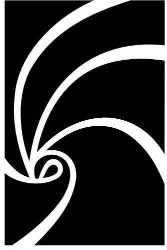


Rational Rose 2000e
Using Rose J

Copyright © 1998–2000 Rational Software Corporation.
All rights reserved.

Part Number: 800-023323-000

Revision 3.0, March 2000, (Software Release 2000e)

This document is subject to change without notice.

GOVERNMENT RIGHTS LEGEND: Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in the applicable Rational Software Corporation license agreement and as provided in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (Oct 1988), FAR 12.212(a) (1995), FAR 52.227-19, or FAR 52.227-14, as applicable.

Rational, the Rational logo, Rational Rose, ClearCase, and Rational Unified Process are trademarks or registered trademarks of Rational Software Corporation in the United States and in other countries. All other names are used for identification purposes only and are trademarks or registered trademarks of their respective companies.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Contents

Contents iii

List of Tables vii

Preface ix

How This Manual Is Organized ix

Related Documentation x

Online Help and Manuals x

Chapter 1 Introduction to Rational Rose J 1

What is Rational Rose J? 1

About Java API Classes and Frameworks 2

About Forward Engineering in Rational Rose J 3

About Reverse Engineering in Rational Rose J 4

About the Integration with Microsoft Visual J++ 5
- Starting Rose from Visual J++ 5

About the Integration with IBM's VisualAge for Java 6
- Setting Up the Link 6
- How the Link Works 6
- How to Enable the Link 7
  - Initializing the VisualAge side of the link 7
  - Ending the link 7

Establishing the link from Rational Rose 7

Chapter 2 **How Rational Rose J Models Java Elements 9**

About Java Elements in a Rational Rose Model 9
How Model Properties and Specifications Affect Your Model 10

Java Class Path 14

Java Classes 17
Making a Rational Rose Class a Java Class 18
Supported Java Semantics for Classes 20

Java Variables (Fields) 22
Variables with Primitive Types 22
Changing String Variables to Attributes 22
Variables with Reference Types 23
Creating Variables through Associations 24
Creating Variables with User Data Types 26
Supported Java Semantics for Variables (Fields) 27

Arrays and Vectors 29

Java Methods 33
Supported Java Semantics for Methods 33

Java Extends 36

Java Implements 37

Java Interfaces 38

Java Nested and Inner Classes 39

Java (.java) Files 40
Java Components and Code Generation 42
Java Components and Reverse Engineering 43

Java Packages 44
Java Packages and Code Generation 44
Java Packages and Reverse Engineering 45

Java Imports 46

Java Beans 46
Read/Write Property 47
Support Individual Change Management 47
Property Types 47
Simple 47
Bound 48
Constrained 49
Javadoc and Comment Text 51

Chapter 3 Forward Engineering with Rational Rose J 55

About the Steps You Follow 55
Assign Java Classes to Java Components in Your Model 55
Check Syntax 56
Check the Class Path 56
Set the Project Properties that Affect Code Generation 56
Back Up Your Source 57
Generate Java Source Code from Your Model 58
View (Browse) and Extend the Generated Source 58
How Controlled Units Affect Code Generation 59
Mapping Components for Code Generation 60
Generating Java Source from a Component Diagram 61
Generating Java Source from a Class Diagram 61
Generating Code for Visual J++ 62
Generating VJ++ Code from a Rose Model 62
Viewing and Completing Java Source 62
Using the BuiltIn Editor 63
Browsing Java Source 64
Completing Generated Java Source 65

Chapter 4 Reverse Engineering with Rational Rose J 67

About Reverse Engineering 67
Reverse Engineering Java Source 68

Reverse Engineering from Visual J++ 69

Appendix A Forward Engineering WalkThrough 71

Introduction 71

Walkthrough 72
- Create a Class and Add a Method 72
- Create a Component Package and Component 73
- Assign the Class to the Component 74
- Edit the New Method 75
- Set the Class Path 77
- Generate Java 78
- View Java Source 79
- Edit Java Source 80
- Compile and Run 80

Appendix B Java to Rational Rose Mapping Quick Reference 81

Appendix C Rational Rose J Model Properties 83

Introduction 83

Project Properties 84

Class Properties 90

Operation Properties 91

Attribute Properties 92

Component (Module Body/Module Specification) Properties 94

Role Properties 95

Index 97

List of Tables

Table 1 Shortcut Keys for the BuiltIn Editor 64
Table 2 Java to Rational Rose Mapping Quick Reference 81
Table 3 Project Properties 84
Table 4 Class Properties 90
Table 5 Operation Properties 91
Table 6 Attribute Properties 92
Table 7 Component Properties 94
Table 8 Role Properties 95

Preface

This guide, *Rational Rose 2000e, Using Rose J*, is for anyone who wants to use Rational Rose to model and generate Java constructs or reverse engineer Java source.

It assumes that you are familiar with Java concepts and constructs, and that you are comfortable with basic Rational Rose concepts and procedures.

If you need to learn to use Rational Rose, you should run the Rational Rose tutorial included on your product CD.

How This Manual Is Organized

This manual contains the following four chapters and three appendices:

- **Chapter 1—**Introduction to Rational Rose J

  Provides an overview of features and basic forward, reverse and round trip engineering concepts as they apply to Rational Rose J.
- **Chapter 2**—How Rational Rose J Models Java Elements

  Provides a detailed mapping between Java and Rational Rose constructs.
- **Chapter 3**—Forward Engineering with Rational Rose J

  Provides detailed instructions for generating Java source from elements in a Rational Rose model.
- **Chapter 4**—Reverse Engineering with Rational Rose J

  Provides detailed instructions for reverse engineering Java source and byte code into in a Rational Rose model.

- **Appendix A—**Forward Engineering WalkThrough

  Provides a use case that creates a Java class and component and generates Java source from the component.
- **Appendix B**—Java to Rational Rose Mapping Quick Reference

  Provides a quick reference table that maps Java and Rational Rose constructs for forward or reverse engineering.
- **Appendix C—**Rational Rose J Model Properties

  Describes all Rational Rose J properties, the model elements to which they apply, and their default values.

Related Documentation

The information in this guide is also provided in the form of online help. In addition, you will find context-based online help as you complete procedures and work in the various Rational Rose J dialog boxes.

After installation and before you begin using Rational Rose J, please review any **Readme.txt** files and **Release Notes** to ensure that you have the latest information about the product.

For additional resources, refer to the *Rational Rose 2000e, Using Rose* guide and online help. If you are new to Rational Rose, visual modeling, or the Unified Modeling Language (UML), you may also want to read the book, *Visual Modeling with Rational Rose and UML*, which is included in the Rose documentation kit.

Online Help and Manuals

Rational Rose J includes comprehensive online help with hypertext links and a two-level search index.

In addition, you can find all of the user manuals online. Please refer to the **Readme.txt** file (found in the Rational Rose installation directory) for more information.

Chapter 1

Introduction to Rational Rose J

What is Rational Rose J?

Rational Rose J is the Java add-in for Rational Rose. As such, it provides:

- Support for model evolution from analysis to design
- Generation of Java source code from a model
- User extensible class path within the model
- Support for zip, jar and cab files in the class path
- Ability to drag and drop java, class, zip, jar and cab files into a model diagram for quick reverse-engineering
- Automatic decompression for zip, jar, and cab files when reversed-engineered into a model
- Design, modeling, and visualization of Java constructs, including packages, classes, interfaces, imports, inheritance, fields, methods, and modifiers
- User-controlled generation of default constructors, finalizers, and static initializers, as well as user-defined generation of field-name prefixes
- Smart component mapping on code generation
- Support for a default package
- Improved support for inner and anonymous classes
- Bean property generation (get/set methods)
- Built in color-coded editor
- Support for Java API frameworks

About Java API Classes and Frameworks

Rational Rose provides Frameworks that enable you to bring the class libraries of the Java API into a model. When you load these base classes into your model you can use them to derive your own classes.

There are two ways to bring a Java Framework into a model:

- **For new models**: You can load Java API classes into a new model by using a Java Framework. If the Framework Add-In is active, each time you start Rose or open a new model the Framework dialog appears. (Click **Add-Ins > Add-In Manager** to check the status of the Framework add-in.)
- **For existing models**: The framework elements are also available as controlled units in the directory:

  `Rose\framework\frameworks\shared components`

  Use **File > Units > Load** to load the appropriate controlled units into your model.

The Java frameworks include:

- JDK Framework. This is a reverse engineered model of current JDK class files. It contains all of the classes in the Java packages and some of the classes in the Sun package.
- JEnterprise Framework. This is a reverse-engineered model of enterprise APIs. Use this framework as a starting point for developing models that use Enterprise JavaBeans™, Java Transaction Service™, or Java Naming & Directory Interface™ (JNDI) components. It also contains the same classes as the JDK Framework. These packages are included to correctly resolve all dependencies for classes in the com.java package. Other packages included in this framework include:
  - javax.ejb, the EJB Framework Package. EJB fully insulates business application developers from the complexity of the enterprise infrastructure (the plumbing).
  - javax.jts, the Java Transaction Service Package. JTS is a low-level API used by sophisticated transactional applications.
  - javax.naming, the JNDI Framework Package. The Java Naming & Directory Interface provides Java applications with multiple naming and direct services in the enterprise.

- JFC Framework. This is a reverse-engineered model of the JFC class files from swingall.jar. The framework contains all of the classes found in the com.java package. It also contains the JDK class libraries in order to resolve all class dependencies for the classes in the com.java package.

Note that since the framework classes are intended to be starting points from which you can build your own classes, they do not have private attributes, relationships, or operations.

For more information on activating and deactivating add-ins and using frameworks, check the Help index or see the *Using Rose* manual.

About Forward Engineering in Rational Rose J

Forward engineering is the process of generating Java source from one or more classes, packages, or components in a Rational Rose model.

Forward engineering in Rational Rose J is component-centered. This means that the Java source generation is based on the component specification rather than on the class specification.

This does not mean, however, that you must work only in component diagrams in order to generate Java source. Instead, you can create a class and then assign it to a valid Java component in the browser, effectively creating the required component from your class.

When you forward engineer a class or package, its characteristics are mapped to corresponding Java constructs. For example, classes in Rational Rose are forward engineered as Java classes, Rational Rose components are forward engineered as Java compilation units (.java files), etc.

Also, when you forward engineer a package, a .java file is generated for each component belonging to the package. Each component's .java file will contain the definitions for any classes assigned to that component.

For complete mapping information, refer to:

- How Rational Rose J Models Java Elements (Chapter 2)
- Java to Rational Rose Mapping Quick Reference (Appendix B)

Note that Rose J offers an auto synchronization feature that automatically initiates code generation any time you create or modify any Java construct in your model. You can enable the feature through the Java Project Specification (**Tools** > **Java** > **Project Specification** > **Detail**).

For more information on forward engineering, refer to:

- Generating Java Source from a Component Diagram (Chapter 3)
- Generating Java Source from a Class Diagram (Chapter 3)
- Forward Engineering WalkThrough (Appendix A)

About Reverse Engineering in Rational Rose J

Reverse engineering is the process of analyzing Java source code, mapping it to Rational Rose classes and components, and storing these classes and components in a Rational Rose model.

You can use Rational Rose J to reverse engineer Java source from:

- Java source code (.java files)
- Java bytecode (.class files)
- .cab, .jar, and .zip files

You can drag and drop .java, .class, .cab, .jar, and .zip files from another source (such as Microsoft Explorer) into a model. When you do this, Rational Rose reverse engineers the files for you.

For complete mapping information, refer to:

- How Rational Rose J Models Java Elements (Chapter 2)
- Java to Rational Rose Mapping Quick Reference (Appendix B)

For more information on reverse engineering, refer to:

- Reverse Engineering Java Source (Chapter 4)

About the Integration with Microsoft Visual J++

Rose J offers a seamless integration with Microsoft Visual J++. If you have Rose running, you can activate Visual J++ simply by setting the **Virtual Machine** property in the Rose J project specification to **Microsoft**, then start generating code from your model. If Visual J++ is not running when you generate code, Rose will automatically start it. The code you generate either becomes part of a new VJ++ project or it updates an existing project.

Alternatively, Visual J++ offers two ways to work with Rose:

- You can simply open Rose and a blank model, or
- You can reverse engineer your VJ++ project and update your Rose model.

Both options are available from the VJ++ **Tools** menu. In both cases, Visual J++ can launch Rose if it's not already running.

Note:

1. *Many files in a VJ++ project inherit from one or more of the WFC classes that Visual J++ provides. When you install Visual J++, these classes are typically accessible in the VJ++ environment. For Rose J to reverse engineer files that use these classes, you must first create an externally accessible copy of the files on your hard drive. To do this, enter the following command at a DOS prompt:*

```
clspack -auto
```

2. *Currently, there is no requirement that Visual J++ 6.0 be installed on your system before you install the Rose to VJ++ link. However, if you install Visual J++ after the link, you will need to register the vjrose dll. To do this, go to the \Rose\java directory and from a DOS prompt, enter the following:*

```
Regsvr32 vjrose.dll
```

Starting Rose from Visual J++

In addition to code generation and reverse engineering (described in Chapters 3 and 4 respectively), Visual J++ offers the option of starting Rose without opening a specific model or reverse engineering VJ++ source. In Visual J++, click **Tools** > **View Rose Model**. If Rose isn't active, this starts it but doesn't load a specific model.

About the Integration with IBM's VisualAge for Java

IBM and Rational Rose support a link between Rose J and VisualAge for Java™. This link enables you to:

- Generate code for a VisualAge project directly from a Rose model
- Reverse engineer code from a VisualAge project into a Rose model

Setting Up the Link

In order for the two tools to communicate you need to:

- Activate the link on the VisualAge side, then
- Set the **Virtual Machine** property in Rose to **IBM (Tools > Java > Project Specification > Detail)**

Once the two environments are set up you can move code between them by generating code from a Rose model or by reverse engineering code from VisualAge into a Rose model.

Note that you cannot create new VisualAge projects from Rose. The link works only with existing projects.

How the Link Works

When you activate the link between the two environments, you enable Rose J to move code in and out of a VisualAge project. VisualAge is a passive partner in the link since you initiate both code generation and reverse engineering from Rose J.

When you generate code from a Rose model, you follow the same code generation procedures that you would follow for any Java classes in a model **(Tools > Java > Generate Code)**. However, when Rose J creates the code, it stages it first in the directory \Rose\java\YourProjectName, then VisualAge imports the code from this file into its own repository.

Similarly, when you reverse engineer a VisualAge project into Rose, you use the standard Rose J reverse engineering dialog. As with code generation, the project is first written to a file in the \Rose\java directory, then the code from the file is reverse-engineered into your model.

Note that in order to reverse engineer files that import from the JDK, JFC or any other library, you must have a reference to those libraries in your classpath environment variable. For the libraries included with VisualAge, you can add the following to your class path:

```
IBMJavaInstallationDirectory\ide\PROGRAM\LIB\CLASSES.ZIP
```

How to Enable the Link

To establish the link between Rose J and VisualAge for Java you need to first initialize the link on the VisualAge side, then select a project on the Rose J side.

Initializing the VisualAge side of the link

1. In VisualAge for Java, click **File > Quick Start**.
2. From the **Quick Start** dialog, click **Basic** and the **RoseLink Plugin Toggle**. Click **OK**.

A message confirms that the link is successful.

Ending the link

1. In VisualAge for Java, click **File > Quick Start**.
2. From the **Quick Start** dialog, click **Basic** and the **RoseLink Plugin Toggle**. Click **OK**.
3. A message notifies you that the link is active and prompts you whether you want to terminate it. Click **Yes** to end the link.

Establishing the link from Rational Rose

1. Set the **Virtual Machine** property to **IBM (Tools > Java > Project Specification > Detail)**.
2. Click **Tools > Java > IBM VisualAge for Java Project**.
3. Select a VisualAge Project from the list and click **OK**.

Note that when you select a VisualAge project, Rose creates a new Project model property setting called **VAJavaProject**. This model property is set to the project name you selected.

