.
- Select a **Component** element and double-click.
- Select **Properties** tab and expand *subject Ada_CG*, choose **Component metaclass** and find *property RespectCodeLayout*.
- Change this property to value **Ordering**.

Note: this tool can be activated only:

- If current language is **Ada**.
- Generated file extension is **ada**, **ads** and **adb**.

7.3. Frequent errors

7.3.1. Syntax error in Ada file

If a syntax error is inserted into source file, the code order respect tool cannot parse the file, and will not merge it with generated code from model. A message appears to show where the error is. User must fix the error into the file before continuing.

An error message like the following one, should appear

```
Generating
D:\Rhapsody\F_CodeRespect\DefaultComponent\DefaultConfig\Default\class_Ada_Task.adb.
Ada syntax error: Encountered " "end" "end "" at line 82, column 9.
Ada syntax error: Encountered " "end" "end "" at line 82, column 9.
```

7.3.2. Syntax error due to model

If Ada code generator generates a file with some syntax error, then the code order respect tool cannot parse the file. In this case, the source file is saved in another file, and the generated file is edited. A message of code generator shows where the error is. User must fix the error into the model. When generated code is clean, the merge becomes possible, and the saved source file can be merged with generated file.

An error message like the following one, should appear

```
Generating
D:\Rhapsody\F_CodeRespect\DefaultComponent\DefaultConfig\Default\class_Ada_Task.adb.
Ada syntax error: Encountered " "end" "end "" at line 83, column 9.
      Your original file is saved as:
D:\Rhapsody\F_CodeRespect\DefaultComponent\DefaultConfig\Default\class_Ada_Task.adb.ordered
      Please update the model and then check the generated file:
D:\Rhapsody\F_CodeRespect\DefaultComponent\DefaultConfig\Default\class_Ada_Task.adb
      When no errors remain, your original file is restored and contains all
updates.
```

7.3.3. Adding a new element

A new element is added at the end of the section (public or private). But this is not necessary the desired location. User will need to move it where he wants. In some case code generator creates some auto generated code. This code may need to be moved.

- When adding a statechart, the class wide declaration, statechart's constants and reactive_part package must be placed at the top of the spec before class record declaration
- When adding a state, the implementation of functions <state>_entry(), <state>_exit() and <state>_process_event() must be moved to the top of body.
- In animated mode, it is recommended to first generate code without code respect order, and to move user code afterwards.

8. Animation in IBM® Rational® Rhapsody® Developer for Ada

IBM® Rational® Rhapsody® Developer for Ada supports tracing and animation of statecharts and sequence diagrams.

8.1. Enabling Animation

Animation is enabled by setting the Instrumentation Mode in the Configuration to "Animation".

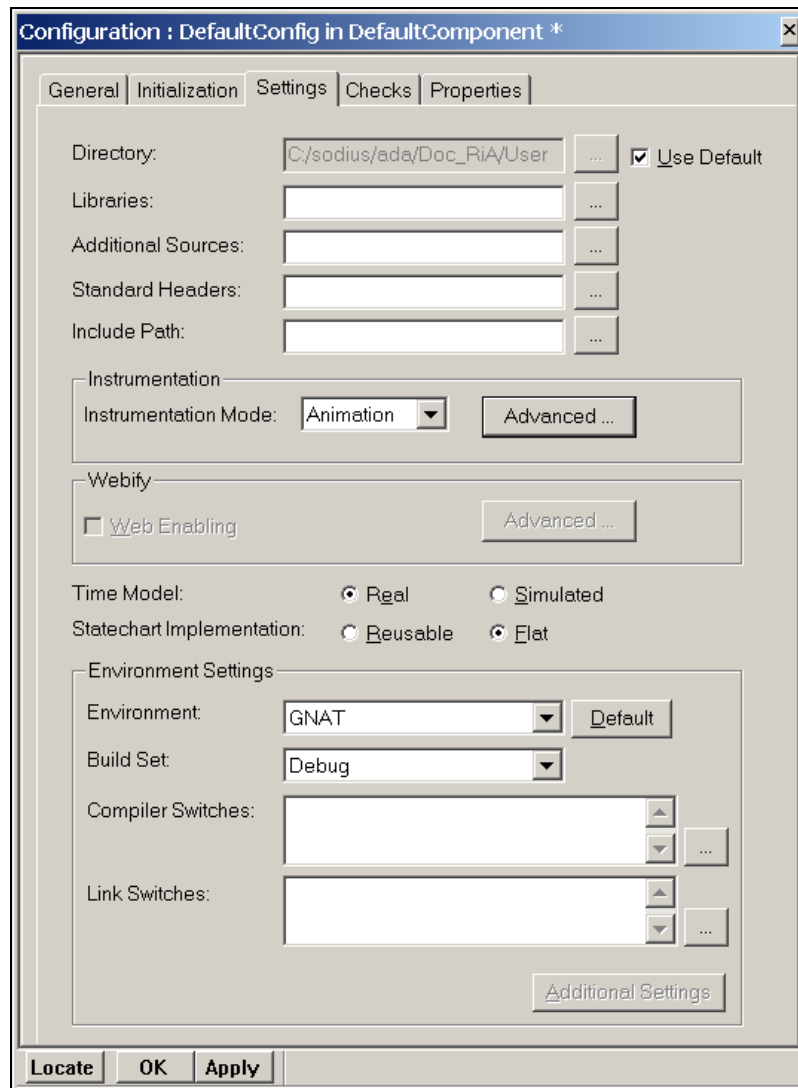


Figure 232: Enabling Animation in the Configuration.

Warning :

If you come back to none animation in Instrumentation Mode, after having generating code, you may have some troubles when compiling the project, because some files are generated in animation mode and not in release mode. To avoid this, you may delete generated files before doing a code generation in none animated mode.

After animation has been enabled, the “Initialize” procedure for any instance to be animated needs be called. This will register the instance with the animation framework.

Animation can be disabled for individual packages, classes, operations, and attributes by setting the corresponding property to “False”.

Element	Property
Package	CG.Type.Animate
Class	CG.Type.Animate
Operation	CG.Operation.Animate
Attribute	CG.Type.Animate

Element	Property
Event	CG.Event.Animate
Argument	CG.Type.Animate

For attributes, arguments, and events that are not standard Ada types, the address of the element will be used for animation. The user can enable the animation of the value by defining an `Add_Attribute` operation on his class for his particular type. Then, by setting the animation property to “Force”, the value of the type will be used instead. For events of user-defined types, it is also necessary to define a `Get_Attribute` operation as well.

8.1.1. Animation of a user defined type

Code generator uses predefined operations to animate predefined types (Integer, Float, Character).

If user defines its own types, then he needs to add some properties and some new functions manually, in order to support those new types.

After having defined this type, two new operations (called for example `Add_Attribute()` and `Get_Attribute()`) must be declared in this type’s package. Operations’ signature and implementation are described further on. Some properties must be set on those 2 new operations :

The property **CG:Operation:Animate** must be unchecked.

The property **Ada_CG:Operation:IsAnimationHelper** must be checked

Some properties must be updated on the type :

Ada_CG:Type:AnimSerializeOperation must be set to : `<type_package>.Add_Attribute`

Ada_CG:Type:AnimUnserializeOperation must be set to : `<type_package>.Get_Attribute`

If a class has an attribute of this user type, then the attribute’s property **CG:Type:Animate** must be set to “Force”.

If an event has a parameter of this user type, then the parameter’s property **CG:Type:Animate** must be set to “Force”.

Here are some examples of serialize/unserialize operations :

Case I : type Integer

```
type My_Integer is range 1..10;
```

```
Function add_attribute()
```

```

procedure Add_Attribute (
  udtName : in String;
  udtValue : in My_Integer;
  udtAttrList : in System.address
) is
begin
  Rhpanim.Add_Attribute( udtName,
                        User_Type_Pkg.My_Integer'Image(udtValue),
                        udtAttrList
                      );
end Add_Attribute;

```

Function Get_attribute()

```

procedure Get_Attribute (
  data : in out My_Integer;
  address : in System.address;
  position : in System.address
) is
  value : integer;
begin
  rhpanim.get_attribute(value,address,position);
  data := User_Type_Pkg.My_Integer'value(value);
end Get_Attribute;

```

Case II : enumerated type :

```

type type_0 is (
  ONE,
  TWO,
  THREE,
  FOUR,
  NULL_NULL
);

```

Function add_attribute()

```

procedure Add_Attribute (
  udtName : in String;
  udtValue : in type_0;
  udtAttrList : in System.address
) is
begin
  Rhpanim.Add_Attribute( udtName,
                        user_type.type_0'Image(udtValue),
                        udtAttrList
                      );
end Add_Attribute;

```

Function get_attribute()

```

procedure Get_Attribute (
  data : in out type_0;
  address : in System.address;

```

```
    position : in System.address
  ) is
    value : string(1..10);
begin
  rhpanim.get_attribute(value, address, position);
  if (value = "ONE"      ) then
    data := User_Type.ONE;
  elsif (value = "TWO    ") then
    data := User_Type.TWO;
  elsif (value = "THREE  ") then
    data := User_Type.THREE;
  elsif (value = "FOUR   ") then
    data := User_Type.FOUR;
  else
    data := User_Type.NULL_NULL;
  end if;
end Get_Attribute;
```

8.2. Animation on Remote Host

By adding the following line to the <Rhapsody_Install_Dir>\Sodius\Sodius.ini file, one can enable remote host animation

```
animationAddress=<RemoteHostIPAddress>
```

Alternatively, one can use the Ada_CG.<Compiler>.UseRemoteHost and Ada_CG.<Compiler>.RemoteHost properties to activate remote host animation.

9. **Generation Rules Customization**

9.1. **Overview**

Rhapsody® in Ada uses a rule-base engine for its code generation. The rules are written using a combination of WYSIWYG (“What-You-See-Is-What-You-Get”) templates and java macros to describe the desired contents for the generated Ada code. Each user can create his own rules, and use them when generating code from the application.

9.2. **Rules Modification**

The rules are available for modification by using the RulesComposer in tools menu of IBM® Rational® Rhapsody® environment. Choose the IBM® Rational® Rhapsody® Developer for Ada ruleset in RulesComposer’s Welcome page, to launch the RulesComposer with the IBM® Rational® Rhapsody® Developer for Ada rules. From the user-friendly interface, the rules are fully modifiable. See the documentation in the <Rhapsody>\Sodius\RulesComposer\help directory.

9.3. **Legacy UML 1.3 metamodel based ruleset**

With the release of Rhapsody 7.0, there has been a change in the underlying code generation RuleSet. Instead of being based on the UML 1.3 metamodel, the RuleSet is now based on the Rhapsody metamodel.

The main benefits of this change are the following :

- Performance improvements gained from the elimination of the Rhapsody to UML transformation
- Better support of Rhapsody unique features

Existing customizations of the legacy UML 1.3 metamodel based RuleSet should be migrated to the Rhapsody metamodel based RuleSet

10. Compilers and related tools support

10.1. Supported compilers/IDEs, tools & environments

Compiler/IDE/Tool	Target Environments
GNAT	Win32
Aonix's ObjectAda	Win32, Raven/PPC
GreenHills Software's AdaMULTI	Win32, Integrity
Praxis High Integrity System's SPARK Examiner	All

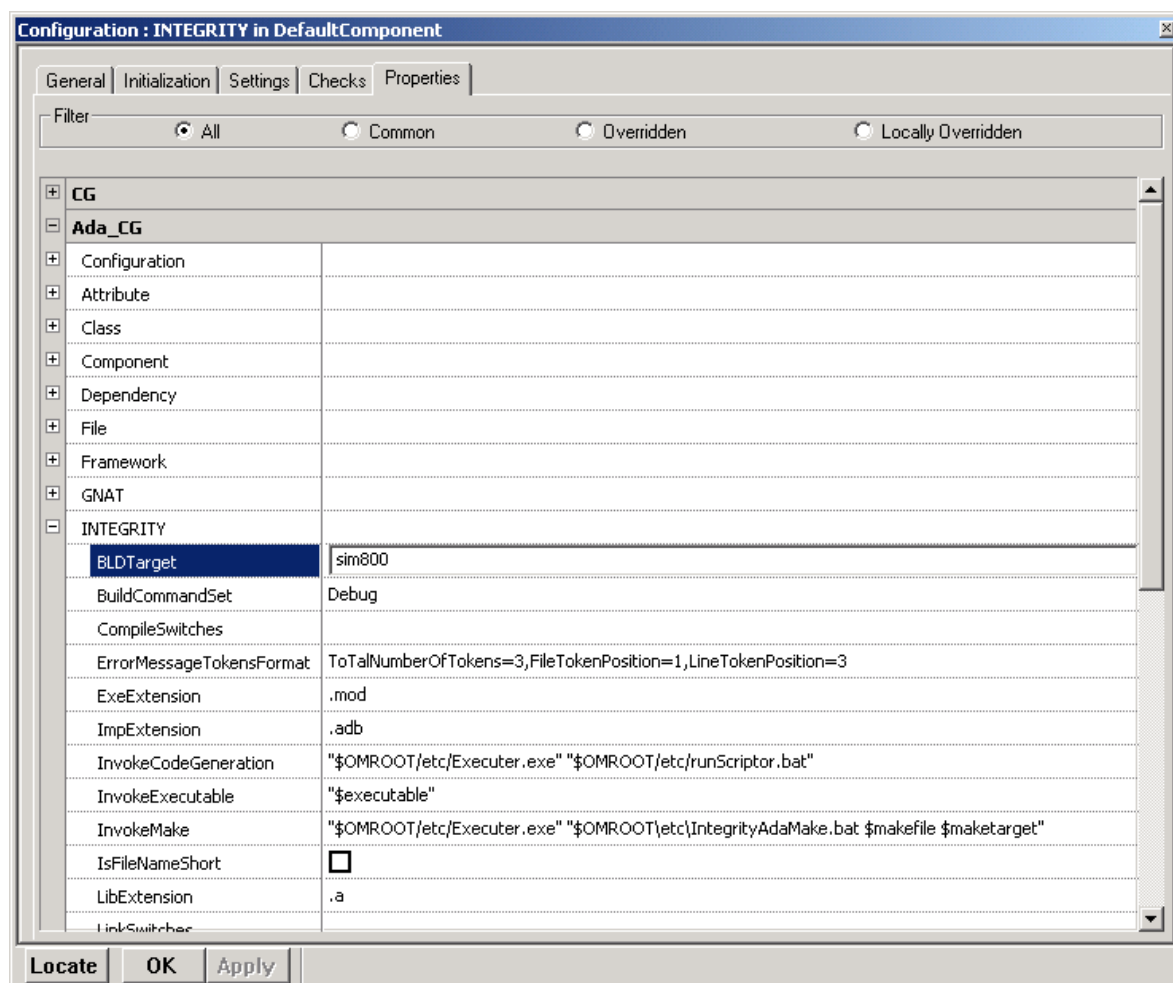
10.2. Environment specific instructions

10.2.1. Using the INTEGRITY simulator with Rhapsody

Please refer to the ["Using Integrity Simulator with Rhapsody"](#) document for detailed instructions.