Chapter 2

How Rational Rose J Models Java Elements

About Java Elements in a Rational Rose Model

A good starting point for learning about Rational Rose J is to understand how it models the Java elements you are most familiar with. This chapter describes how the most commonly used Java constructs are visualized in a Rational Rose model, including:

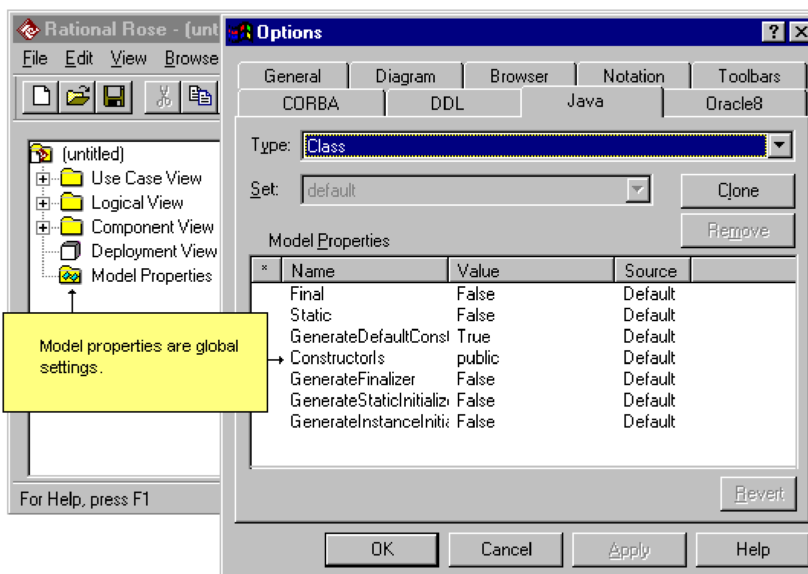
- Class path
- Java classes, variables, and methods
- Arrays and vectors
- Nested and inner classes
- Implements and extends relationships
- Interfaces
- .java files
- Packages and imports
- Java Beans
- Javadoc comments

In addition, this chapter explains how to view and set Java-specific settings that control how these constructs are modeled.

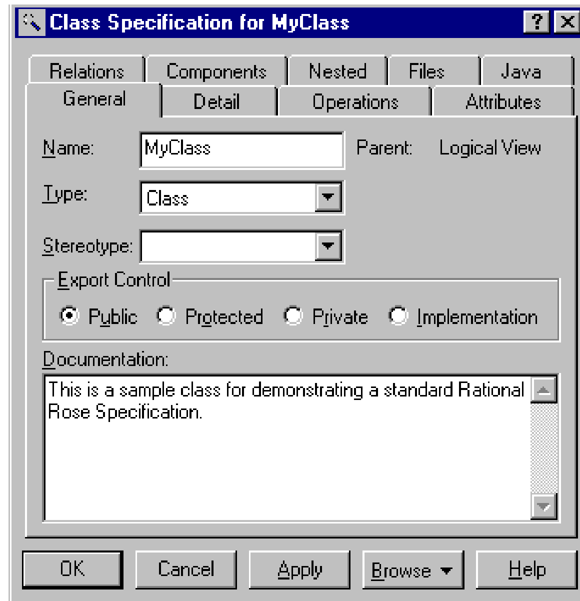
How Model Properties and Specifications Affect Your Model

There are two generic mechanisms that Rational Rose J uses to control how your model elements behave and how Java source is generated for them:

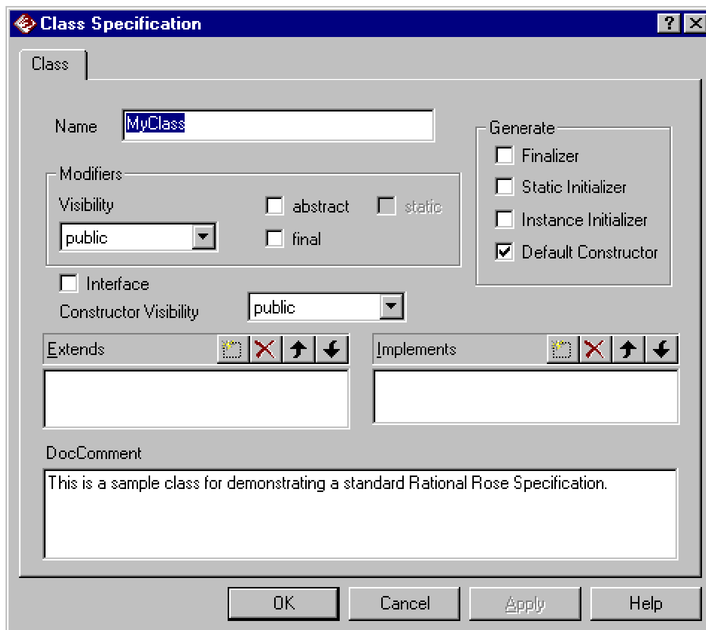
- **Model properties.** These properties provide global settings for a project and its classes, attributes, operations, components, and roles.



- **Standard Rational Rose specifications**, which control individual classes, attributes, operations, components, and roles.



In addition to these, Rational Rose J offers an alternative set of context-based custom **Java specifications** that extract settings from both the standard Rational Rose specifications and the model property settings. For example:



These custom specifications enable you to display and modify settings in a single dialog that uses Java terms rather than Rational Rose/UML terms. Any change you make on a Java specification updates the corresponding Rational Rose specification or model property.

The Java specifications are:

- **Project Specification** - This consists of the **Class Path**, **Detail**, and **Style** tabs.

  The **Class Path** tab enables you to set up the CLASSPATHs that Rose J will use when you generate code from a model or reverse engineer existing source. (Setting the CLASSPATH is a required step for both code generation and reverse engineering.)

  The **Detail** tab enables you to establish default values for various model properties and lets you control the behavior of Rose J features. For example it lets you:

  - Set default return types for methods, default data types for variables, and the editor to use for browsing generated source code.
  - Enable automatic synchronization which causes Rose J to automatically generate code whenever you create, delete, or modify various Java constructs in your model.
  - Specify the platform or IDE you're using (**Virtual Machine** setting). The Sun Microsystems JDK is the default, but Rose provides integration with both Microsoft Visual J++ and IBM's VisualAge for Java.
  - Disable the Rose J specifications (and enable the standard Rose specification as the default specification).

  The **Style** tab lets you choose which type of comment you implement in the Java code that you generate. You can select the standard Rose comment style, the standard Java comment style, or you can generate Javadoc tags for creating HTML-based documentation. It also lets you control how generated code is indented and whether opening braces should start on a new line.

  You access the Project Specifications from **Tools** > **Java** > **Project Specification**.

- **Class Specification** - This specification lets you add Java class modifiers, generate constructors and finalizers, create extends and implements relationships, and set the class access level (visibility).
- **Field Specification** - This specification lets you display and modify how Rose models Java variables. It lets you specify modifiers, access level, data type, initial value, and container class. It also lets you create Java beans by setting bean-specific properties.

- **Method Specification** - Using this specification, you can set a method's modifiers, return type, and access level, as well as create argument and throws (exception) relationships.
- **Component Specification** - Using this specification, you can view and set the import specifications for the component, version control tags that are included in the header of a .java file, copyright notices, and comment text.

When a component or class becomes a Java-specific component or class, the Java specification becomes the default specification that appears when you double-click on any model element (either in the browser or a diagram) associated with the component and class. If a class or component is not associated with a particular language, the standard Rose specification is the default. (Note that you can disable the Rose J specifications via the **Always Show Standard Specification Dialog for Classes** field on the Java Project Detail Specification.)

There are two ways that a class or component becomes a Java-specific class or component:

- You use the Notation model property that establishes Java as your default language. When you do this, every class or component that you create automatically becomes a Java class or component.
- You assign a class to a component whose language is set to Java. In this case, you create a component, set its language to Java, then assign a class to the component. Here, the language assignment is local to the component and its associated classes. Your global language setting is something other than Java (usually Analysis).

Note that you can't use Java specifications to create model elements. You can only display and modify the settings for existing elements. To create a model element (such as a class, attribute, method, component, etc.), use the shortcut menu or the standard Rose specifications.

Java Class Path

Rose J requires class path settings for the JDK API classes, as well as for your own user libraries. A class path is a system environment variable and is not specific to Rational Rose. Various Java tools and applications (including the Java Virtual Machine) rely on your system's CLASSPATH settings to resolve references to user-defined classes, specifically to package and import references.

For reverse engineering, Rose J must have a class path to the JDK class library. For example, depending on the version of the JDK you're using, you'll need a CLASSPATH setting to classes.zip, rt.jar, or other appropriate library file. (Check your JDK documentation for current names and locations.) To illustrate, a sample path to the rt.jar file in JDK 1.2.1 would be x:\jdk1.2.1\bin\jre\rt.jar.

When setting up your class path for your user libraries, consider:

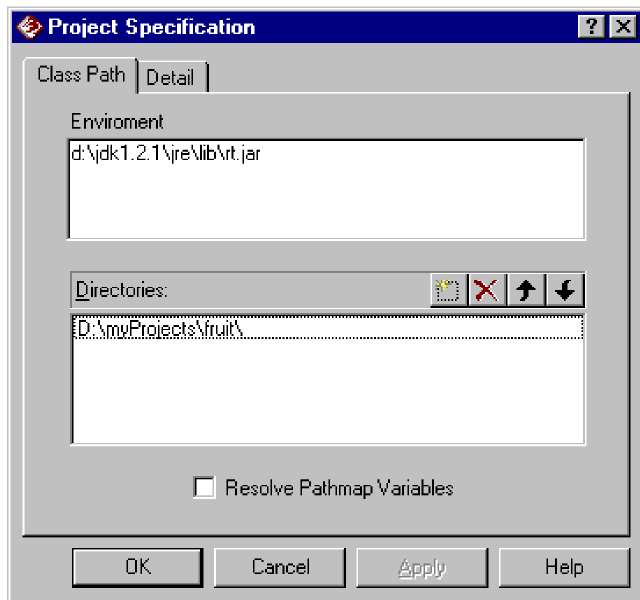
- Where is the file? Specifically, what is the fully qualified path name?
- What are the package or import statements in the source?

You must have a directory in your CLASSPATH setting for the leftmost directory in a package/import statement. Rose converts each package/import statement into a path in your file system, then adds each directory in your CLASSPATH until it makes up a fully qualified path to the class file.

You can use the Class Path tab of the Java Project Specification to dynamically set up a specific class path tailored to your current model.

To display the Java Project Specification, click **Tools > Java > Project Specification**.

For example:

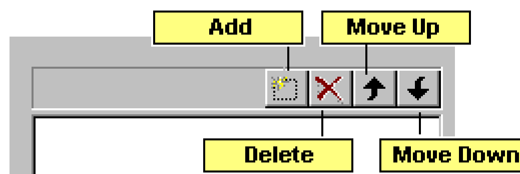


When you start Rational Rose, Rose J retrieves the current CLASSPATH environment variable settings from your system and lists them in the Environment section of the Java Project Specification. (This section is read-only.)

Note: *If you change your system's class path environment variable, you will want Rose J's Java Project Specification to change as well. If your system runs Windows NT, restarting Rational Rose updates the specification. If you run Windows 9x you need to reboot your system to update the specification.*

You can use the **Directories** section of the **Class Path** tab if you want to dynamically add directories that extend your class path. Any settings you add here are saved with your model. (They aren't added to your class path environment variable.) You can add as many directories as you need, when you need them.

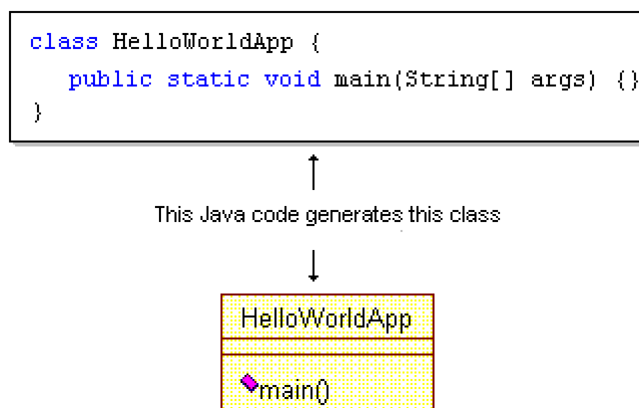
To add or modify a path in this list, use the control buttons:



If you use path map variables in your model, select the **Resolve Pathmap Variables** checkbox to change path map variables to physical path (directory) names.

Java Classes

Rational Rose J models Java classes as Rational Rose classes:



You create a Java class in a model using the same techniques you use to create any class in Rational Rose. Naturally, if you've reverse-engineered Java classes into a model, Rational Rose J generates model classes based on your code, including the components and packages that reflect how your classes are organized.

Making a Rational Rose Class a Java Class

A Rational Rose class takes on the personality of a Java class in three ways:

- Your default notation for your model is set to Java (**Tools** > **Options** > **Notation** > **Default Language** > **Java**). In this case, *any* new class you create will be a Java class.
- Your default notation is not Java (for example, it's the **Analysis** default), but you assign a class to a component whose **Language** field you've set to Java. As described in detail later in this chapter, Rational Rose J uses Rational Rose components to model actual .java files.
- You reverse-engineered Java classes into a model.

If Java is not your model's default notation, or you don't assign a class to a Java component, the class can be affected in two significant ways:

- You won't be able to open a Java specification for the class or for any of its variables (fields), or methods. To set the properties you want, you will need to use the standard Rational Rose specifications.
- If you generate code for unassigned classes, Rational Rose J creates a new component for each class using the class name. (If you generate code for five unassigned classes, you end up with five new components and five new .java files.)

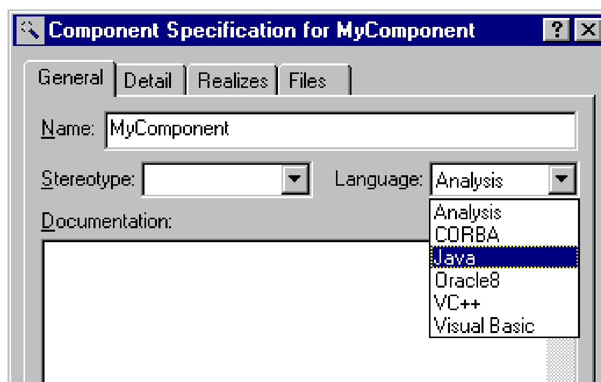
If your goal is to generate multiple classes to a single .java file, you will need to create the component and assign the classes to the component before you generate the code.

Note: *By default, the access control level for Rose classes is Public. If you create and assign multiple classes to a component, make sure the access control levels are correct. When you generate code from a model, Rose J enforces the Java rule that the name of a public class must match exactly the name of the component it is assigned to and there can be only one public class assigned to a component.*

There are several ways to create components and assign classes to them. Here is one method:

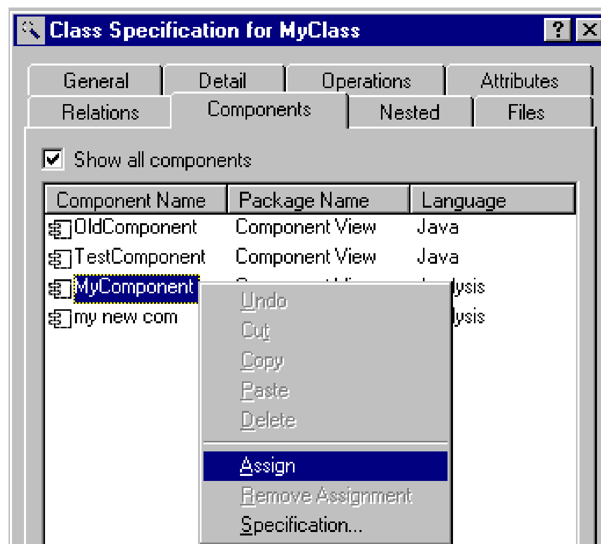
1. Click **Component View** in the browser, then click the right mouse button to display the shortcut menu.
2. From the shortcut menu click **New** > **Component** then key in a name for the new component.

3. In the browser, double-click the new component to open its Rose specification. Make sure the **Language** is set to **Java**:



Note: If you established Java as your model's default language (via the **Notation** *tab in Model Properties) all components you create in your model will automatically have Java set as the language.*

4. To assign a class to the component, you can click the class in the browser and drag it to the component. Or, you can open the **Components** tab in the Class Specification and use the shortcut menu to assign the class to the component from the list:



Supported Java Semantics for Classes

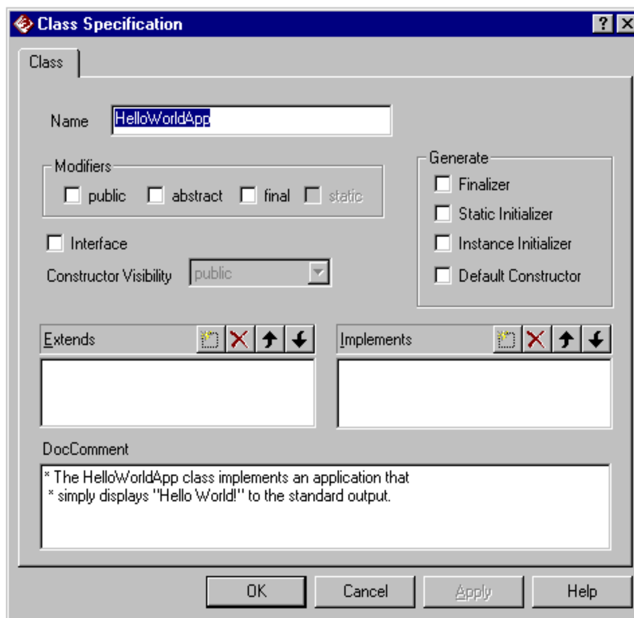
The Java semantics that Rational Rose J supports (and generates code for) include:

- **Modifiers**. These include public (always added by default), abstract, final, and static
- **Generators**. These are finalizer, static initializer, instance initializer, and default constructor. Of these, only the default constructor is automatically generated.
- **Constructor Visibility**. You can set the visibility to public, private, protected, or package. The default is public.
- **Extends**. You can indicate if a class extends another class.
- **Implements**. You can indicate if a class implements one or more interfaces.
- **DocComment**. You can annotate the class declaration. There are three types of comments you can generate: standard Rose format (the default), standard Java format, or Javadoc comments with specific tags for generating HTML-based documentation. See the description of Javadoc later in this chapter for details**.**

In addition, be aware of these conventions which Rational Rose J supports:

- Class names only need to be unique within the same Rational Rose component package.
- If a class you create in Rational Rose J is a public class, its associated component must have the same name, regardless of how many other non-public classes are also associated with the component. (For example, the public class Cats must be associated with a component called Cats. The file Rational Rose J creates for the code it generates will be named Cats.java.)