10.2.2. INTEGRITY BSP support

When generating code for GreenHills' INTEGRITY operating system, you can modify the board target for the selected component via its properties (select the features entry in its context menu).



10.2.3. Raven/PPC BSP Support

By default, the following BSP libraries are registered when compiling for Raven PPC:

- raven\standard_model
- system\simulator
- lib\extensions

You can override these defaults by modifying the Ada_CG.Raven_PPC.BSP_Libraries property.

10.2.4. GNAT issues

On Win32 platforms, using GNAT 3.15p or earlier releases, some generated executables using I/O features may cause generated applications to hang.

Using recent versions of GCC such as 3.4.2, the problem is solved.

10.3. Compiler usage note for ObjectAda and GreenHills compilers

If you regenerate code for an Ada Library that has already gone through a generate-build cycle, make sure that you rebuild it too (that is “clean and build” or “rebuild”), and not only build it. If you do not, you are very likely to encounter compilation errors when compiling code that is using this library saying that the source file that you are using is newer than the registered file.

10.4. Compiler support limitations

GreenHills compilers won't compile packages named main. Main is used for the ada runtime entrypoint.

Using Rhapsody with Aonix ObjectAda compilers requires that Rhapsody be installed in a directory with no spaces in it.

10.4.1. Rhapsody Frameworks support

Compiler	Behavioral framework	Animation
GNAT / Win32	Yes	Yes
ObjectAda / Win32	Yes	Yes
ObjectAda / Raven/PPC	No	No
AdaMulti / Win32	Yes	Yes
AdaMulti / Integrity	Yes	Yes

Behavioral framework and model animation are not supported for ObjectAda RavenPPC as they violate some of the Raven profile restrictions.

10.4.2. Compilation error messages

The compilation error messages for ObjectAda get displayed in Rhapsody, and you can access the offending line in the related source file with the following limitation: error messages generated by the source registration utility (adareg) are not navigable while other error messages (generated by adacomp, directly or via adabuild) are navigable.

10.4.3. Notes on Pre-compiled libraries

Pre-compiled libraries use with Aonix ObjectAda compiler

Code generated for ObjectAda supports directory based libraries only. This means that object files for that library must be in the same directory as the sources (this is a limitation to be overcome in future releases).

The consequences are that you can specify the directory where your library is located using the “Include Path” field in the “General” tab of a component/configuration properties and the generated compilation commands will look in this directory for sources and object files as well (using `adaopts -p` command). If you put something in the “libraries” field, the behavior at link time is unpredictable if you do not modify the generated compilation batch file.

If you have several libraries to link to, put one library per line in the “Include Path” field, but do not use any separator such as ‘,’ or ‘;’.

If you insist on using library archive files in a different location from the library source files, here is what to do:

1. In the “Include path” field, specify the directory `<sourcedir>` where the sources for the library are located. Regenerate your makefile.
2. In the generated makefile, replace lines starting with `“adaopts -p <sourcedir>”` by `“adaopts -ep <sourcedir>”`. Do not replace lines starting with `adaopts -p` that register other directories different from the ones you specified in “Include path” field.
3. In the “libraries” field, specify the full path or the relative path from the generation directory to the library archive file(s) you want to use
4. Or instead of step 3, add the following line to the generated makefile where `<libdir>` is the directory where your library archive is located. This is usually more convenient as you can put all your libraries in a single directory and get ObjectAda find them with a single line.
`“adaopts -ip <libdir>”`

Please refer to ObjectAda documentation for more details on source and library registration and linking

Library archive files and GreenHills compilers

Object files for libraries generated by IBM® Rational® Rhapsody® Developer for Ada are put in archive files (.a for GreenHills and GNAT, and .lib for Aonix). Such archive files are located in the same directories as source files for the libraries.

For GreenHills, if you want to use libraries whose object files are not archived in a file but in a subdirectory of the library directory (as is often the case when you do not use library archive files) with code generated by Rhapsody you have to do the following:

If you are using Multi 3.5 or an older version :

- In the top level build file of the generated component, add the following command (do not forget the leading tab), where `<librarydirectory>` is the directory where your library is located.

```
:adalibdirs=<librarydirectory>
```

10.5. Compiler and assimilated tools related properties

The following properties have an impact on the generated compilation commands :

Element	
<Property>	<Description>
Configuration	
GNAT	
Ada_CG.GNAT.BuildCommandSet	Sets debug switch for generated gnatmake commands
Ada_CG.GNAT.CompileSwitches	Inserts user-defined compilation switches into gnatmake commands
Ada_CG.GNAT.LinkSwitches	Inserts user-defined link switches into gnatmake commands
AdaMULTI PowerPC (v4.0 and newer)	
Ada_CG.INTEGRITY5.BLDAdditionalOptions	Inserts user-defined options in the build file for the component
Ada_CG.INTEGRITY5.BLDMainExecutableOptions	Inserts user-defined options in every executable build file generated for the current component configuration
Ada_CG.INTEGRITY5.BLDMainLibraryOptions	Inserts user-defined options in the build file for the component if it is of library type
Ada_CG.INTEGRITY5.BLDTarget	Sets the board target
Ada_CG.INTEGRITY5.BuildCommandSet	Activates debug mode for generated top level build file
Ada_CG.INTEGRITY5.CompileSwitches	Inserts user-defined compilation switches into top level build file
Ada_CG.INTEGRITY5.DebugSwitches	Sets debug level used in debug build
Ada_CG.INTEGRITY5.LinkSwitches	Inserts user-defined compilation switches into top level build file
Ada_CG.INTEGRITY5.IntegrityRoot	Holds the value of the "os_dir" parameter generated in build files.
AdaMULTI PowerPC (v3.5 and older)	
Ada_CG. INTEGRITY.BLDAdditionalOptions	Inserts user-defined options in the build file for the component
Ada_CG.INTEGRITY.BLDMainExecutableOptions	Inserts user-defined options in every executable build file generated for the current component configuration
Ada_CG.INTEGRITY.BLDMainLibraryOptions	Inserts user-defined options in the build file for the component if it is of library type