All Java semantics are available from the Java Class Specification:



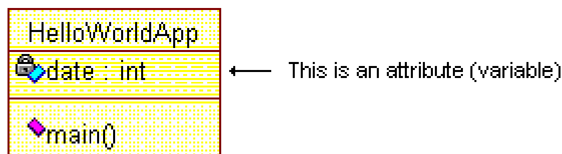
Note: *When you use the Java Class Specification to fine-tune a class definition, the specification enforces standard Java rules. (For example, it won't let you add the final modifier to an abstract class.) If you use the standard Rose specification instead, this checking is not done and errors may not be obvious unless you check syntax, either explicitly (you can select syntax checking from the* **Tools > Java** *menu) or as part of code generation.*

Java Variables (Fields)

Rational Rose J models a Java variable either as an **attribute** of a class or as a **role** in an association between two classes. The difference between the two is based on the type you want to declare for the variable.

Variables with Primitive Types

If the variable's type is a primitive type (int, byte, short, long, float, double, char, or boolean), the variable is modeled as an **attribute** of the class. For example:



The code generated for the class would look like this:

```
class HelloWorldApp {
   private int date;
}
```

There are several techniques for adding an attribute to a class, including selecting the class in a class diagram, clicking the right mouse button to display the shortcut menu, then clicking **New Attribute**.

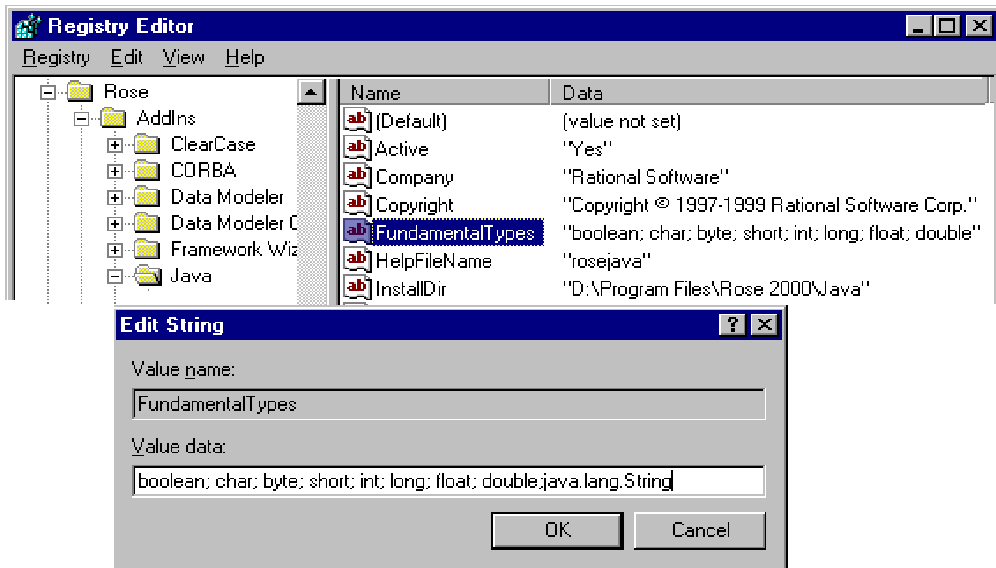
Once you've added an attribute to a class, you can display its Java Field Specification to make changes to its definition.

Changing String Variables to Attributes

By classic Java standards, String is a class, not a primitive data type. When you reverse engineer Java code, Rose J parses a String variable as a relationship to the class String in java.lang. If you prefer, you can have Rose J model all String variables as though String were a primitive data type by adding java.lang.String to the registry setting:

HKEY_LOCAL_MACHINE\SOFTWARE\Rational\Software\Rose\AddIns\Java\FundamentalTypes

For example:



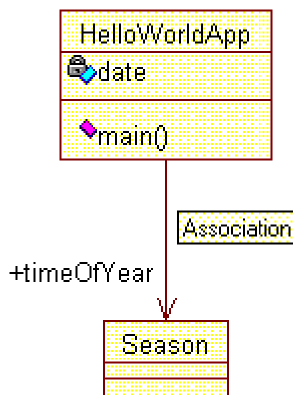
Variables with Reference Types

There are two ways to create a reference variable:

- By drawing an association between classes
- By creating an attribute and designating a class as the attribute's data type

Creating Variables through Associations

When a variable's type is a reference to another class, you model this relationship by creating an association between the class whose variable you are defining and the class that models the entity you are referencing. For example:



In this case, timeOfYear is a variable of the HelloWorldApp class. The variable's type is the class Season.

If you were to generate code for the HelloWorldApp class the variable definitions would appear as:

```
private int date;
   Season timeOfYear;
```

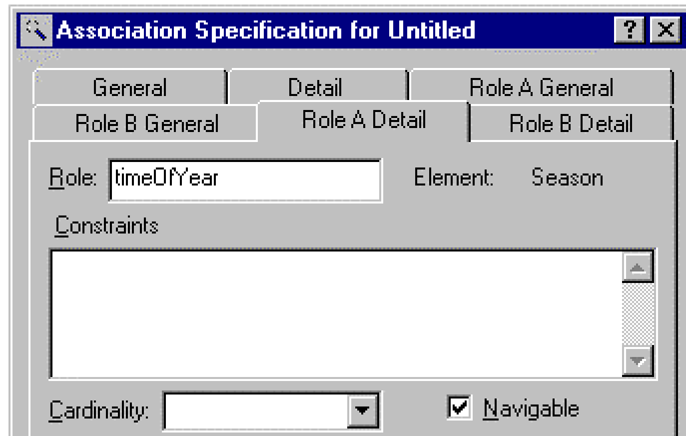
Note that Rational Rose J doesn't generate code for the association itself. Rather, it only generates code for a navigable role.

There are always two ends to an association, Role A and Role B. By default, when you draw an association it's one way (unidirectional). You start from Role B and end at Role A with Role A being the navigable role (also by default).

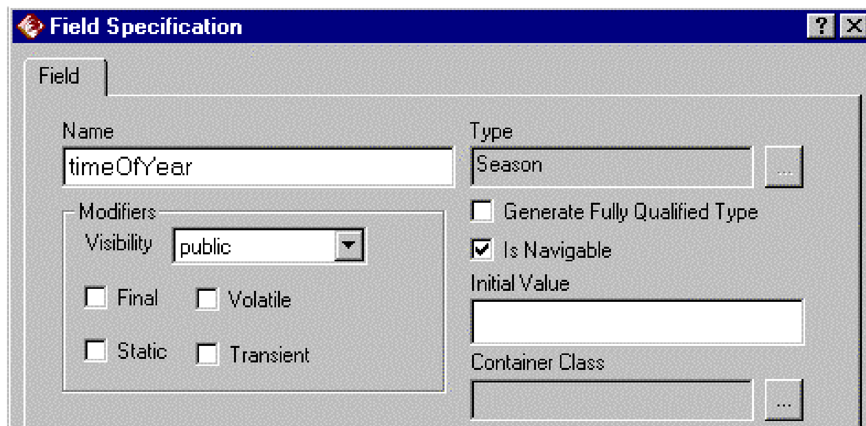
To generate a reference variable, you will always want the navigable role to be associated with the class you're referencing. Hence, in our example, HelloWorldApp is Role B (we started drawing the association from this class) Season is Role A and Season is the navigable role. This way, we can generate the Season reference variable for the HelloWorldApp class.

There are two places where you can view and set a role's navigability:

- On the **Role Detail** tab in the Association Specification:



- On the Rose J Field Specification:



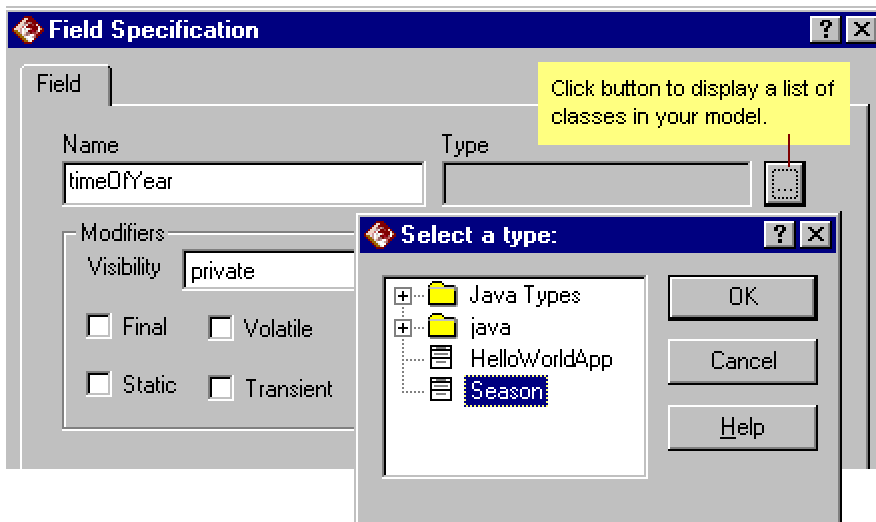
Note that you can draw associations without explicitly naming the roles. In our example, we assigned the name timeOfYear to Role A. By default, however, Rose assigns role names by adding the word *the* to the class name at the other end of the association. For example, if we hadn't named the role, Rose would have named Role A theHelloWorldApp and Role B theSeason. If we had used these names, the generated code for HelloWorldApp would be:

```
public class HelloWorldApp
{
   private Season theSeason;
```

When you generate code for a reference type of variable, you can use the Generate Fully Qualified Type setting if you want Rose J to generate the complete path of the type class in the source code. This is essential if you have duplicate class names in different packages. It ensures that the code references the correct class.

Creating Variables with User Data Types

Another way to create a reference variable is to create an attribute and give it a data type of another class. For example:



Now instead of showing an association to another class, the reference variable looks like an attribute:

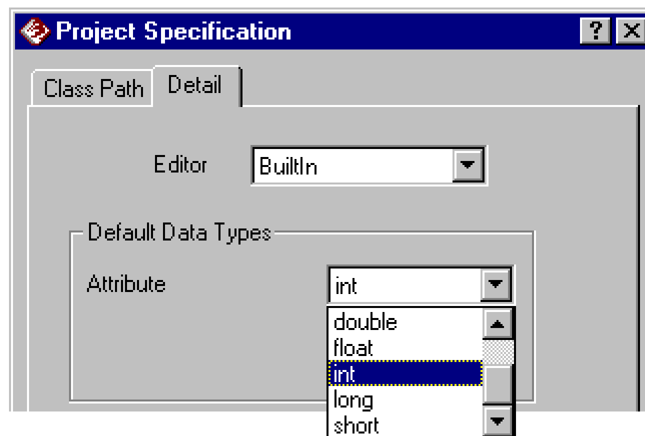
This seems like a simpler way to define reference variables but when you generate code for the class then reverse-engineer it back into a Rose model, Rose will convert the attribute to an association between the two classes.

As with the alternative technique for creating reference variables, you can use the Generate Fully Qualified Type setting if you want Rose J to generate the complete path of the Type class in the source code.

Supported Java Semantics for Variables (Fields)

Rational Rose J supports (and generates code for) these semantics, whether the variable is modeled as a class attribute or as a role:

- **Type**. This can be either a primitive type or a reference type. The standard Rational Rose default type is int (integer). However, you can establish a Java-specific default via the **Detail** tab of the Java Project Specification:

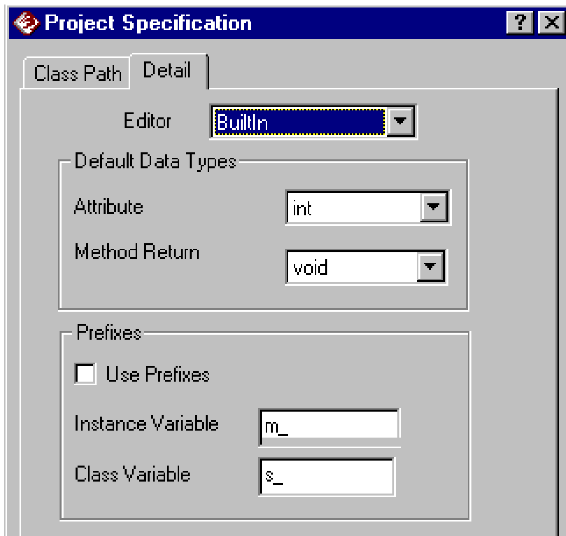


- **Visibility**. This can be public, private, package, or protected. Note that if a variable's type is primitive or if you use the Java Field Specification to name the data type, the Rose J default is private. If the type is a reference and you created it by drawing an association, the default is public.
- **Modifiers**. These are final, static, volatile, and transient.
- **Initial Value**. You can define an initial value for the variable.
- **Container Class**. You can specify that a variable get its type from a container class such as the Java Vector class or a container class that you've developed. (Vectors and arrays are discussed in more detail later in this chapter.)
- **Java Bean Properties**. These are discussed in more detail later in this chapter.
- **DocComment**. You can annotate the variable declaration. There are three types of comments you can generate: standard Rose format (the default), standard Java format, or Javadoc comments with specific tags for generating HTML-based documentation. See the description of Javadoc later in this chapter for details.

All of these settings are available from the Java Field Specification.

Note: *When you use the Java Field Specification to fine-tune a variable, the specification enforces standard Java rules. If you use the standard Rose specification instead, this checking is not done and errors may not be obvious unless you check syntax, either explicitly (you can select syntax checking from the* **Tools > Java** *menu) or as part of code generation.*

In addition to these semantics, the Java Project Specification enables you to establish default prefixes to use when generating code for variables:

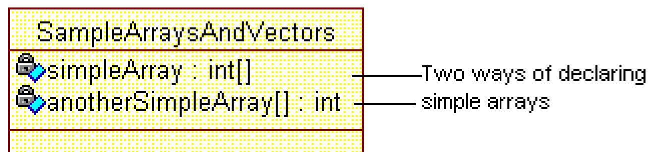


Arrays and Vectors

Rational Rose J models arrays and vectors the same way it models variables, meaning that it uses attributes in a class and roles in association relationships.

Note: *Rational Rose J provides a convenient way to generate arrays and vectors, but they are best forward-engineered (code generation) only. If you subsequently reverse-engineer the generated code, the model will vary from the original.*

In this first example, consider two simple arrays of integers where the brackets [] have been added to the attribute definitions:



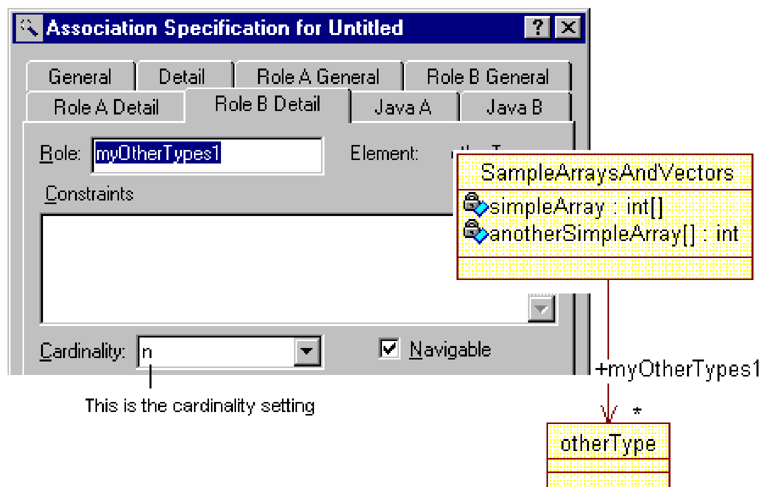
If you generated code from this class you would get:

```
public class SampleArraysAndVectors {
   private int[] simpleArray;
   private int anotherSimpleArray[];
```

To create an array of objects, you would create a variable exactly as described for a variable with a referenced type. Namely, you would draw the association between supplier and client, name a role as the array, and disable the other role's navigability.

To generate the variable as an array vs. a variable, *you set array role's cardinality to a value greater than 1* (for example, n). The cardinality is the setting that triggers Rational Rose J to generate the array. You set cardinality in the Association Specification.

For example:

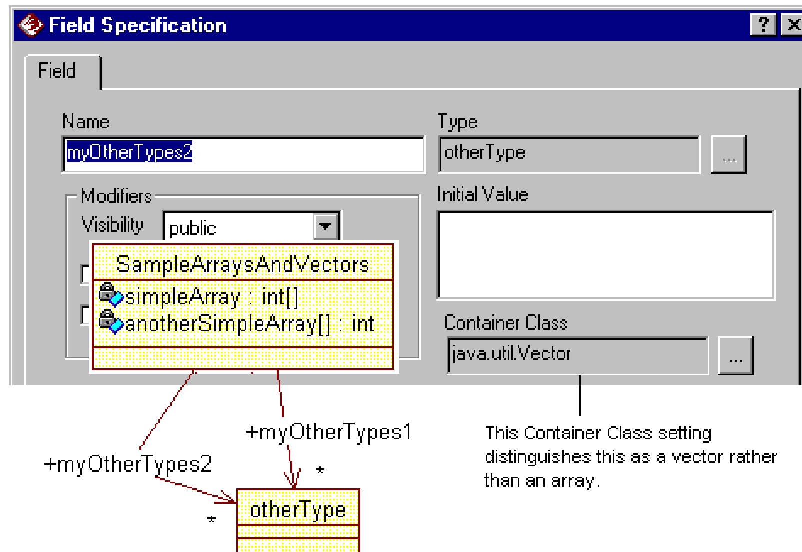


This construct would generate this code:

```
public class SampleArraysAndVectors {
   private int[] simpleArray;
   private int anotherSimpleArray[];
   public otherType myOtherTypes1[];
```

Now consider vectors. You can create a vector by creating the same association between classes as for an array, but identifying a container class, such the Vector Java class from java.util, in the Java Field Specification. (The presence of a value in the **Container Class** overrides the cardinality that triggers an array.)

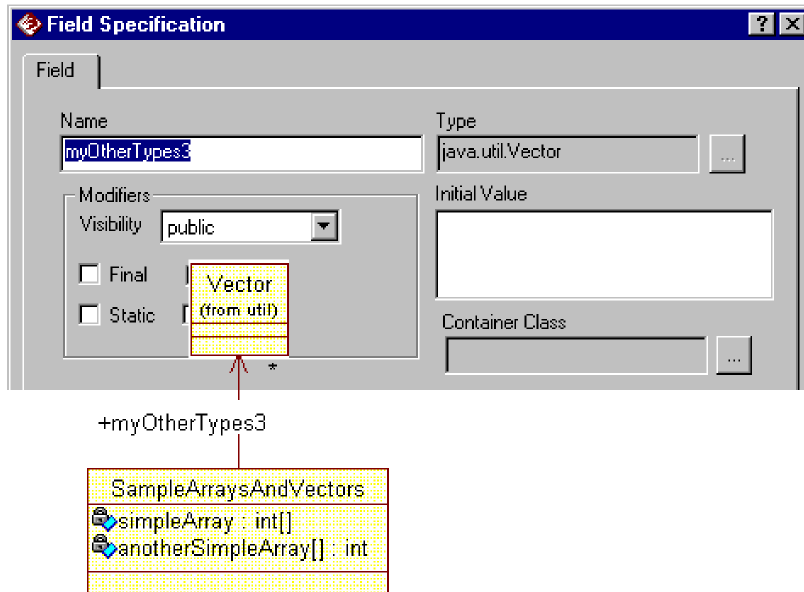
For example:



The code generated for this would be:

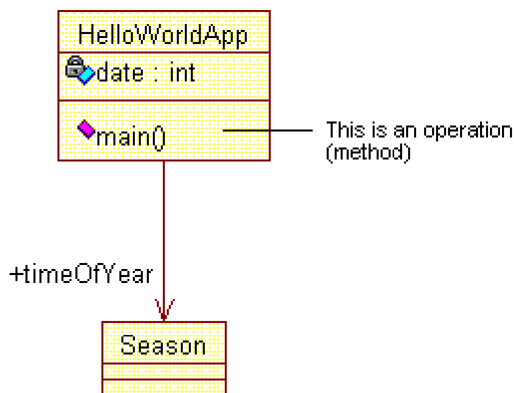
```
public class SampleArraysAndVectors {
   private int[] simpleArray;
   private int anotherSimpleArray[];
   public otherType myOtherTypes1[];
   public Vector myOtherTypes2;
```

Another way to generate a vector would be to specify the variable's (role's) type as java.util.Vector. For example:



Java Methods

Rational Rose J models Java methods as operations:



You can create a Java method using the same techniques you would use to create an operation. For example, you can select a class, either in the browser or in a diagram, click the right mouse button to display the shortcut menu, then click **New Operation**.

Like attributes and roles, operations support Java-specific semantics that enable you to generate Java methods. The easiest way to do this is to use the Java Method Specification to select the semantics you need.

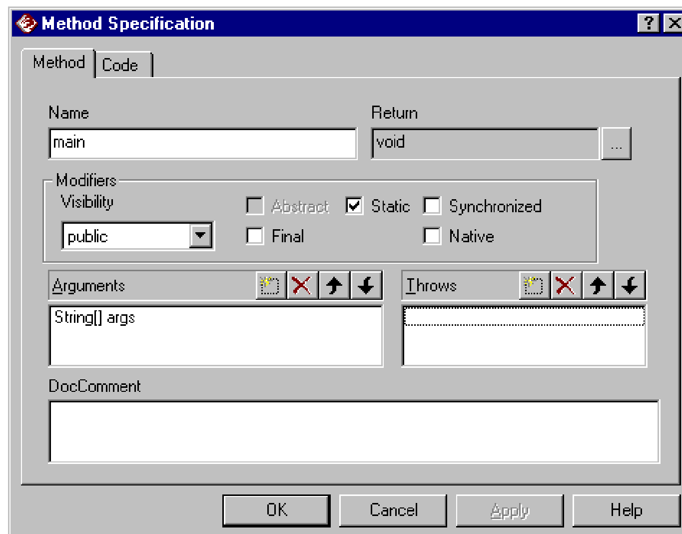
Supported Java Semantics for Methods

The semantics Rational Rose J supports are:

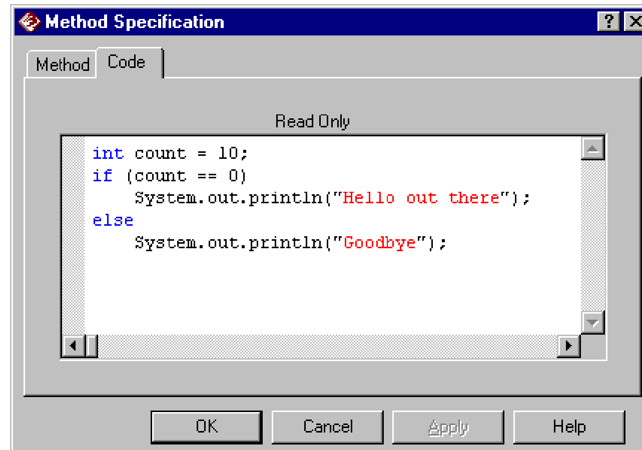
- **Return Type**. You can opt for the default (void), or select from the types defined in your model.
- **Modifiers.** These are abstract, final, static, synchronized, and native.
- **Visibility.** These are public, private, protected, and package.
- **Arguments**
- **Throws.** You can identify the exception classes to associate with the method.

- **DocComment**. You can annotate the method declaration. There are three types of comments you can generate: standard Rose format (the default), standard Java format, or Javadoc comments with specific tags for generating HTML-based documentation. See the description of Javadoc later in this chapter for details.

All of these semantics are available from the Java Method Specification:



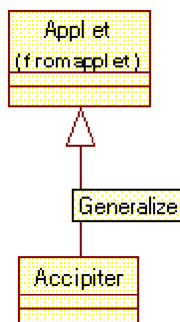
In addition, when you reverse engineer existing Java classes whose methods have code, the code is displayed in a read-only format from the **Code** tab of the Java Method Specification. For example:



Note: *When you use the Java Method Specification to fine-tune a method definition, the specification enforces standard Java rules. If you rely on the standard Rose specification instead, this checking is not done and errors may not be obvious unless you check syntax, either explicitly (you can select syntax checking from the* **Tools > Java** *menu) or as part of code generation.*

Java Extends

Rational Rose J models Java extends relationships as generalization relationships. For example, the Accipiter class below extends the Java class Applet:



To create an extends relationship in a Rational Rose diagram, you can use the Generalization drawing tool to draw the relationship ***from*** the class that's extending another class (e.g., from Accipiter in the example above to Applet). Or, once the class exists and has been assigned to a Java component, you can open the Java specification for the class and use the **Extends** field to select a class.

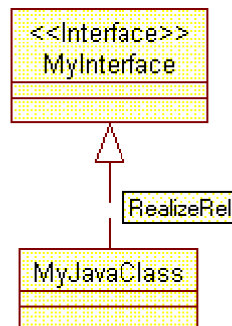
Note: *If you use the Java specification rather than the drawing tool to create the extends generalization relationship, the relationship does not show up on the class diagram unless you use the* **Expand Selected Classes** *from the* **Query** *menu.*

Rational Rose J supports the following semantics related to generalization:

- Enforcement that a class only extends at most one other class.
- Allowance for an interface to extend multiple interfaces.

Java Implements

Rational Rose J models **implements** as a realization relationship between a subclass and a superclass that has a stereotype of **Interface**.



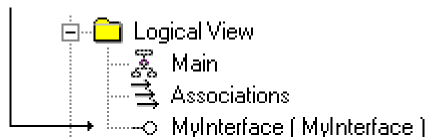
To create a Realize relationship, you can use the Realize drawing tool to draw the relationship ***from*** the Java class (MyJavaClass) ***to*** the Interface class (MyInterface). Or, you can use the **Implements** field in the Java Class Specification to select the appropriate Interface.

Note: *If you use the Java Specification rather than the drawing tool to create the realizes relationship, the relationship does not show up on the class diagram unless you use the* **Expand Selected Classes** *from the* **Query** *menu.*

Java Interfaces

Rational Rose J models a Java interface as a class with a stereotype of **Interface**.

In the Browser, the interface looks like this.



While the class icon in a diagram displays the Interface stereotype.

A Java interface is essentially an abstract class that contains only constant (static, final) variables and method signatures without implementations.

You can use the Java Class, Field, and Method Specifications to define a class that conforms to the semantics of a Java interface. Specifically, you can:

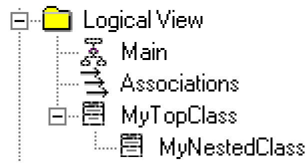
- Assign the Interface stereotype to the class.
- Add the modifiers **static** and **final** to any variables that are part of the interface.
- Define the method signatures that are part of the interface.

(Like all Java classes in a Rational Rose model, interface classes must be associated with a component in order for Rational Rose J to generate code for the class.)

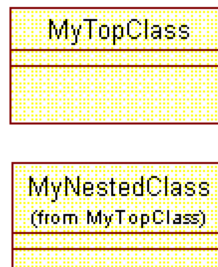
Java Nested and Inner Classes

In Java, a nested class is defined as a static class that is a member of another class. An inner class is a non-static nested class. Rose J models nested and inner classes as standard Rose nested classes. For example:

In the Browser, a nested class appears below its parent or top class.



In a diagram, the nested class displays the name of its parent or top class.

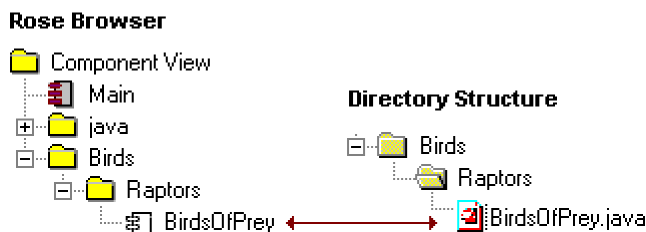


You create a nested class by first creating the parent (or top) class. From the parent's Standard Class Specification (not the Java Class Specification), click the **Nested** tab. From the shortcut menu click **Insert** and enter the name of the nested class. Click **Done**.

To display a nested class in a diagram, either drag it from the browser or click **Query** > **Add Classes**. The nested class is displayed in the list box of **Classes** beside the name of the parent class. Select the nested class and place it in the **Selected Classes** list box. When the diagram is displayed, the nested class is displayed with the parent class name in parenthesis below it.

Java (.java) Files

Rational Rose J models .java files as Rational Rose **components** in a model's Component View.

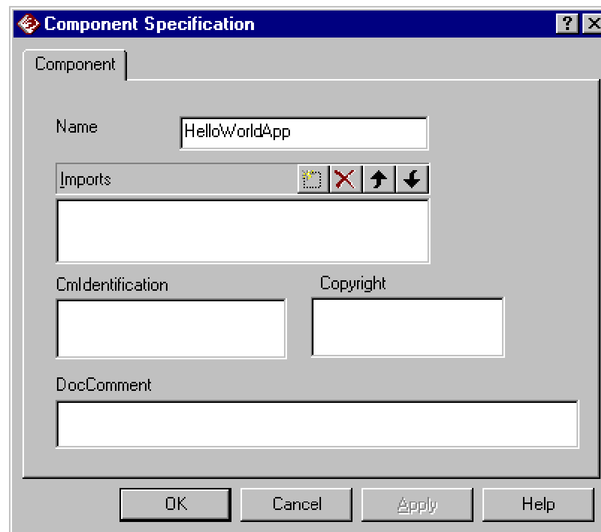


Associated with those components are the Java classes that are part of your model's Logical View.

In Rational Rose, components and component diagrams represent the physical (versus the logical) structure of your model. Hence, in code generation and reverse engineering Rational Rose relies on components to model the physical Java file structure. (And note that in the example, components can be part of component packages, which are Rational Rose J's way of modeling Java packages.)

Rational Rose Tip: To add existing .java and .class files as well as compressed Java .zip, .cab, and .jar files to your model you can drag and drop them into a diagram from other sources, such as from Microsoft Explorer. When you do this, Rational Rose unpacks the files and reverse engineers their contents into your model. Note, however, that Rational Rose cannot recreate or repackage the .zip, .cab, and .jar files when you generate code from your model.

Once you've created a component and set its **Language** to Java, you can use the Java Component Specification to display and modify settings:



These settings include:

- **Imports**. This field lists any additional Java import statements you want to include when you generate Java code for a component. The contents of this field (if any) are included directly in the generated *.java file.
- **CmIdentification**. This is a version control identification tag (character string) that you define and that is included in the header of the .java file when you generate code for a class/component. By default, this field is blank.
- **Copyright**. This is a copyright message (character string) that you define and that is included in the header of the .java file when you generate code for a class/component. By default, this field is blank.
- **DocComment**. This text field enables you to add comments that describe the component.There are three types of comments you can generate: standard Rose format (the default), standard Java format, or Javadoc comments with specific tags for generating HTML-based documentation. See the description of Javadoc later in this chapter for details.

Java Components and Code Generation

In order to generate Java source code from your model, the Java class or classes you've selected must be assigned to a component in your model's Component View.

If you don't create components before you generate code for the classes, Rational Rose J creates the components for you. Specifically, it creates a component for each class you've selected. Each new component will have the same name as the class assigned to it. When code generation completes, you have a .java file for each class.

If you want to generate multiple classes to a single .java file, you need to manually create a component, assign Java as its Component Language (if Java is not your default model language), then assign the class(es) to the new component.

You can then generate code from any of the classes or from the component itself. The .java file that Rational Rose J generates contains the code for all of the classes in the component.

Note: *Rational Rose J enforces the rule that a .java file can contain no more than one public class. It also enforces the requirement that the component the public class is assigned to must have exactly the same case-sensitive name as the class. (If the class CatsAndDogs is public, the component it is assigned to must be called CatsAndDogs.) And be careful…when you create a class in Rational Rose, the default access level is always public.*

If, in your model, you create a dependency between classes or components, Rational Rose J generates the appropriate import statements in the Java source code. For example, if a class is an applet (it extends the applet class), when you generate code for the class, Rational Rose J creates the import statement for the Java applet class:

```
import java.applet.Applet;
```

See Java Imports for more information about how Rational Rose J models import statements.