Element	
<Property>	<Description>
Ada_CG.INTEGRITY.BLDTarget	Sets the board target
Ada_CG.INTEGRITY.BuildCommandSet	Activates debug mode for generated top level build file
Ada_CG.INTEGRITY.CompileSwitches	Inserts user-defined compilation switches into top level build file
Ada_CG.INTEGRITY.DebugSwitches	Sets debug level used in debug build
Ada_CG.INTEGRITY.LinkSwitches	Inserts user-defined compilation switches into top level build file
AdaMULTI Win32 (v4.0 and older)	
Ada_CG.Multi4Win32.BLDAdditionalOptions	Inserts user-defined options in the build file for the component
Ada_CG.Multi4Win32.BLDMainExecutableOptions	Inserts user-defined options in every executable build file generated for the current component configuration
Ada_CG.Multi4Win32.BLDMainLibraryOptions	Inserts user-defined options in the build file for the component if it is of library type
Ada_CG.Multi4Win32.BuildCommandSet	Activates debug mode for generated top level build file
Ada_CG.Multi4Win32.CompileSwitches	Inserts user-defined compilation switches into top level build file
Ada_CG.Multi4Win32.DebugSwitches	Sets debug level used in debug build
Ada_CG.Multi4Win32.LinkSwitches	Inserts user-defined compilation switches into top level build file
AdaMULTI Win3 (v3.5 and older)	
Ada_CG.MultiWin32.BLDAdditionalOptions	Inserts user-defined options in the build file for the component
Ada_CG.MultiWin32.BLDMainExecutableOptions	Inserts user-defined options in every executable build file generated for the current component configuration
Ada_CG.MultiWin32.BLDMainLibraryOptions	Inserts user-defined options in the build file for the component if it is of library type
Ada_CG.MultiWin32.BuildCommandSet	Activates debug mode for generated top level build file
Ada_CG.MultiWin32.CompileSwitches	Inserts user-defined compilation switches into top level build file
Ada_CG.MultiWin32.DebugSwitches	Sets debug level used in debug build
Ada_CG.MultiWin32.LinkSwitches	Inserts user-defined compilation switches into top level build file

Element	
<Property>	<Description>
ObjectAda Win32	
Ada_CG.OBJECTADA.BuildCommandSet	Activates debug switches for generated adacomp and adabuild commands
Ada_CG.OBJECTADA.CompileSwitches	Inserts user-defined compilation switches into adacomp or adabuild commands
Ada_CG.OBJECTADA.DebugSwitches	Sets debug level used in debug switches
Ada_CG.OBJECTADA.LinkSwitches	Inserts user-defined compilation switches into adabuild commands
Raven_PPC	
Ada_CG.RAVEN_PPC.BuildCommandSet	Activates debug switches for generated adacomp and adabuild commands
Ada_CG.RAVEN_PPC.CompileSwitches	Inserts user-defined compilation switches into adacomp or adabuild commands
Ada_CG.RAVEN_PPC.DebugSwitches	Sets debug level used in debug switches
Ada_CG.RAVEN_PPC.LinkSwitches	Inserts user-defined compilation switches into adabuild commands
Ada_CG.RAVEN_PPC.BSP_Libraries	Default BSP libraries to link to.
SPARK	
Ada_CG.SPARK.BriefErrorMessages	Generates a /brief option on SPARK Examiner calls.
Ada_CG.SPARK.OpenHTMLReports	Instructs Rhapsody to open the HTML reports on examination completion
Ada_CG.SPARK.TargetConfigurationFileName	Specifies a target configuration file name to be passed on as an argument to the SPARK Examiner

11. Model Limitations

11.1. Limitations for Ada 83

When creating a model for Ada 83, the following guidelines should be followed.

- Do not create virtual operations.
- Do not create abstract operations.
- Do not create abstract classes.
- Do not use generalizations.

11.2. General Limitations

- Active classes must be defined to be private.

Appendix A: Properties for IBM® Rational® Rhapsody® Developer for Ada

Compiler and assimilated tools specific properties are not listed here. You can find their description in the [“Compiler and assimilated tools related properties”](#) section.

Element	
<Property>	<Description>
Actor	
Ada_CG.Class.AccessTypeName	Overrides the access type name.
Ada_CG.Class.ClassWideAccessTypeName	Overrides the class-wide access type name.
Ada_CG.Class.Final	Makes the class record non-“tagged”.
Ada_CG.Class.GenerateAccessType	Turns off the generation of the access type.
Ada_CG.Class.GenerateClassWideAccessType	Controls the generation of the class-wide access type.
Ada_CG.Class.GenerateRecordType	Turns off the generation of the record type.
Ada_CG.Class.HasUnknownDiscriminant	If true, an unknown discriminant (<>) will be generated for this class.
Ada_CG.Class.ImplementationEpilog	Adds an epilog to the package body.
Ada_CG.Class.ImplementationPragmas	Holds user-defined pragmas to generate in body.
Ada_CG.Class.ImplementationPragmasInContextClause	Holds user-defined pragmas to generate in context clause of body.
Ada_CG.Class.ImplementationProlog	Adds a prolog to the package body.
Ada_CG.Class.InitializationCode	Adds initialization code in the class package body.
Ada_CG.Class.IsLimited	Indicates if the record type is to be generated as limited.
Ada_CG.Class.IsNested	Indicates if the class is to be generated as a nested package.
Ada_CG.Class.IsPrivate	Indicates if the class is to be generated as a private package.
Ada_CG.Class.NestingVisibility	Indicates where in the nesting package the specification of the nested package should be generated.
Ada_CG.Class.OptimizeStatechartsWithoutEventsMemoryAllocation	Controls whether the generated statechart code will use dynamic memory allocation or not on statecharts that use only triggered operations.
Ada_CG.Class.RecordTypeName	Overrides the generated record type name.
Ada_CG.Class.SpecificationEpilog	Adds an epilog to the package specification.

Element	
<Property>	<Description>
Ada_CG.Class.SpecificationPragmas	Holds user-defined pragmas to generate in spec.
Ada_CG.Class.SpecificationPragmasInContextClause	Holds user-defined pragmas to generate in context clause of spec.
Ada_CG.Class.SpecificationProlog	Adds a prolog to the package specification.
Ada_CG.Class.TaskBody	Overrides the generated task body.
Ada_CG.Class.UseAda83Framework	If “True”, then generated code for statecharts, events and guarded operations and attributes will use Ada 83 constructs. If “False”, Ada 95 constructs will be used.
Ada_CG.Class.Visibility	Determines the location of the record type.
Ada_CG.File.ImplementationFooter	Overrides the generated implementation footer.
Ada_CG.File.ImplementationHeader	Overrides the generated implementation header.
Ada_CG.File.SpecificationFooter	Overrides the generated specification footer.
Ada_CG.File.SpecificationHeader	Overrides the generated specification header.
Ada_CG.Operation.AlphabeticalSort	Controls alphabetical sorting of operations on generation.
Ada_CG.Operation.VirtualMethodGenerationScheme	Enables backward compatibility mode for methods of interface and abstract classes.
CG.Class.Concurrency	Determines if the class is active.
CG.Class.ImplementStatechart	Controls the generation of statechart code for this class.
CG.Class.UseAsExternal	Turns off generation for this object.
CG.Type.Animate	Turns off animation for this object.
Argument	
Ada_CG.Argument.AccessTypeUsage	Controls whether the actual type for the attribute is the class record type, the regular access type, or the class-wide access type. Only works if the type is a class.
Ada_CG.Argument.AsAccess	Used to set parameter passing mode as access.
Ada_CG.Argument.ClassWide	Controls the generation of a class-wide modifier for the argument.
Ada_CG.Type.AnimEnumerationTypeImage	Activates usage of Image attribute for enumerated types when using animation.
Ada_CG.Type.AnimSerializeOperation	Overrides generated serialize operation for animation.
CG.Argument.Animate	Enables animation of the argument.
CG.Type.Animate	Used to “force” animation of the argument.