Java Components and Reverse Engineering

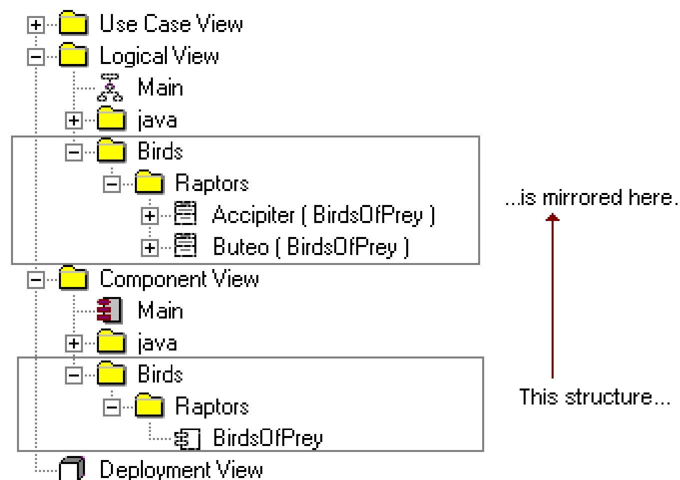
When you reverse engineer Java source, Rational Rose J creates the Java components for you. For example, if you reverse engineer a .java file that defines 10 classes, Rational Rose J creates a component using the .java file name. All 10 classes in the model are assigned to that component.

If your .java file contains import statements for other .java files, Rational Rose J creates a dependency relationship between them. For example, if you reverse engineer a class called CatsAndDogs that extends the Java applet class, Rational Rose J creates a dependency between CatsAndDogs and Applet. See Java Imports for more information about how Rational Rose J models import statements.

Note that when Rational Rose J models components and component packages from your Java source, it creates a mirror image of the component structure in your Logical View using logical component packages and logical components. For example, the source statement:

```
package Birds.Raptors;
```

generates this structure in your model:



Java Packages

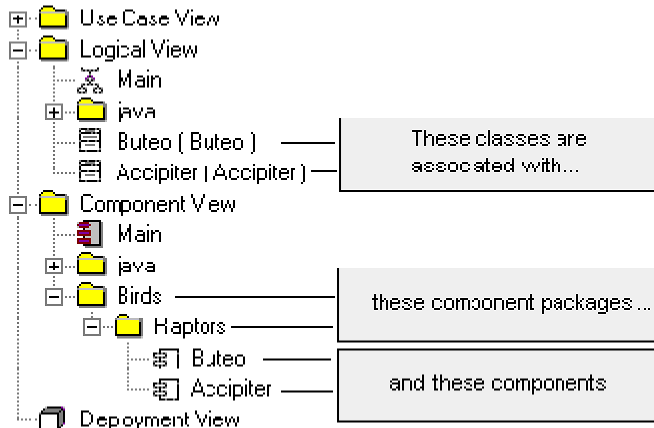
A Java package is the logical equivalent of a component package in Rational Rose, with one very important distinction: where a Java package is a collection of classes or other packages, a Rational Rose component package is a collection of components or other component packages. Rational Rose J is component-centered, meaning that each Java class you model in Rational Rose (and that you intend to generate Java source code for) has to be assigned to a Rational Rose component. In this regard a component is Rational Rose's way of modeling a .java file.

Java Packages and Code Generation

When you generate code from your model, Rational Rose J:

- Creates the appropriate package declarations in your source.
- Creates the directory structure that mirrors the component package structure in your model.

Consider this example from the browser:



If you generated Java source code from one of the classes in the model, the code, including the Java package declaration would look like this:

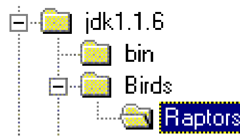
```
//Source file: D:/jdk1.1.6/Birds/Raptors/Buteo.java

package Birds.Raptors;

public class Buteo {

    public Buteo() {
    }
}
```

And the directory structure in Rational Rose J would generate would look like this:



Rational Rose J's ability to automatically create new directories from the packages in your model is controlled by the Create Missing Directories property. You set this property via the Java Project Specification or via Rational Rose model properties.

The class path setting you establish in Rational Rose J determines which directory/package is at the top of your directory/package structure. Note that the comment Rational Rose generates at the top of the code example specifies the fully qualified path to the .java file. From this comment, you know that the class path setting for the sample model is set to D:\jdk1.1.6. There is no limit to how deeply you nest your packages/directories.

Java Packages and Reverse Engineering

When you reverse engineer Java source, Rational Rose J creates the Java component packages for you based on the package statements in the code.

Note that when Rational Rose J models components and component packages from your Java source, it also creates a mirror image of the component package structure in your Logical View using logical component packages and logical components.

Java Imports

Java import specifiers resolve references to classes that are outside of a .java file. An import specifier lists the fully qualified name of another Java class.

For example, the import specifier *import java.applet.Applet* states that the identifier Applet is a Java class defined in package java.applet. Likewise, the import specifier *import java.util.** states that at compilation time, the class definitions scoped to the *java.util package* should be included as valid Java identifiers. Rational Rose J will generate import specifiers based on the following model relationships:

- Dependency relationship between components in different packages
- Dependency relationship between a component and a package other than the parent package of the component
- Dependency relationship between classes allocated to different packages
- Association relationship between classes allocated to different packages
- Generalization relationship between classes allocated to different packages
- Realization relationship between classes allocated to different packages

Java Beans

Rational Rose J models Java Beans as Rational Rose attributes or roles with Java Bean properties. The properties are controlled by setting an attribute's or role's model properties. You access these properties via an attribute or association in the Rational Rose Specification, or by using the Java Field Specification. (Java Custom Specifications are only available when a class has been assigned to a component whose language has been set to Java.)

Read/Write Property

You can specify whether or not a bean should allow an external customer to set or get a property. If you select read access, Rational Rose generates a bean get method. For write access, Rose generates a bean set method.

Support Individual Change Management

If a bean property is bound or constrained, a Java Bean must inform other classes through a registration mechanism. Rose enables a bean to use a single registration class (the default behavior). Or you can elect to have a property use its own registration mechanism. By specifying that each property has its own change management listener, Rose enables you to optimize classes and performance.

Property Types

The following are the property types you can set for Java beans.

Simple

This generates a property declaration and a get/set method, depending on how you set the Read/Write property. For example, for a variable named myProp of type myType, the following code is generated:

```
myType getMyProp() {
   return myProp;
   }

void setMyProp(myType aMyProp) {
   myProp = aMyProp;
   }
```

Bound

For a Bound property setting, Rational Rose generates the import statement:

```
import java.beans.*,
```

For the attribute, Rational Rose generates a declaration, a set method, and an optional get method, depending on the setting of the Read/Write property. (Note that a bound property without a set method is inconsistent. If you specify that the variable is bound and read-only, Rational Rose will issue a warning when you generate code.)

The get method is the same as the get method that is generated for the Simple property setting.

If any variable of the class is bound and its Individual Change Management property is not set (i.e., False), the set method and the PropertyChangeSupport, common addPropertyChangeListener, and removePropertyChangeListener methods are generated for the class as follows:

```
protected PropertyChangeSupport commonPCS = new
PropertyChangeSupport(this);

void setMyProp(myType aMyProp) {
   myType oldMyProp = myProp;
   myProp = aMyProp;
   commonPCS.firePropertyChange("myProp", oldMyProp, myProp);
   }

public void addPropertyChangeListener(PropertyChangeListener
listener) {
   commonPCS.addPropertyChangeListener(listener);
   }

public void removePropertyChangeListener(PropertyChangeListener
listener) {
   commonPCS.removePropertyChangeListener(listener);
   }
```

If a variable's Individual Change Management property is set (True), the set method and the specific PropertyChangeSupport, specific addPropertyChangeListener, and removePropertyChangeListener methods are generated as follows:

```
protected PropertyChangeSupport myPropPCS = new
PropertyChangeSupport(this);

void setMyProp(MyType aMyProp) {
   MyType oldMyProp = myProp;
   myProp = aMyProp;
   myPropPCS.firePropertyChange("myProp", oldMyProp, myProp);
   }

public void addMyPropListener(PropertyChangeListener listener)
{
   myPropPCS.addPropertyChangeListener(listener);
   }

public void removeMyPropListener(PropertyChangeListener
listener) {
   myPropPCS.removePropertyChangeListener(listener);
   }
```

Note that if a variable has a primitive data type, the second and third parameters of the calls to firePropertyChange would be the appropriate object wrappers, as follows:

```
myPropPCS.firePropertyChange("myProp", new Integer(oldMyProp),
new Integer(myProp));
```

Constrained

For a Constrained property setting, Rational Rose generates the import statement:

```
import java.beans.*,
```

For the variable, Rational Rose generates a declaration, a set method, and an optional get method, depending on the setting of the Read/Write property. (Note that a constrained property without a set method is inconsistent. If you specify that the variable is constrained and read-only, Rational Rose will issue a warning when you generate code.)

The get method is the same as the get method that is generated for the Simple property setting.

If any variable of the class is constrained and its Individual Change Management property is not set (i.e., False), the set method and the VetoableChangeSupport instance, common addVetoableChangeListener, and removeVetoableChangeListener methods are generated for the class as follows:

```
protected VetoableChangeSupport commonVCS = new
VetoableChangeSupport(this);

void setMyProp(MyType aMyProp) throws PropertyVetoException {
   commonVCS.fireVetoableChange("myProp", myProp, aMyProp);
   myProp = aMyProp;
   }
public void addVetoableChangeListener(VetoableChangeListener
listener) {
   commonVCS.addVetoableChangeListener(listener);
   }
public void removeVetoableChangeListener(VetoableChangeListener
listener) {
   commonVCS.removeVetoableChangeListener(listener);
   }
```

If a variable's Individual Change Management property is set (True), the set method and the specific VetoableChangeSupport instance, specific addVetoableChangeListener, and removeVetoableChangeListener methods are generated as follows:

```
void setMyProp(MyType aMyProp) throws PropertyVetoException {
   myPropVCS.fireVetoableChange("myProp", myProp, aMyProp);
   myProp = aMyProp;
   }
public void addMyPropListener(VetoableChangeListener listener)
{
   myPropVCS.addVetoableChangeListener(listener);
   }
public void removeMyPropListener(VetoableChangeListener
listener) {
   myPropVCS.removeVetoableChangeListener(listener);
   }
```

Javadoc and Comment Text

Rational Rose specifications enable you to document your model elements by adding text to various Documentation fields. Rose J uses the text you supply to create comments in the Java code it generates. There are three comment types that Rose J can generate:

- **Rose Default**. This style uses the standard /** */Rose comment style. For example:

```
//Source file: c:/temp/MyNewClass.java
/**
This is a test of the comment style.
 */
public class MyNewClass
{
   public MyNewClass() {}
}
```

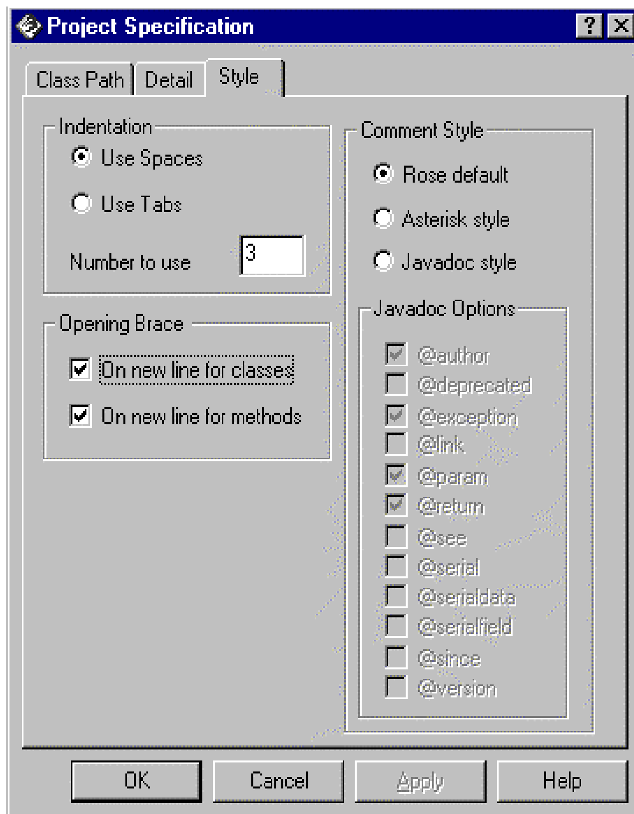
- **Asterisk Style**. This style inserts an asterisk at the start of the comment. This is the standard Java style. For example:

```
//Source file: c:/temp/MyNewClass.java
/**
* This is a sample asterisk style comment.
 */
public class MyNewClass
{
   public MyNewClass() {}
}
```

- **Javadoc style**. This style uses Javadoc tags that the Javadoc compiler uses to create HTML pages that describe various Java constructs such as classes, interfaces, constructors, methods, etc. For example:

```
/**
   * @param balance
   * @param type
   * @param number
   * @return
   * @exception
   *
   @roseuid 36782AC702D9
    */
   public BankAccountServant(short balance, String type,
String number) {
    balance_ = balance;
    type_ = type;
    number_  = number;
  }
```

You set the style on from the Java Project Specification's **Style** tab.



If you select Javadoc as the style, you enable selection of any of the Javadoc options listed. Note that @exception, @param, @return, @serial, @serialdata, and @serialfield are generated only for methods. All other tags listed are common to classes and methods. Note, too, that the comment style you implement affects all Java elements in your model.

Chapter 3

Forward Engineering with Rational Rose J

About the Steps You Follow

Forward engineering with Rational Rose J is a process comprised of the following major tasks:

- Assign Java classes to Java components in your model
- Check syntax (optional)
- Check the class path
- Set the Project Properties that affect code generation
- Back up your source
- Generate Java source code from your model
- View (browse) and extend the generated source

Assign Java Classes to Java Components in Your Model

Rational Rose J models .java files as components. (Your model's Component View models your physical file structure.) Therefore, to generate code successfully, Rose J requires that the Java classes in your model be assigned to Java components in the Component View of your model.

You have two assignment options:

- You can let Rose J create the components for you when you initiate code generation for one or more classes. When you do this, Rose J generates one .java file and one component for each class. In order to take advantage of this feature, you must set the default notation for your model to Java (**Tools** > **Options** > **Notation** > **Default Language**).

Rose J will not automatically generate code for multiple classes to a single .java file or component. (Note, too, that if you assign your Java classes to a logical package, Rose J creates a mirror image of your physical package in your Component View and uses it to create a directory/Java package based on the model package.)

- You can create the components yourself then explicitly assign the classes to the components. When you do this you can generate multiple classes to a single .java file.

Check Syntax

This is an optional step. You can choose to check the syntax of your model components before you try generating code. Note, though, that syntax checking is done automatically for you when you generate code. Rose J's Syntax checking is based on Java code semantics.

Check the Class Path

Rose J's **Class Path** tab of the Java Project Specification enables you to set up a specific Java class path tailored to your current model. Rose J uses this class path when you generate code or reverse engineer existing Java source.

You can use the Directories field of the **Class Path** tab if you want to dynamically add directories that extend your class path. Any settings you add here are saved with your model. You can add as many directories as you need, when you need them.

Set the Project Properties that Affect Code Generation

The properties that affect code generation are:

- **Stop on Error**. When enabled, Rose stops generating Java code at the first error it encounters. By default, this property is not set, thus allowing code generation to proceed even when there may be errors. (Note that errors can be viewed via the Rational Rose Log window.)
- **Create Missing Directories**. When the Create Missing Directories setting is enabled (the default), Rose creates any undefined directories that are referenced as packages in a Rose model when you generate Java code.

- **Automatic Synchronization Mode**. When this feature is enabled, Rose J automatically initiates code generation any time you create, delete, rename, or modify a Java element in your library. By default, this feature is off.
- **VM or Virtual Machine**. This setting determines which Java environment or IDE you're working in. The default, Sun, indicates you're working with a standard JDK. Alternatively, you could set the VM to IBM for integrating with VisualAge for Java, or Microsoft for integrating with Visual J++.

You can set the values for all of these settings via the **Detail** tab of the Java Project Specification (**Tools** > **Java** > **Project Specification**).

In addition, you can determine the format for the code you generate and control how comments are created with these properties:

- **Indentation**. You can specify whether to use spaces or tabs and the number of spaces or tabs to use when formatting code. The default is three spaces.
- **Opening Braces**. By default, opening braces start on a new line for both classes and method declarations.
- **Comment Style**. Rose J generates comment text from your model documentation fields according to the style you select: the Rose default, standard Java comments, or Javadoc. If you select Javadoc, Rose J generates the Javadoc tags you select. You can use the Javadoc compiler to generate HTML documentation.

You can set the values for all of these settings via the **Style** tab of the Java Project Specification (**Tools** > **Java** > **Project Specification**).

Back Up Your Source

When you generate code to an existing .java file, Rational Rose creates a backup of your current source using the file extension .~jav. However, if you intend to round-trip engineer your Java source, be sure

to back it up before you begin. If you generate code from a model more than once before looking for a backup, the .~jav file that Rose creates will not contain your original code.

Generate Java Source Code from Your Model

You can generate source code from a diagram or from the browser. You can select one or more classes or one or more components (logical or physical) then use the **Java** > **Tools** or shortcut menu to select **Generate Java**.

If this is the first time you've generated code for a model element, a mapping dialog appears (and described in detail later in this chapter) enabling you to map packages and components to your Rose J class path settings. If there are errors or warnings, a message will alert you and you can view the messages in the Rose Log window (available from the **Window** menu). Once code generation is complete, the .java files and related directory structure are in place. You can also view the newly generated java source from within Rose.

Note that you can enable a Rose J feature that will automatically initiate code generation any time you create or modify a Java element in your model. The **Automatic Synchronization Mode** feature is available from the Java Project Specification **Detail** tab (**Tools** > **Java** > **Project Specification** > **Detail**).

View (Browse) and Extend the Generated Source

After generating Java, you may want to view (browse) the generated source and create the actual functionality for your application or applet.

Rational Rose J provides a Builtin Editor for viewing and editing the Java source in .java files. This Builtin Editor is the default editor for a model's generated .java files. You can control which source editor is the default for your model by setting or resetting the model's **Editor** property

in the **Detail** tab of the Java Project Specification. If you prefer to use another editor, you can change the setting to **WindowsShell**, but make sure your file association for .java files has been set appropriately.

You can modify the generated source from within the editor. To update your model with the changes, you need to reverse engineer the .java file back into your model.

How Controlled Units Affect Code Generation

If you use controlled units, Rational strongly recommends that you load all units before you generate code. Although you are not required to do this, any code you generate may be incorrect or incomplete due to missing model elements. Depending on how they are used, controlled units can profoundly affect relationships and the Java constructs that rely on them. Specifically:

- If the supplier class of an association or the association itself is in an unloaded controlled unit, no Java field is generated for the association.
- If the supplier class of a generalization relationship is in an unloaded controlled unit, no Java depends relationship is generated.
- If the supplier class of a realizes relationship is in an unloaded controlled unit, no Java import statement for a component is generated.
- If the supplier component in a dependency relationship is in an unloaded controlled unit, no import package statement is generated.

If you do not load all of your model's controlled units before beginning code generation, Rose J displays the following message:

Not all units are loaded. Incorrect code might be generated. Do you want to continue?

You can continue, but be aware of the possible risk.

Mapping Components for Code Generation

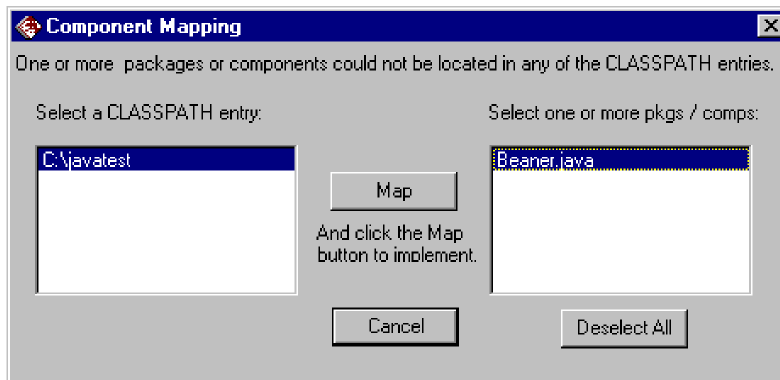
During code generation, if a package or component in your model is not mapped to a folder or file listed in your Rational Rose Java Class Path setting, Rational Rose displays the **Component Mapping** dialog, which allows you to do the mapping during code generation.

Follow these steps to map packages or components during code generation (and remember that you cannot force this mapping to occur before generation—the mapping dialog appears only after you've initiated code generation and only if Rose J cannot determine the appropriate mapping):

1. From the list of class path entries, click the path (directory structure) where you want to put the .java file when it is generated.
2. From the list of packages and components, click one or more items to map to the class path entry you selected.

   Note that the list of packages and components shows only the topmost level. Once you locate this level relative to your chosen class path, all of its subpackages and components in your model will be located within this hierarchy.
3. Click **Map** to map your selected package or component to the selected directory structure and complete code generation.

In this sample dialog we map the component called Beaner.java to the c:\javatest entry in the class path:



Generating Java Source from a Component Diagram

Follow these steps to generate Java source from a component diagram in Rational Rose:

1. Open your model and display the component diagram that contains the packages or components for which you want to generate Java source.
2. Check your class path by opening the Java Project Specification: click **Tools** > **Java** > **Project Specification**. If needed, use the Directories field of the **Class Path** tab to extend your class path.
3. Select one or more packages and components in the diagram.
4. Click **Tools** > **Java** > **Generate Code**. If Rational Rose J cannot map the component to an existing entry in your class path, it displays the mapping dialog described in the previous subsection.
5. Check the Rational Rose Log window to view the results of the Java generation, including any errors that may have occurred.
6. View and edit the generated code.

Generating Java Source from a Class Diagram

From a class diagram, you can generate Java source from classes or from logical packages. If Java is not your model's default language, you need to assign the classes to appropriate Java components. If Java is your default language, you can generate code for each class without assigning a class to a component.

Follow these steps to generate Java source from classes in a class diagram in Rational Rose:

1. Open your model and display the class diagram that contains the classes and packages for which you want to generate Java source.
2. Check your class path by opening the Java Project Specification: click **Tools** > **Java** > **Project Specification**. If needed, use the **Directories** field of the **Class Path** tab to extend your class path.
3. Select one or more classes or packages in the diagram.
4. Click **Tools > Java** > **Generate Code**. If Rational Rose J cannot map the component to an existing entry in your class path, it displays the mapping dialog described earlier in this chapter.

5. Check the Rational Rose Log window to view the results of the Java generation, including any errors that occurred.
6. Correct any errors and repeat steps 4 and 5 until no errors are returned.

Generating Code for Visual J++

There are three ways to establish a link between Rose J and Microsoft Visual J++:

- By generating source code from a Rose model
- By choosing **Open Model** from VJ++
- By reverse engineering files from the VJ++ Project Explorer

Both Rose and Visual J++ are able to activate each other, depending on the operation you are performing.

Generating VJ++ Code from a Rose Model

1. Before generating code, make sure you set the **Virtual Machine** property on the Java Project Specification to **Microsoft**.
2. Select the classes or components for which you want to generate source code.
3. Click **Tools** > **Java** > **Generate Java**.
4. If Visual J++ is not open, it will automatically start. If the project for which you're generating code already exists, Rose updates it. Otherwise, a new project is created.

Note: *In addition to code generation and reverse engineering, Visual J++ offers the option of starting Rose without opening a specific model or reverse engineering VJ++ source. From Visual J++, click* **Tools** > **View Rose Model**.

Viewing and Completing Java Source

The generated Java source for each component is stored in separate **.java** files. The generated source contains Java programming elements based on the objects and relationships defined in your model objects.

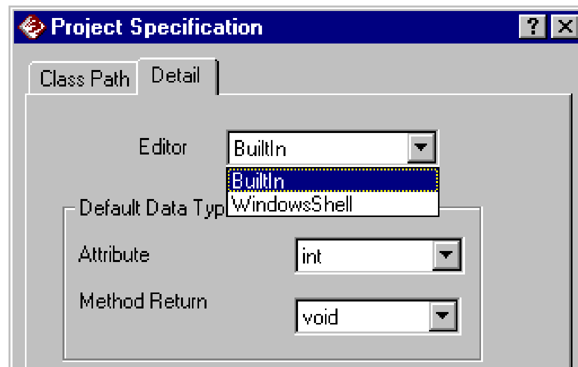
After generating Java, you will want to view (browse) the generated source and create the actual functionality for your application or applet.

Using the BuiltIn Editor

Rational Rose J provides a Builtin Editor for viewing and editing the Java source in .java files. This Builtin Editor is the default editor for a model's generated .java files.

You can control which source editor is the default for your model by setting or resetting the model's **Editor** property in the Java Project Specification or in the model property settings.

For example:



If you prefer to use another editor as the default, set the **Editor** property to **WindowsShell**. Just be sure you've made the appropriate association between .java file types and the editor of your choice. (For example, in Windows Explorer, click **View** > **Options** to display the **File Types** dialog. Click JAVA files and edit the setting to point to the editor of your choice.)

When you use the Builtin editor, you will see the source displayed with the following characteristics:

- Keywords in blue
- Literal strings in red
- Comments in green

Use the editor's **Format** menu to change the colors and fonts used, as well as to set tabs for indention.

The following table lists the shortcut keys that the BuiltIn Editor supports:

Table 1 Shortcut Keys for the BuiltIn Editor

To Perform This Action...	Use This Shortcut Key...
Clear	DELETE
Copy	CTRL+C or CTRL+INS
Cut	SHIFT+DELETE
Find	CTRL+F
Find next (used after CTRL+F)	F3
Go to Beginning of Line	HOME
Go to Bottom of File	END+DOWN ARROW
Go to End of Line	END
Go to Next Bookmark	F2
Go to Top of File	CTRL+HOME
Paste	CTRL+V or SHIFT+INS
Replace	CTRL+H
Select All	CTRL+A
Select Down	SHIFT+DOWN ARROW
Select from cursor location to beginning of line	SHIFT+HOME
Select from cursor location to end of line	SHIFT+END
Select from cursor location to bottom of file	CTRL+SHIFT+END
Select from cursor location to top of file	CTRL+SHIFT+HOME
Set Bookmark	CTRL+F2
Undo	ALT+BACKSPACE
Word Left	CTRL+LEFT ARROW

Browsing Java Source

Follow these steps to browse (display) Java source created through forward engineering:

1. Right-click the component or class whose source you want to browse.
2. On the shortcut menu, click **Java** > **Browse Code**. The source file is displayed using the application with which the file is associated.

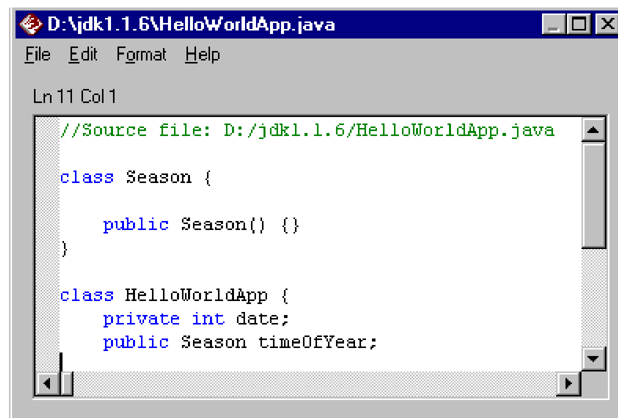
There are several reasons why a source file might not appear:

- The BuiltIn Editor is not your default editor and .java files are not associated with an application.

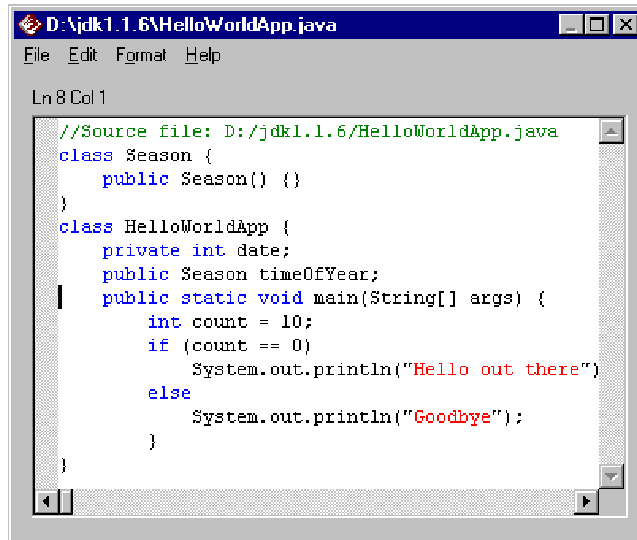
  In this case, from Windows Explorer click **View > Options**. Use the **File Types** tab to specify the application to use when opening **.java** files. Check Windows Help if you need more information on working with file types.
- You reverse engineered a **.class** file. These files do not contain Java source.
- There were errors when you attempted to generate code and code generation did not complete. Check the Rational Rose Log for any error messages.

Completing Generated Java Source

Suppose you created a component called **HelloWorldApp** in your model. The generated Java source might look like this:



You can complete the Java source by defining a **main** method that implements the actual functionality of the class. For example:



You can then reverse-engineer changes back into your model (and the new code will appear on the **Code** tab of the Java Method Specification). This is a very simple example of iterative development.

Chapter 4

Reverse Engineering with Rational Rose J

About Reverse Engineering

Reverse engineering is the process of creating or updating a model by analyzing Java source. As Rational Rose J reverse engineers each source or byte code file, it finds the classes and objects in the file and includes them in your Rational Rose model.

You can reverse engineer .java, .class, .cab, .jar, and .zip files. In fact, you can initiate reverse-engineering by simply dragging and dropping any of these file types into a Rose class diagram. Note, however, that if you subsequently generate code from compressed files (.cab, .jar, or .zip files), these files are not recompressed for you.

For complete information on how Java constructs map to Rational Rose and UML, refer to:

- How Rational Rose J Models Java Elements (Chapter 2)
- Java to Rational Rose Mapping Quick Reference (Appendix B)

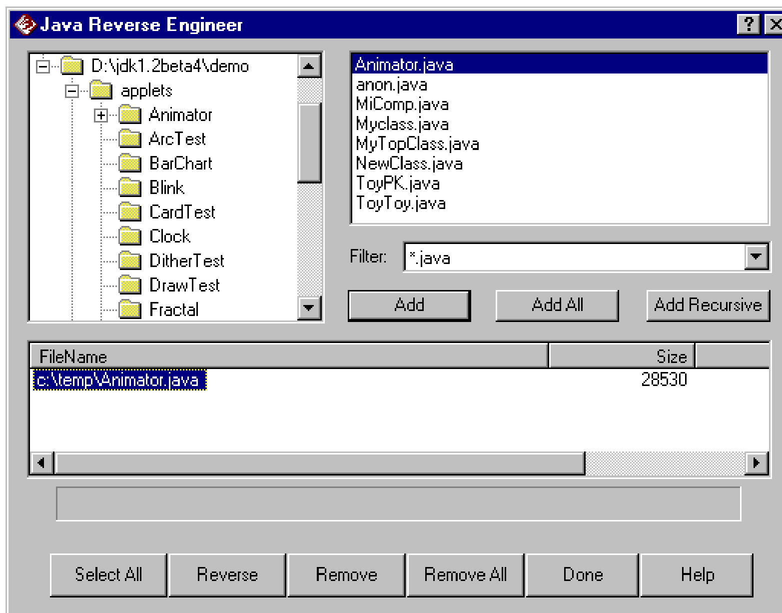
The remainder of this chapter provides detailed instructions for reverse engineering Java source into a Rational Rose model.

Note: *You must set have a class path setting to the Java API library in order to reverse engineer existing source. Please read about the class path in Chapter 2.*

Reverse Engineering Java Source

Follow these steps to reverse engineer all or part of Java source code:

1. If you are updating an existing model, open the model.
2. Check your class path by opening the Java Project Specification: click **Tools** > **Java** > **Project Specification**. If needed, use the **Directories** field of the **Class Path** tab to extend your class path.
   Note that you must have a class path to the Java API library in order to reverse engineer. See Chapter 2 for details.
3. In the browser or in a class or component diagram, right-click to display the shortcut menu. Click **Java** > **Reverse Engineer**. This displays the **Java Reverse Engineer** dialog:



4. Click the Java file type whose code you want to reverse engineer.
5. Click a folder in the tree to display the list of files it contains. (Traverse the tree to find the folder or subfolder that contains the files to be reverse engineered.)

6. Do one of the following to place the Java files of the type you selected into the **Selected Files** list:
   - In the list box, click on one or more individual files and click **Add**.
   - Click **Add All** to add all of the files.
   - Click **Add Recursive** to take all of the files of the selected file type that are contained in the currently selected folder **and all of its subfolders** and place them in the **Selected Files** list.
     (Use **Add Recursive** to select files in multiple folders without having to search for and select each one.)
7. Click on one or more files in the **Selected Files** box or click **Select All** to confirm the entire list of files to reverse engineer.
8. Click **Reverse** to create or update your model from the Java source you specified.
9. If Rose J encounters an error, a message appears and you should check the Rose Log window for a description of the error. Otherwise, click **Done** to close the dialog.

When reverse engineering is complete, you can view the new model elements in the browser, create your own logical views, and drag and drop the new elements into the views you create.

Reverse Engineering from Visual J++

There are three ways to establish a link between Rose J and Microsoft Visual J++:

- By generating source code from a Rose model
- By clicking **Tools** > **Open Model** from VJ++
- By reverse engineering files from the VJ++ Project Explorer

Both Rose and Visual J++ are able to activate each other, depending on the operation you are performing.