Element	
<Property>	<Description>
Attribute	
Ada_CG.Attribute.Accessor	Controls the name of the accessor.
Ada_CG.Attribute.AccessorGenerate	Controls the generation of the accessor.
Ada_CG.Attribute.AccessTypeUsage	Controls whether the actual type for the attribute is the class record type, the regular access type, or the class-wide access type. Only works if the type is a class.
Ada_CG.Attribute.DeclarationPosition	Controls the declaration position of a static attribute relatively to the section (public part of spec, private part of spec, body) it is declared in and to the “virtual” location of the class record type if it is/was declared in this section.
Ada_CG.Attribute.DeferredInitializationPosition	Applicable to public constants only, this property controls where in the private part the deferred initialization is generated.
Ada_CG.Attribute.GenerateRenamesForSingleton	Controls the generation of renaming statements for attributes in singleton classes.
Ada_CG.Attribute.ImplementationEpilog	Adds an epilog to the attribute accessors body.
Ada_CG.Attribute.ImplementationProlog	Adds a prolog to the attribute accessors body.
Ada_CG.Attribute.InlineAccessor	Controls generation of inline pragma for the accessor.
Ada_CG.Attribute.InlineMutator	Controls generation of inline pragma for the mutator.
Ada_CG.Attribute.IsAliased	Determines if attribute is aliased.
Ada_CG.Attribute.Mutator	Controls the name of the mutator.
Ada_CG.Attribute.MutatorGenerate	Controls generation of the mutator.
Ada_CG.Attribute.ParentDiscriminantValue	Holds the value to assign to the parent discriminant if it exists.

Element	
<Property>	<Description>
Ada_CG.Attribute.RedefiningDiscriminantPolicy	<p>If the attribute is a <<Discriminant>> attribute and it is already defined as a <<Discriminant>> in one of the parent classes of the current class, this property controls the generation policy for this discriminant</p> <p>AsNew : attribute is generated as a regular discriminant</p> <p>AsNewAndOverriding : attribute is generated as a regular discriminant and the parent discriminant of the same name is assigned the value defined in the “Ada_CG.Attribute.ParentDiscriminantValue” property.</p> <p>AsOverriding : the parent discriminant of the same name is assigned the value defined in the “Ada_CG.Attribute.ParentDiscriminantValue” property.</p>
Ada_CG.Attribute.Renames	Holds the name of the variable this attribute is renaming (only works for static attributes in a class or for attributes in a package).
Ada_CG.Attribute.SpecificationEpilog	Adds an epilog to the attribute specification.
Ada_CG.Attribute.SpecificationProlog	Adds a prolog to the attribute specification.
Ada_CG.Attribute.Visibility	Determines the visibility of the attribute.
Ada_CG.Type.AnimEnumerationTypeImage	Activates usage of Image attribute for enumerated types when using animation.
Ada_CG.Type.AnimSerializeOperation	Overrides generated serialize operation for animation.
CG.Attribute.Animate	Turns off animation of this attribute.
CG.Attribute.Generate	Turns off generation of the attribute.
CG.Attribute.IsGuarded	Generates a guarded attribute.
CG.Type.Animate	Turns off animation of this attribute type.
Class	
Ada_CG.Class.AccessTypeName	Overrides the access type name.
Ada_CG.Class.ClassWideAccessTypeName	Overrides the class-wide access type name.
Ada_CG.Class.DeclarationPosition	<p>Applicable to nested classes only.</p> <p>Determines declaration position relatively to the section (public part of spec, private part of spec, body) it is declared in and to the “virtual” location of the class record type if it is/was declared in this section.</p>

Element	
<Property>	<Description>
Ada_CG.Class.Final	Makes the class record non-"tagged".
Ada_CG.Class.GenerateAccessType	Controls the generation of the access type.
Ada_CG.Class.GenerateClassWideAccessType	Controls the generation of the class-wide access type.
Ada_CG.Class.GenerateRecordType	Turns off the generation of the record type.
Ada_CG.Class.ImplementationEpilog	Adds an epilog to the package body.
Ada_CG.Class.ImplementationPragmas	Holds user-defined pragmas to generate in body.
Ada_CG.Class.ImplementationPragmasInContextClause	Holds user-defined pragmas to generate in context clause of body.
Ada_CG.Class.ImplementationProlog	Adds a prolog to the package body.
Ada_CG.Class.InitializationCode	Adds initialization code in the class package body.
Ada_CG.Class.IsLimited	Indicates if the record type is to be generated as limited.
Ada_CG.Class.IsNested	Indicates if the class is to be generated as a nested package.
Ada_CG.Class.IsPrivate	Indicates if the class is to be generated as a private package.
Ada_CG.Class.IsStatic	<p>This property indicates whether the class is a regular class (Unchecked) or a static class (Checked).</p> <p>A static class has no record type and all its attributes and operation are static. The parameter "this" is never generated</p>
Ada_CG.Class.NestingVisibility	Indicates where in the nesting package the specification of the nested package should be generated.
Ada_CG.Class.OptimizeStatechartsWithoutEventsMemoryAllocation	Controls whether the generated statechart code will use dynamic memory allocation or not on statecharts that use only triggered operations.
Ada_CG.Class.RecordTypeName	Overrides the generated record type name.
Ada_CG.Class.RelativeEventDataRecordTypeComponentsNaming	<p>Enables relative naming of event data record type components representing events and triggered operations parameters. If set to true, there shall be no events or triggered operations sharing an argument name, as they would generate record components with the same name, which would be uncompileable.</p> <p>When using triggered operations in a statechart, this property should be modified at the class level.</p>

Element	
<Property>	<Description>
Ada_CG.Class.Renames	Holds the name of the class or package this class is renaming.
Ada_CG.Class.SpecificationEpilog	Adds an epilog to the package specification.
Ada_CG.Class.SpecificationPragmas	Holds user-defined pragmas to generate in spec.
Ada_CG.Class.SpecificationPragmasInContextClause	Holds user-defined pragmas to generate in context clause of spec.
Ada_CG.Class.SpecificationProlog	Adds a prolog to the package specification.
Ada_CG.Class.Visibility	Determines the location of the record type.
Ada_CG.File.ImplementationFooter	Overrides the generated implementation footer.
Ada_CG.File.ImplementationHeader	Overrides the generated implementation header.
Ada_CG.File.SpecificationFooter	Overrides the generated specification footer.
Ada_CG.File.SpecificationHeader	Overrides the generated specification header.
CG::File::InvokePostProcessor	Runs a post-processing utility on the code that is generated by Rational Rhapsody. For example, you could run a “beautify” program to get a specific coding style.
Ada_CG.Operation.AlphabeticalSort	Controls alphabetical sorting of operations on generation.
Ada_CG.Operation.IsEntry	Determines if the operation is a task entry or a regular operation in <<AdaTask>> and <<AdaTaskType>> classes.
Ada_CG.Operation.ThisName	Modifies the name of the “this” parameter for instance level operations.
Ada_CG.Operation.VirtualMethodGenerationScheme	Enables backward compatibility mode for methods of interface and abstract classes.
CG.Class.Concurrency	Determines if the class is active.
CG.Class.ImplementStatechart	Controls the generation of statechart code for this class.
CG.Class.UseAsExternal	Turns off generation for this object.
CG.Type.Animate	Turns off animation for this object.
SPARK.Class.ExaminerLevelBody	Sets Examiner level for this class body.
SPARK.Class.ExaminerLevelSpec	Sets Examiner level for this class spec.
Class <<AdaTask>, <<AdaTaskType>>	
Ada_CG.Class.TaskBody	Overrides the generated task body.
Ada_CG.Operation.TaskDefaultScheme	Sets the task default entry scheme.