To reverse engineer a VJ++ project into a Rose model, follow these steps:

1. In Visual J++, select the project elements you want to bring into a Rose model.
2. Click **Tools** > **Update Rose Model**.

3. If Rose is not already running, Visual J++ automatically starts it. As reverse engineering progresses, the **Virtual Machine** property in Rose is set to **Microsoft** (if it isn't already set) and class path settings are updated to reflect the VJ++ project.
4. When reverse engineering is complete, the new classes/components are part of your model. (Check the browser.)

Note: *In addition to code generation and reverse engineering, Visual J++ offers the option of starting Rose without opening a specific model or reverse engineering VJ++ source. From Visual J++, click* **Tools** > **View Rose Model**.

Appendix A

Forward Engineering WalkThrough

Introduction

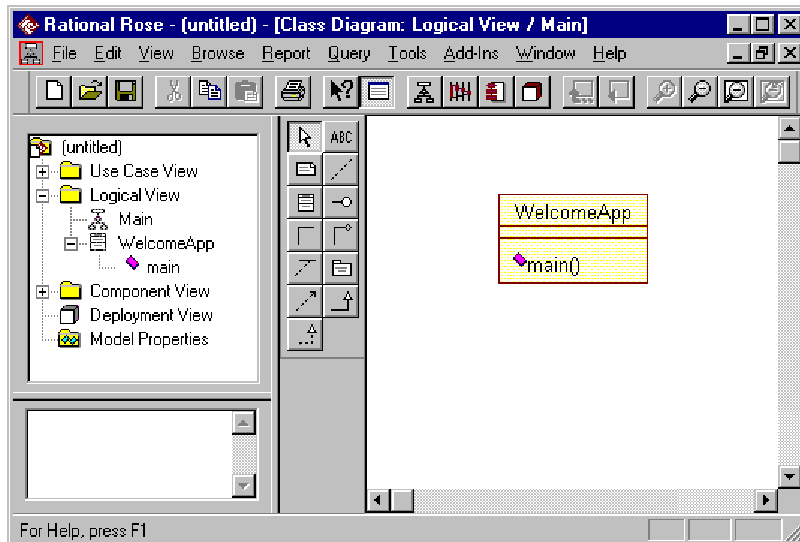
This forward engineering walkthrough takes you through a use case that creates a simple Java application which, when executed, displays the message “Welcome to Rational Software!”

Note: *These instructions assume that you are already comfortable with the Rational Rose tool, browsing the various diagram types, creating model elements, and so on. If not, you should complete the Rational Rose tutorial (included on the product CD) before continuing with this Java use case.*

Walkthrough

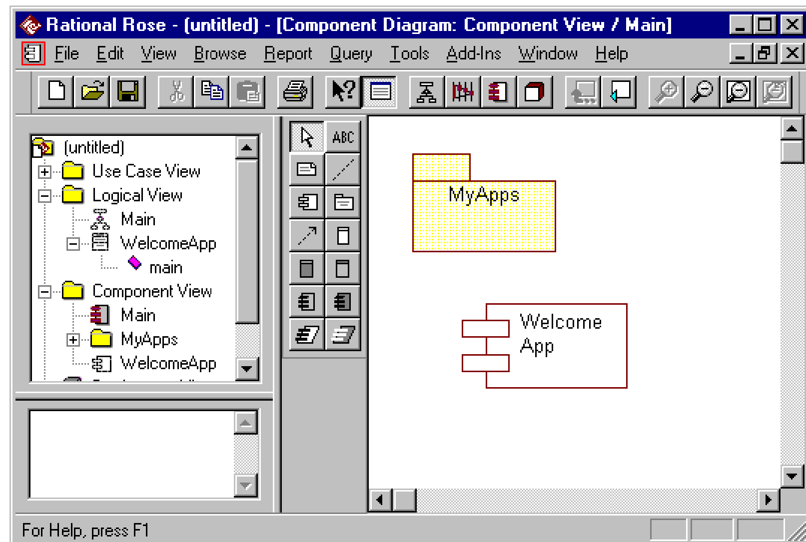
Create a Class and Add a Method

1. In the logical view of your new model, create a class called WelcomeApp.
2. Right-click the WelcomeApp class and click **New Operation** to add a new operation to the class. Name the operation **main**:

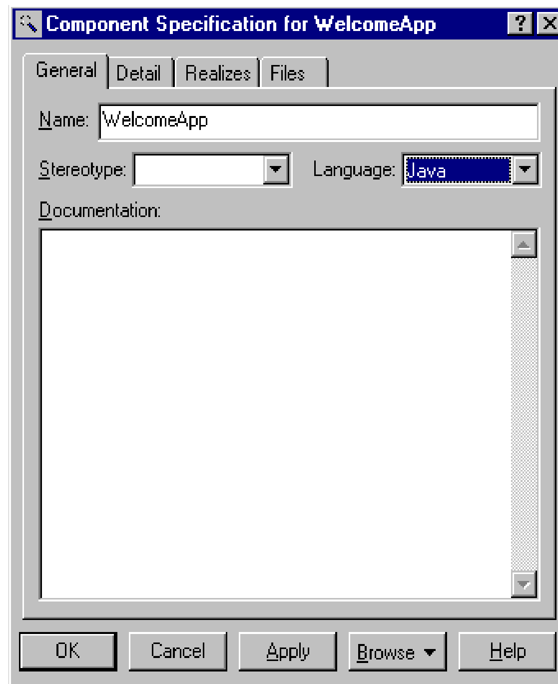


Create a Component Package and Component

1. In the Component View, create a package called **MyApps**.
2. Create a component called **WelcomeApp** and assign it to the MyApps package by dragging the component to the package name in the browser.



3. Double-click the component in the browser or in the diagram to open its specification and verify that the **Language** field is set to **Java**.



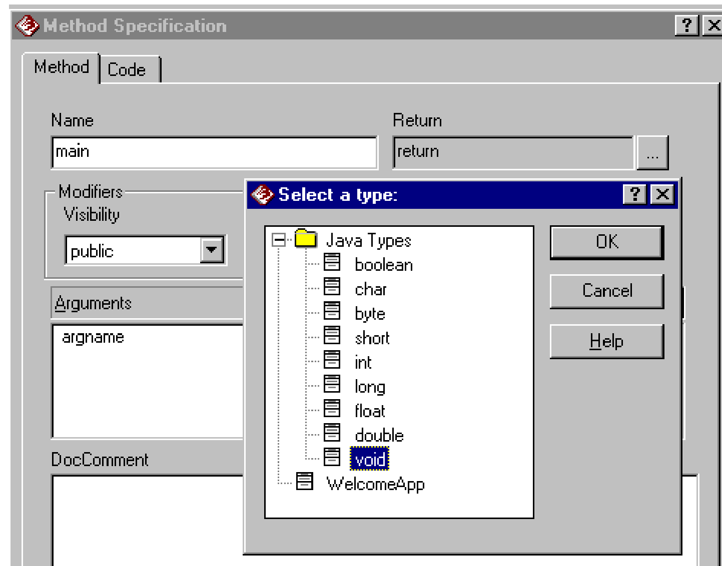
Assign the Class to the Component

1. In the browser, drag the WelcomeApp class to the WelcomeApp component to assign the class to the component.
2. Check the browser to make sure the class now has the component name next to it in parentheses.

Note: *By default, the WelcomeApp class is a public class. (When you generate code for this class, Rational Rose J generates a public modifier.) The component to which you assign any public class must have **exactly** the same case-sensitive name.*

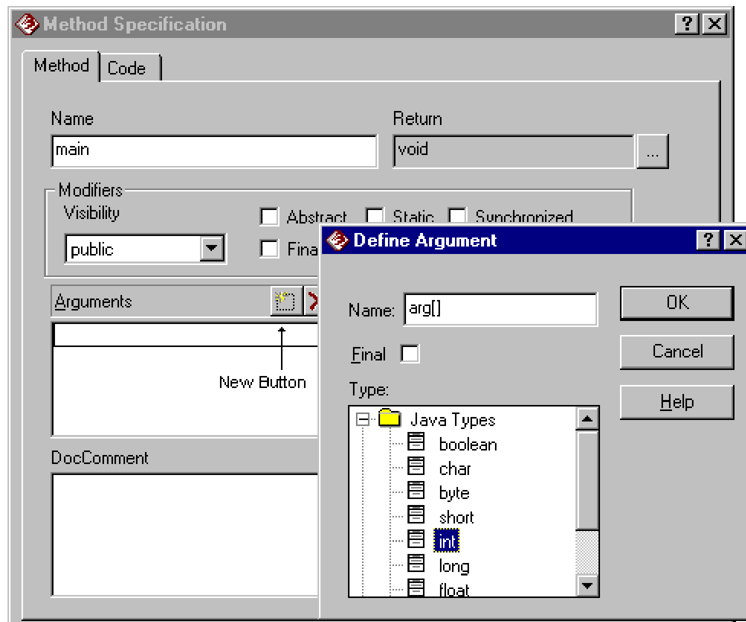
Edit the New Method

1. Double-click the main() operation (either in the browser or the diagram) to open and edit its Java Method Specification.
2. Change the return class to void by clicking the browse button on the **Return** field. This displays a list of the valid return classes. Select **void**.



3. On the Method Specification, select the **static** modifier check box.

4. On the **Arguments** field, click the new button to display a dialog that allows you to name an argument and specify its type. For our example, define an array (arg) of integers with a type of **int**:

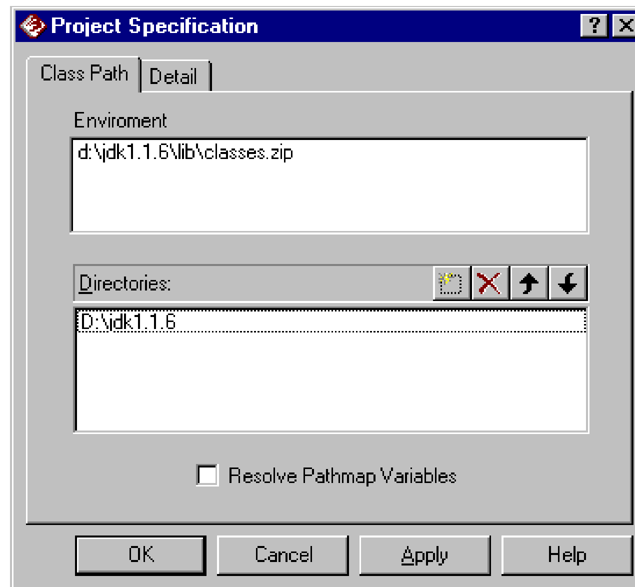


5. Click **OK** to both the **Define Argument** dialog and to the Method Specification to save your changes.

Set the Class Path

Establish where you want Rational Rose J to put the files it generates by setting the class path. (For details regarding setting the class path, please see Chapter 2 of this manual.

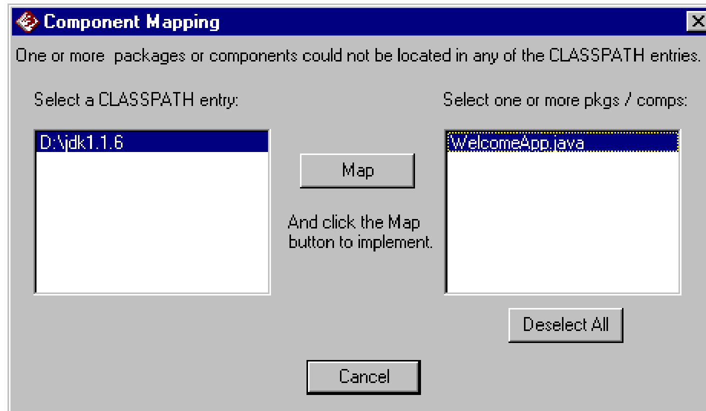
1. Display the Java Project Specification: click **Tools > Java > Project Specification**.
2. Click the new button to browse your system for the folder/directories you want to include in your class path. For example:



3. Click **OK** to save your changes.

Generate Java

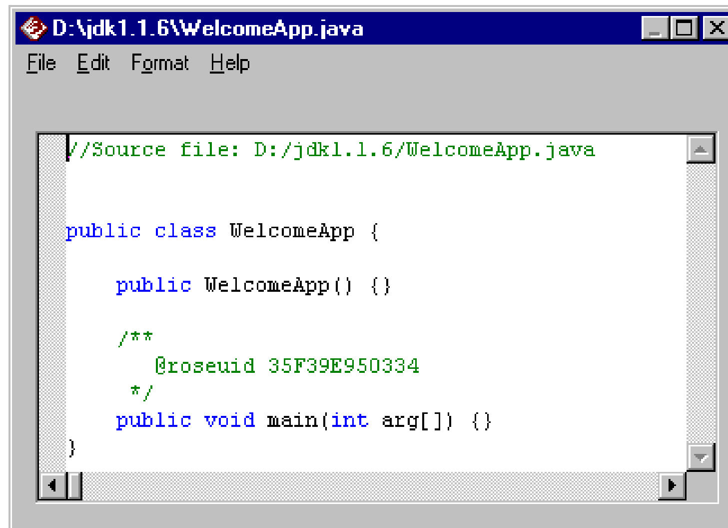
1. In the browser or in a diagram, right-click the **WelcomeApp** component or class, click **Java** > **Generate Java**.
2. Since this is the first time you've generated code for this model, Rational Rose J displays a mapping dialog prompting you to map the file it will generate to a directory in your class path. Click on the class path name to select it, click on the component name to select it, then click **Map**:



3. As soon as the mapping is complete, code generation begins. Check the Rational Rose Log window for information on errors.

View Java Source

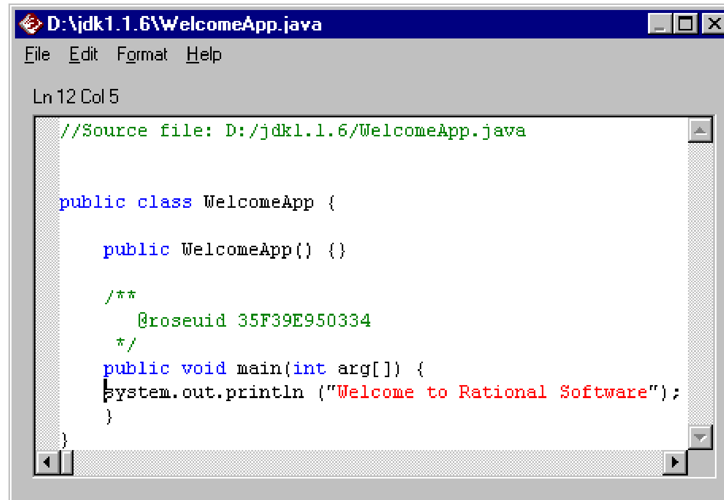
1. Right-click the WelcomeApp component or class, click **Java > Browse Java Source**.
2. Your generated **.java** file is automatically opened and looks like this:



3. Optionally, check Windows Explorer to find the new directory you created and the generated **.java** file it contains.

Edit Java Source

1. Add the command to print the **Welcome to Rational Software!** message (or any message you prefer).



2. Save the file.

Compile and Run

1. Use the javac compiler to compile your completed .java file.
2. Run the application and see what you get. (It should be the message, “Welcome to Rational Software!”)

Appendix B

Java to Rational Rose Mapping Quick Reference

The following table provides a concise mapping between elements of the Java programming language and modeling elements of Rational Rose:

Table 2 Java to Rational Rose Mapping Quick Reference

Java Element	**Rational Rose Model Element**
Package	Package in the Component View.
Import	■ Dependencies between components and packages in the Component View (forward and reverse engineering). ■ Relationships between classes which are not located in the same package. In forward engineering, generates a Java import.
.java files	Component (Module Specification) in the Component View.
Class	Class.
Interface	Class with stereotype of “Interface.”
Implements Relationship	Realizes relationship between Java class (subclass) and Java interface (superclass).
Extends Relationship	■ Generalization relationship between Java classes. ■ Generalization relationship between Java interfaces.

Table 2 Java to Rational Rose Mapping Quick Reference

Field/variable	■ Attribute or Supplier relationship between Rational Rose classes. ■ Java instance variables have static property value set to False. ■ Java class variables have static property set to True.
Method	Operation.
Class Modifiers	Properties on classes (e.g. class.final). Abstract modifier is an element of a Rational Rose class specification.
Field Modifiers	Properties on fields and roles (final, volatile, etc.). Static modifier is an element of a Rational Rose role and attribute specification.
Method Modifiers	Properties on operations (operation.static, operation.final, etc.).
{public, protected, private} access	{public, protected, private} access.
package-level access	Implementation access.

Appendix C

Rational Rose J Model Properties

Introduction

Model properties control the way code is generated from the various elements of a Rational Rose model.

- Project properties provide the high-level control of the code generation process. For example, project properties determine what to do when an error is encountered during code generation, whether to create new directories as needed, and so on.
- Component (module), class, operation, attribute, and role properties control the aspects of code generation that apply to each specific model element, respectively.