Element	
<Property>	<Description>
Ada_CG.Operation.TaskDefaultSchemeDelayStatement	Sets the task default entry delay statement for timed default entry scheme.
CG.Class.ActiveThreadPriority	Determines the task priority.
Class <<Singleton>>	
Ada_CG.Class.SingletonExposeThis	
Ada_CG.Class.SingletonInstanceVisibility	Controls where the singleton unique instance is generated. Available values are : <ul style="list-style-type: none"> ○ Body(default) ○ Private
Component	
Ada_CG.Component.AdaVersion	Sets the code style to be Ada 83.
Ada_CG.Component.UseAdaFramework	Selects between the Ada 83, Ada 95, and ravenstar compatible Ada 95 frameworks.
Ada_CG.Component.UseBoochComponents	Selects between the Booch 83 and 95 components to be used for collections.
Ada_CG.Component.RespectCodeLayout	Enables code order respect
Ada_CG.Configuration.DefaultActiveGeneration	Determines the generation of the Default Active class.
Ada_CG.File.ImplementationFooter	Overrides the generated implementation footer.
Ada_CG.File.ImplementationHeader	Overrides the generated implementation header
Ada_CG.File.SpecificationFooter	Overrides the generated specification footer
Ada_CG.File.SpecificationHeader	Overrides the generated specification header
Configuration	
Ada_CG.Configuration.ImplementationProlog	Adds a prolog to the generated entrypoint.
Ada_CG.Configuration.ImplementationEpilog	Adds an epilog to the generated entrypoint.
Ada_CG.Configuration.GenerateAnnotationsForNonSPARKConfigurations	Enables generation of SPARK annotations even if active environment is not SPARK. This is only effective if SPARK profile is loaded.
Ada_CG.Configuration.LocalVariablesDeclaration	Provides the local variables for auto-generated entrypoint.

Element	
<Property>	<Description>
Ada_CG::Configuration::DescriptionBeginLine	<p>This property specifies the prefix for the beginning of comment lines in the generated code. This functionality uses a documentation system (such as Doxygen), which looks for a certain prefix to produce the documentation.</p> <p>This property affects only the code that is generated for descriptions of model elements; other auto-generated comments are not affected.</p> <p>The default value is "--"</p>
Ada_CG::Configuration::DescriptionEndLine	<p>This property specifies the prefix for the end of comment lines in the generated code. This functionality uses a documentation system (such as Doxygen), which looks for a certain prefix to produce the documentation.</p> <p>This property affects only the code that is generated for descriptions of model elements; other auto-generated comments are not affected.</p> <p>The default value is ""</p>
Ada_CG.File.ImplementationFooter	Overrides the generated implementation footer.
Ada_CG.File.ImplementationHeader	Overrides the generated implementation header
Ada_CG.File.SpecificationFooter	Overrides the generated specification footer
Ada_CG.File.SpecificationHeader	Overrides the generated specification header
Ada_CG.Relation.ObjectInitialization	Determines what kind of initialization shall occur for configuration's initial instances
CG.Configuration.GenerationDirectoryPerModelComponent	<p>Determines if each package will be generated into its own subdirectory.</p> <p>This property is obsolete. It is replaced by CG.Package.GenerateDirectory</p>
CG.Configuration.LineWrapLength	Specifies the length of the code line generated during code generation.
Dependency	
Ada_CG.Dependency.ImplementationEpilog	Adds an epilog to the resulting with clause (if any) in package body.
Ada_CG.Dependency.ImplementationProlog	Adds a prolog to the resulting with clause (if any) in package body.
Ada_CG.Dependency.SpecificationEpilog	Adds an epilog to the resulting with clause (if any) in package specification.
Ada_CG.Dependency.SpecificationProlog	Adds a prolog to the resulting with clause (if any) in package specification.

Element	
<Property>	<Description>
CG.Dependency.GenerateRelationWithActors	Determines generation when target is an actor.
CG.Dependency.UsageType	Determines the location (package specification or package implementation) of the “With” and optional “Use” or “Use Type” statements.”
Dependency <<Renames>>	
CG.Dependency.UsageType	For <<Renames>> dependencies this property is only applicable to dependencies between operations, it controls whether the operation is “renaming as specification” or “renaming as body”
Dependency <<Usage>>	
Ada_CG.Dependency.AccessTypeUsage	Controls whether the actual type referred to in a use type clause is the class record type, the regular access type, or the class-wide access type. Only works if the type is a class.
Ada_CG.Dependency.CreateUseStatement	Creates a “Use” or “Use Type” statement.
Ada_CG.Dependency.GeneratePragmaElaborate	Generated an “elaborate” pragma fo the supplier class or package in the client class or package
Ada_CG.Dependency.GeneratePragmaElaborateAll	Generated a “preelaborate” pragma fo the supplier class or package in the client class or package
Ada_CG.Dependency.UsesStatementPosition	Specifies whether the “Use” or “Use type” clause should be generated in the package context clause (before the “package” keyword) or in the package declaration or body (after the “package” keyword)
Event	
Ada_CG.Class.RelativeEventDataRecordTypeComponentsNaming	<p>Enables relative naming of event data record type components representing events and triggered operations parameters. If set to true, there shall be no events or triggered operations sharing an argument name, as they would generate record components with the same name, which would be uncompileable.</p> <p>When using events in a statechart, this property should be modified on the events being used.</p>
CG.Event.Animate	Turns off animation of the event.
Generalization	
CG.Generalization.Generate	Turns off generation.
Operation	

Element	
<Property>	<Description>
Ada_CG.Operation.DeclarationPosition	Determines declaration position relatively to the section (public part of spec, private part of spec, body) it is declared in and to the “virtual” location of the class record type if it is/was declared in this section.
Ada_CG.Operation.EntryCondition	Specifies the task entry guard.
Ada_CG.Operation.GenerateForwardDeclarationInPackageBody	Controls generation of forward declaration of an operation defined as “private”.
Ada_CG.Operation.GenerateImplementation	Controls generation of body for the operation
Ada_CG.Operation.ImplementationEpilog	Adds an epilog to the package body.
Ada_CG.Operation.ImplementationName	Overrides the name to be used for the operation in the generated code. Useful as a workaround for defining operations that differ only by their return type in a same class or package.
Ada_CG.Operation.ImplementationProlog	Adds a prolog to the package body.
Ada_CG.Operation.Inline	Indicates to inline the operation.
Ada_CG.Operation.IsAnimationHelper	Indicates if this operation should only be generated when animating model.
Ada_CG.Operation.IsEntry	Determines if the operation is a task entry or a regular operation in <<AdaTask>> and <<AdaTaskType>> classes.
Ada_CG.Operation.Kind	Determines if the operation is abstract.
Ada_CG.Operation.LocalVariablesDeclaration	Provides the local variables.
Ada_CG.Operation.PreserveUserCode	Controls the generation of tags that allow to preserve user changes to a file over successive generations.
Ada_CG.Operation.Renames	Holds the name of the operation this operation is renaming (signatures of operations must match).
Ada_CG.Operation.RenamesKind	Specifies if the renaming of the operation designated in the “Ada_CG.Operation.Renames” property is “as specification” or “as body”.
Ada_CG.Operation.ReturnTypeByAccess	Controls whether the return type is generated as an access type, a class-wide access type or a regular type or not. Effective only if the return type is a class.
Ada_CG.Operation.SpecificationEpilog	Adds an epilog to the package specification.
Ada_CG.Operation.SpecificationProlog	Adds a prolog to the package specification.
Ada_CG.Operation.ThisAccessTypeUsage	Controls whether the actual type for the “this” parameter is the class record type, the regular access type, or the class-wide access type.

Element	
<Property>	<Description>
Ada_CG.Operation.ThisByAccess	Sets passing mode for this parameter to “access” if set
Ada_CG.Operation.ThisName	Modifies the name of the “this” parameter for instance level operations.
Ada_CG.Operation.ThisPassingMode	When set to a value different from “FromGUI”, overrides the setting of the “constant” checkbox of an operation and the one of the “Ada_CG.Operation.ThisByAccess” property.
Ada_CG.Operation.VirtualMethodGeneration Scheme	Enables backward compatibility mode for methods of interface and abstract classes.
CG.Operation.Animate	Turns off animation for this operation.
CG.Operation.Generate	Turns off generation for this operation.
CG.Operation.Concurrency	Sets the operation to be guarded.
Package	
Ada_CG.File.ImplementationFooter	Overrides the generated implementation footer.
Ada_CG.File.ImplementationHeader	Overrides the generated implementation header
Ada_CG.File.SpecificationFooter	Overrides the generated specification footer
Ada_CG.File.SpecificationHeader	Overrides the generated specification header
Ada_CG.Operation.AlphabeticalSort	Controls alphabetical sorting of operations on generation
Ada_CG.Package.ContributesToNamespace	Turns off participation of this package in its contained elements namespaces.
Ada_CG.Package.DeclarationPosition	Applicable to nested packages only. Determines declaration position relatively to the section (public part of spec, private part of spec, body) it is declared in and to the “virtual” location of the class record type if it is/was declared in this section.
Ada_CG.Package.ImplementationEpilog	Adds an epilog to the package body.
Ada_CG.Package.ImplementationPragmas	Holds user-defined pragmas to generate in body
Ada_CG.Package.ImplementationPragmasIn ContextClause	Holds user-defined pragmas to generate in context clause of body
Ada_CG.Package.ImplementationProlog	Adds a prolog to the package body.
Ada_CG.Package.InitializationCode	Adds initialization code in the package body
Ada_CG.Package.IsNested	Indicates if the package is to be generated as a nested package
Ada_CG.Package.IsPrivate	Indicates if the package is to be generated as a private package

Element	
<Property>	<Description>
Ada_CG.Package.NestingVisibility	Indicates where in the nesting package the specification of the nested package should be generated
Ada_CG.Package.Renames	Holds the name of the package this package is renaming
Ada_CG.Package.SpecificationEpilog	Adds an epilog to the package specification.
Ada_CG.Package.SpecificationPragmas	Holds user-defined pragmas to generate in spec
Ada_CG.Package.SpecificationPragmasInContextClause	Holds user-defined pragmas to generate in context clause of spec
Ada_CG.Package.SpecificationProlog	Adds a prolog to the package specification.
Ada_CG.Relation.ObjectInitialization	Determines what kind of initialization shall occur for package instances
Ada_CG.Package.UseAda83Framework	If "True", then generated code for statecharts, events and guarded operations and attributes will use Ada 83 constructs. If "False", Ada 95 constructs will be used.
CG.Package.UseAsExternal	Turns off generation of this package and of its contained elements.
CG.Package.GeneratePackageCode	Turns off generation of this package, but not of its contained elements.
CG.Type.Animate	Turns of animation of this package.
SPARK.Package.ExaminerLevelBody	Sets Examiner level for this package body
SPARK.Package.ExaminerLevelSpec	Sets Examiner level for this package spec
Port	
Ada_CG.Port.Generate	Turns off generation.
Project	
Ada_CG.File.ImplementationFooter	Overrides the generated implementation footer.
Ada_CG.File.ImplementationHeader	Overrides the generated implementation header
Ada_CG.File.SpecificationFooter	Overrides the generated specification footer
Ada_CG.File.SpecificationHeader	Overrides the generated specification header
Relation	
Ada_CG.Relation.BidirectionalRelationsScheme	<p>Controls how bidirectional relations are implemented</p> <p>Possible values are :</p> <ul style="list-style-type: none"> IntermediateParentClasses SubtypingAndRenaming (default)

Element	
<Property>	<Description>
Ada_CG.Relation.InitializeComposition	Controls how a composition relation is initialized. Possible values are : <ul style="list-style-type: none"> • InInitializer (default) • InRecordType • None
Ada_CG.Relation.IsAliased	Determines if relation is aliased
Ada_CG.Relation.Visibility	Sets the visibility of the relation getter and setter.
Ada_CG.Relation.ObjectInitialization	Determines what kind of initialization shall occur for instances
CG.Relation.Generate	Turns off generation of this relation.
CG.Relation.GenerateRelationWithActors	Determines generation when target is an actor.
CG.Relation.GetGenerate	Turns off generation of the relation getter.
CG.Relation.Implementation	Chooses the relation implementation style. When set to "User", no accessors are generated for the relation.
CG.Relation.SetGenerate	Turns off generation of the relation setter.
OMContainers.Access.AccessKind	For relations with the CG.Relation.Implementation property set to "Default", this property controls whether a regular or a class-wide access type will be used as the relation implementation type.
OMContainers.User.CType	If CG.Relation.Implementation is set to "User", the contents of this property will be used as the type to generate in the declaration of the record component holding the relation. This property supports the following Rhapsody keywords : <ul style="list-style-type: none"> ○ cname : returns the name of the association end ○ RelationTargetType : returns the name of the type of the relation target
Type	
Ada_CG.Type.AccessTypeUsage	For typedef types, indicates if the basic type is referred to as an access type, a class-wide access type or a regular type or not. Effective only if the basic type is a class.
Ada_CG.Type.AnimEnumerationTypeImage	Activates usage of Image attribute for enumerated types when using animation
Ada_CG.Type.AnimSerializeOperation	Overrides generated serialize operation for animation

Element	
<Property>	<Description>
Ada_CG.Type.DeclarationPosition	Determines type declaration position relatively to the section (public part of spec, private part of spec, body) it is declared in and to the “virtual” location of the class record type if it is/was declared in this section.
Ada_CG.Type.Final	Makes the type record non-“tagged”. Only applicable to struct types.
Ada_CG.Type.LanguageMap	Holds the Ada declaration for Rhapsody Language Independent Types
Ada_CG.Type.Visibility	Sets the visibility of the type.
CG.Type.Animate	Turns off the animation of this type.
CG.Type.UseAsExternal	Turns off generation of this type.
Statechart	
Ada_CG.Statechart.HistoryConnectorDepth	Controls the depth of history connectors. Only effective when using the Ada 95 Behavioral framework. History connectors are always shallow when using the Ada 83 framework.

Appendix B: Tags for IBM® Rational® Rhapsody® Developer for Ada

Element	
<Profile>.<MetaType>.<Tag>	<Description>
Attribute	
AdaCodeGeneration.Attribute.generatePragmaAtomic	Generates an “atomic” pragma for this attribute/variable. Only works for package variables or static class attributes.
AdaCodeGeneration.Attribute.generatePragmaVolatile	Generates a “volatile” pragma for this attribute/variable. Only works for package variables or static class attributes.
AdaCodeGeneration.Attribute.representationClauses	Contains the representation clauses to be generated for this attribute. Note that this is only applicable for attributes defined on a package and for static attributes defined on classes.
SPARK.Attribute.IsAbstract	Controls the generation of the Ada declaration for the attribute. Used to model abstract own variables
SPARK.Attribute.IsInitialized	If this tag is set to true, the attribute name will be added to the initialization annotation of the class or package it is defined in
SPARK.Attribute.OwnMode	If this attribute is an own variable, controls the mode used in the own annotation of the class or package it is defined in
SPARK.Attribute.OwnKind	Controls the participation of the own attribute to the own annotation of the class or package it is defined in
Class	
AdaCodeGeneration.Attribute.generatePragmaAtomic	Generates an “atomic” pragma for this class record type.
AdaCodeGeneration.Class.generatePragmaElaborateBody	Generates an “elaborate” pragma for this class
AdaCodeGeneration.Class.generatePragmaPreelaborate	Generates a “preelaborate” pragma for this class
AdaCodeGeneration.Class.generatePragmaPure	Generates a “pure” pragma for this class
AdaCodeGeneration.Attribute.generatePragmaVolatile	Generates a “volatile” pragma for this class record type.
AdaCodeGeneration.Class.representationClauses	Contains the representation clauses to be generated for this class
SPARK.Class.HideBody	Controls the generation of the hide annotation for this class package body

Element	
<Profile>.<MetaType>.<Tag>	<Description>
SPARK.Class.HideElaborationCode	Controls the generation of the hide annotation for this class package elaboration code
SPARK.Class.HidePrivatePart	Controls the generation of the hide annotation for this class package private part
SPARK.Class.Inherit	Contains the comma separated list of packages this class is inheriting from
SPARK.Class.Initializes	Contains the comma separated list of own variables this class is initializing
SPARK.Class.OwnSpec	Contains the list of own variables (with optional modes and types) to be generated in the own annotation of this class package spec
SPARK.Class.OwnBody	Contains the list of own variables (with optional modes and types) to be generated in the own annotation of this class package body
Dependency	
SPARK.Dependency.Inherit	Indicates if this <<usage>> dependency shall also generate an inherit annotation
SPARK.Dependency.GlobalMode	Indicates the mode for the supplier variable of this <<SPARK_Global>> dependency
Operation	
SPARK.Operation.DerivesBody	Enter the dependency clauses for the operation body derives annotation in this tag.
SPARK.Operation.DerivesSpec	Enter the dependency clauses for the operation specification derives annotation in this tag.
SPARK.Operation.GlobalBody	Enter the global variables and their usage mode for this operation body in this tag
SPARK.Operation.GlobalSpec	Enter the global variables and their usage mode for this operation specification in this tag
SPARK.Operation.HideBody	set this tag to true if you want the body for this operation to be hidden from the examiner
SPARK.Operation.PostConditionBody	Use this tag to capture the postcondition annotation for a procedure body or the return annotation for a function body
SPARK.Operation.PostConditionSpec	Use this tag to capture the postcondition annotation for a procedure specification or the return annotation for a function specification
SPARK.Operation.PreConditionBody	Use this tag to capture the precondition annotation for a procedure or function body
SPARK.Operation.PreConditionSpec	Use this tag to capture the precondition annotation for a procedure or function specification

Element	
<Profile>.<MetaType>.<Tag>	<Description>
Package	
AdaCodeGeneration.Package.generatePragmaElaborateBody	Generates an “elaborate” pragma for this package
AdaCodeGeneration.Package.generatePragmaPreelaborate	Generates a “preelaborate” pragma for this package
AdaCodeGeneration.Package.generatePragmaPure	Generates a “pure” pragma for this package
SPARK.Class.HideBody	Controls the generation of the hide annotation for this package body
SPARK.Class.HideElaborationCode	Controls the generation of the hide annotation for this package elaboration code
SPARK.Class.HidePrivatePart	Controls the generation of the hide annotation for this package private part
SPARK.Class.Inherit	Contains the comma separated list of packages this package is inheriting from
SPARK.Class.Initializes	Contains the comma separated list of own variables this package is initializing
SPARK.Class.OwnSpec	Contains the list of own variables (with optional modes and types) to be generated in the own annotation of this package spec
SPARK.Class.OwnBody	Contains the list of own variables (with optional modes and types) to be generated in the own annotation of this package body
Type	
AdaCodeGeneration.Attribute.generatePragmaAtomic	Generates an “atomic” pragma for this type.
AdaCodeGeneration.Attribute.generatePragmaVolatile	Generates a “volatile” pragma for this type.
AdaCodeGeneration.Type.representationClauses	Contains the representation clauses to be generated for this type

Appendix C: Stereotypes for IBM® Rational® Rhapsody® Developer for Ada

Stereotype	Applicable to	<Description>
<Profile>		
abstract	Class	
AdaProtectedObject	Class	
AdaProtectedType	Class	
AdaTask	Class	
AdaTaskType	Class	
Container	Package	Indicates that a package does not contribute to the namespace of its contained elements.
Discriminant	Attribute	Only applicable to struct attributes and to class instance-level attributes. This stereotype specifies that the attribute shall be generated as a record type discriminant instead of a record type component.
entry	Operation	
entrypoint	Class	
HSER	Operation	Stands for Highly Synchronous
Interface	Class	
LSER	Operation	Stands for Loosely Synchronous
Parent_Instantiation	Dependency	Indicates that the instantiation DI for a derived template class D depends on a given instantiation BI of the base class B of D
Renames	Dependency	Indicates that a client variable, operation, class or package is just a renaming of its supplier.
separate	Operation	
Singleton		

TaskDefaultAction	Operation	
Usage	Dependency	
SPARK		
INFORMED_Boundary_Variable_Package	Class Package	
INFORMED_Main_Program	Class	
INFORMED_Type_Package	Class Package	
INFORMED_Utility_Package	Package	
INFORMED_Variable_Package	Class Package	
SPARK_Global	Dependency	
SPARK_Initializes	Dependency	
SPARK_Proof	Actor Class Operation Type	
SPARK_Refined_By	Dependency	