This appendix describes all of the Rational Rose J model properties and provides their default values.

You should note, however, that the descriptions of the model properties do not take into account the syntax checking rules which are embedded in the Rational Rose J code generator.

For example, although an attribute of a class can have its Attribute.Final, Attribute.Volatile, and Attribute.Transient properties all marked as True, Rational Rose J will issue a warning when you generate Java because this is not allowed in the Java programming language.

Note: *You can set properties by using the Java custom specifications for class, field, method, and component. Or, you change the properties directly. This is described in the Using Rational Rose manual.*

Project Properties

Project properties provide the high-level control of the code generation process. For example, project properties determine what to do when an error is encountered during code generation, whether to create new directories as needed, and so on.

The following table describes the project properties that apply to Rational Rose J projects:

Table 3 Project Properties

Property	Type	Description	Default
CreateMissingDirectories	Boolean	If True, Rose will attempt to create any undefined directories referenced as part of the Rose model.	True
StopOnError	Boolean	If True, Rose will halt code generation on the first error generated.	False
UsePrefixes	Boolean	If True, Rose will prepend a user specified prefix on generated fields and strip the prefixes during reverse engineering.	False
AutoSync	Boolean	If True, Rose J automatically initiate code generation whenever you create, delete, or modify a Java element in your model.	False

Table 3 Project Properties

Editor	Enum	Sets the editor to launch when browsing Java source code from within Rose. If set to BuiltIn, Rose opens its own editor. Use WindowsShell to enable another editor such as Notepad. (Make sure your file associations are set up correctly for other editors.)	BuiltIn
VM	Enum	Sets the Java environment (virtual machine). Sun is based on Sun Microsystem's Java Developer's Kit (JDK). If you select Microsoft and you have Visual J++ installed, Rose can establish a link between your model and your VJ++ project when you generate or reverse engineer code.	Sun
ClassPath	string	Use the ClassPath to identify the root directory Rose will use for the *.java files it creates when you generate (forward engineer) code from a model or reverse engineer *.java files into a model.	none
InstanceVariablePrefix	String	Prefix to prepend on instance variables if UsePrefixes = True	m_
ClassVariablePrefix	String	Prefix to prepend on class variables if UsePrefixes = True	s_

Table 3 Project Properties

DefaultAttributeDataType	String	Default data type string to use for attributes if Attribute.Type specification field is blank.	int
DefaultOperationReturnType	String	Default return type string to use for operations if Operation.ReturnType specification field is blank.	Void
NoClassCustomDlg	Boolean	When set (True), Rose J displays the standard Rose specifications for Java elements in your model. It disables the Rose J custom specifications.	False
GlobalImports	String	When you enter one or more values for this property, Rose automatically generates import statements for the classes/packages you identify.	(blank)
OpenBraceClassStyle	Boolean	This setting lets you indicate whether or not an opening brace starts on a new line in generated Java code. By default this setting is enabled and opening braces start on a new line. If unchecked (or set to False) an opening brace can be inline.	True

Table 3 Project Properties

OpenBraceMethodStyle	Boolean	This setting lets you indicate whether or not an opening brace starts on a new line in generated Java code. By default this setting is enabled and opening braces start on a new line. If unchecked (or set to False) an opening brace can be inline.	True
UseTabs	Boolean	Use this setting if you want to use tabs to indent generated Java code. (Your alternative is to use spaces.) Use the NumbertoUse setting (or the SpacingItems model property) to specify the number of tabs to use. The default is three tabs. Note that using spaces (not tabs) for indenting is the default setting.	False
UseSpaces	Boolean	Use this setting if you want to use spaces to indent generated Java code. (Your alternative is to use tabs.) Use the NumbertoUse setting (or the SpacingItems model property) to specify the number of spaces to use. The default is three spaces. Note that using spaces for indenting is the default setting.	True

Table 3 Project Properties

SpacingItems	Enum	Use this setting to specify how many spaces or tabs to use when indenting generated Java code. (This setting controls the Use Tabs and Use Spaces settings.) The default is three.	3
RoseDefaultCommentStyle	Boolean	Use this setting to enable the standard Rose comment style in generated code.	True
AsteriskCommentStyle	Boolean	Use this setting to enable the standard Java comment style in generted code.	False
JavaCommentStyle	Boolean	Use this setting to enable the Javadoc tags that Rose J will insert into generated code. By enabling this setting, you enable the @tag options.	False
JavadocAuthor	Boolean	Enables the @author Javadoc tag.	True
JavadocDeprecated	Boolean	Enables the @deprecated Javadoc tag.	False
JavadocException	Boolean	Enables the @exception Javadoc tag.	True
JavadocParam	Boolean	Enables the @param Javadoc tag.	True
JavadocReturn	Boolean	Enables the @return Javadoc tag.	True
JavadocSee	Boolean	Enables the @see Javadoc tag.	False

Table 3 Project Properties

JavadocSerial	Boolean	Enables the @serial Javadoc tag.	False
JavadocSerialdata	Boolean	Enables the @serialdata Javadoc tag.	False
JavadocSerialfield	Boolean	Enables the @serialfield Javadoc tag.	False
JavadocSince	Boolean	Enables the @since Javadoc tag.	False
JavadocVersion	Boolean	Enables the @version Javadoc tag.	False
JavadocLink	Boolean	Enables the @link Javadoc tag.	False

Class Properties

During Java source generation, Rational Rose model classes are mapped to Java classes. Class properties control the details of the source generation for each Java class.

The following table describes the properties that apply to Rational Rose model classes being generated into Java classes:

Table 4 Class Properties

Property	Type	Description	Default
Final	Boolean	If True, Rational Rose will include a final modifier in the source code for the class.	False
Static	Boolean	If True, Rational Rose will include a static modifier for the class.	False
GenerateDefault Constructor	Boolean	If True, Rational Rose will include a default constructor in the source code for the class.	True
ConstructorIs	Enum	{public, protected, private}. Represents the access control for the constructor to use if GenerateDefault Constructor = True.	public
GenerateFinalizer	Boolean	If True, Rational Rose will include a finalizer in the source code for the class.	False
GenerateStatic Initializer	Boolean	If True, Rational Rose will include a static initializer in the source code of the class.	False
GenerateInstance Initializer	Boolean	If True, Rational Rose will include an instance initializer in the source code of the class.	False

Operation Properties

During Java source generation, Rational Rose model operations are mapped to Java methods. Operation properties control the details of the source generation for each Java method.

The following table describes the properties that apply to Rational Rose model operations being generated into Java methods:

Table 5 Operation Properties

Property	Type	Description	Default
Abstract	Boolean	If True, Rational Rose will attempt to include an abstract modifier in the source code for the operation.	False
Static	Boolean	If True, Rational Rose will attempt to include a static modifier in the source code for the operation.	False
Final	Boolean	If True, Rational Rose will attempt to include a final modifier in the source code for the operation.	False
Native	Boolean	If True, Rational Rose will attempt to include a native modifier in the source code for the operation.	False
Synchronized	Boolean	If True, Rational Rose will attempt to include a synchronized modifier in the source code for the operation.	False

Attribute Properties

During Java source generation, Rational Rose model attributes are mapped to various types of Java modifiers. Attribute properties control the details of the source generation for each Java modifier.

The following table describes the properties that apply to Rational Rose model attributes being generated into Java modifiers:

Table 6 Attribute Properties

Property	Type	Description	Default
Final	Boolean	If True, Rational Rose will attempt to include a final modifier in the source code for the attribute.	False
Transient	Boolean	If True, Rational Rose will attempt to include a transient modifier in the source code for the attribute.	False
Volatile	Boolean	If True, Rational Rose will attempt to include a volatile modifier in the source code for the attribute.	False
PropertyType	String	Supports values for Java Beans, including NotAProperty, Simple, Bound, and Constrained.	NotAProperty

Table 6 Attribute Properties

IndividualChangeMgt	Boolean	Lets you specify the registration mechanism for a bean property.	False
Read/Write	String	Determines if Rational Rose will generate a get and/or set method for a bean.	Read&Write
GenerateFullyQualifiedTypes	Boolean	When set, Rose J generates fully qualified path names for all type classes.	False

Component (Module Body/Module Specification) Properties

During Java source generation, Rational Rose model components (modules) are mapped to Java compilation units. Component properties control the details of the source generation for each Java compilation unit.

The following table describes the properties that apply to Rational Rose model components being generated into Java compilation units:

Table 7 Component Properties

Property	Type	Description	Default
CopyrightNotice	String	Copyright message that you define and that is included in the header of a generated .java file.	<<blank>>
CmIdentification	String	Represents a user-defined version control identification tag to include in the header of the compilation unit, if any.	<<blank>>

Role Properties

During Java source generation, Rational Rose model roles are mapped to various types of Java fields and modifiers. Role properties control the details of the source generation for each Java compilation unit.

The following table describes the properties that apply to Rational Rose model components being generated into Java compilation units:

Table 8 Role Properties

Property	**Type**	**Description**	**Default**
ContainerClass	String	Specifies the collection/container class name to use if the supplier cardinality of the relationship is unbounded. If blank, singly-dimensioned array is used.	<<blank>>
InitialValue	String	Specifies the initial value for the Java field, if any.	<<blank>>
Final	Boolean	If True, Rational Rose will attempt to include a final modifier in the source code for the relationship.	False
Transient	Boolean	If True, Rational Rose will attempt to include a transient modifier in the source code for the relationship.	False
Volatile	Boolean	If True, Rational Rose will attempt to include a volatile modifier in the source code for the relationship.	False

Table 8 Role Properties

PropertyType	String	Supports values for Java Beans, including NotAProperty, Simple, Bound, and Constrained.	NotAProperty
Individual ChangeMgt	Boolean	Lets you specify the registration mechanism for a bean property.	False
Read/Write	String	Determines if Rational Rose will generate a get and/or set method for a bean.	Read&Write

Index

Symbols

~jav backup files 58

A

abstract modifier
  class 20
  methods 33
Abstract property 91
activating the link to Visual J++ 5
addPropertyChangeListener 48, 49
addVetoableChangeListener 50
analyzing Java source code 4
arguments for methods 33
arrays 29
assigning classes to components
  how to 18
assigning Java classes to components 55
Association Specification 30
associations and variables (fields) 24
AsteriskCommentStyle property 88
attribute 22
attribute properties 92
Automatic Synchronization Mode 58
Automatic Synchronization Mode property 57
automatically generating code 57
AutoSync property 84

B

backing up your source 57
Beans, Java 46
bound property (Java Beans) 48
braces, how formatted in generated code 57
browsing generated source 64
BuiltIn Editor
  about 63
  shortcut keys 63
Builtin Editor 58

C

cab files 40
  drag and drop into a model 4, 67
cardinality 30
change management listener 47
Change Management, Individual 47
checking syntax 56
class
  abstract 38
  assigning to a component 18
  comments 20
  constructor visibility 20
  default constructor 20
  exception 33
  extends 20
  final 20

finalizer 20
generators 20
how modeled 17
how to create 17
implements 20
interface 38
mapped to Java classes 3
modifiers 20
static 20
static initializer 20
stereotype 38
class files
cannot browse 65
drag and drop into a model 4
Class Path 45
class path, setting 14
class properties 90
ClassPath property 85
classpath, checking before code generation 56
ClassVariablePrefix property 85
CmIdentification 41
CmIdentification property 94
code generation 55
from a class diagram 61
loading controlled units before 59
mapping components 60
code, displaying for a method 35
comments
for a class 20
for a variable (field) 28
for components 41
for methods 34
comments in generated code 57
completing generated source 62
Component Mapping dialog 60
component packages 40, 44, 45
component properties 94
Component Specification 14, 41
components
adding a copyright 41
and reverse engineering 43
as .java files 40
assigning classes to 55
CmIdentification 41
code generation 42
comments 41
imports 41
constrained property (Java Beans) 49
constructor visibility 20
ConstructorIs property 90
container class 28, 31
ContainerClass property 95
controlled units
loading before code generation 59
converting String to a primitive data type 22
Copyright 41
Copyright Notice 94
create
class 17
Create Missing Directories property 45, 56, 84

D

default constructor 20
default notation 55
default variable prefixes 29
DefaultAttributeDataType property 86
DefaultOperationReturnType property 86
design-level properties 84
disabling Rose J specifications 13
display generated source 64
drag and drop 4, 67

E

Editor property 63, 85
exceptions 33
extends 36
class 20

F

field
- comments 28
- container class 28
- final 28
- primitive type 22
- reference type 23
- static 28
- transient 28
- volatile 28

Field Specficiation 28
Field Specification 13
fields
- arrays 29
- container class 31
- default prefixes 29
- modifiers 28
- vectors 31
- with reference types 23

file not associated with application 65
final modifier
- class 20
- methods 33
- variable (field) 28

Final property 90, 91, 92, 95
finalizer, class 20
firePropertyChange 49
formatting generated code 13, 57
forward engineering 55
- component diagrams used for 3
- defined 3
- generating from class diagram 61
- generating from component diagram 61
- of packages 3
- overview 55
- viewing and completing generated source 62

G

generalization relationship 36
Generate Fully Qualified Types property 93
generated source
- cannot browse 65

GenerateDefaultConstructor property 90
GenerateFinalizer property 90
GenerateInstanceInitializer property 90
GenerateStaticInitializer property 90
generating classes to a single file, 18
generating code automatically 57
generating code from a model 55
generating code to a Visual J++ project 5, 62
generating comments in code 13, 57
generating Java source
- from class diagram 61
- from component diagram 61

generators, class 20
GlobalImports property 86

H

HTML documentation from code 13

I

implements 37
- class 20

imports 41, 42, 43, 46, 49
indentation in generated code 57
Individual Change Management 47
Individual ChangeMgt property 96
IndividualChangeMgt property 93
InitialValue property 95
inner class 39
InstanceVariablePrefix property 85
integrating with IBM's VisualAge for Java 6
integrating with Visual J++ 5, 57
interface 38

J

jar files 40
- drag and drop into a model 4
- dragging and dropping into a model 67

Java Bean properties 28
Java Beans 46
- bound property 48
- constrained property 49
- Individual Change Managemen 47
- simple 47

Java Class
- constructor visibility 20
- default constructor 20
- extends 20
- generators 20
- how modeled 17
- implements 20
- modifiers 20
- static initializer 20

Java Component Specification 41
Java Field Specification 28
- Java Bean properties 46

java files 4, 40
Java Method Specification 33, 34
- Code sheet 35

Java model properties
- overview 83

Java Project Specification
- Create Missing Directories 45
- directories, create missing 45
- editor setting 63

Java Specifications
- about 12

Java to UML mapping
- list of Java constructs 9

JavaCommentStyle property 88
Javadoc tags in generated code 13
Javadoc, generating 57
JavadocAuthor property 88
JavadocDeprecated property 88
JavadocException property 88
JavadocLink property 89
JavadocParam property 88
JavadocReturn property 88
JavadocSee property 88
JavadocSerial property 89
JavadocSerialdata property 89
JavadocSerialfield property 89
JavadocSince property 89
JavadocVersion property 89

L

language, default 14
listener 47

M

mapping components for code generation 60
method
- displaying code for 35

method properties 91
Method Specification 14, 33, 34
- Code tab 35

methods
- abstract 33
- arguments 33
- comments 34
- final 33
- how modeled 33
- native 33
- static 33
- synchronized 33
- throws 33
- visibility 33

Microsoft Visual J++ 5
model properties
- about 10
- overview 83

modifiers
- class 20

method 33
variables (fields) 28

N

native modifier 33
Native property 91
nested class 39
NoClassCustomDlg property 86
Notation model property 14
notation, setting the default 55

O

OpenBraceClassStyle property 86
OpenBraceMethodStyle property 87
operation properties 91

P

package 44
forward engineering a 3
packages 40
Path Map variables 17
primitive data types 22
project properties 84
Project Specification, about 13
properties
attribute 92
class 90
component 94
design-level 84
methods 91
operation 91
project 84
role 95
properties for code generation 56
properties, model
about 10
overview 83
PropertyChangeSupport 48, 49
PropertyType property 92, 96
public modifier
class 20

Q

quick reference
Java to Rose mapping 81

R

Read/Write property 93, 96
Java Beans 47
realization relationship 37
reference type 23
registration class 47
registration mechanism (Java Beans) 47
removePropertyChangeListener 48, 49
removeVetoableChangeListener 50
Resolve Pathmap Variables setting 17
reverese engineering a Visual J++ project 69
reverse engineering 67
components 43
defined 4
drag and drop files 4
procedure 67
role properties 95
roles 22, 29
Rose Specifications 11
as the default 13

S

setting comment style 13
setting the default notation 14
setting the Virtual Machine property 5
setting up the link to Visual J++ 57
simple property type (Java Beans) 47
SpacingItems property 88
specifications, Java
about 12
Specifications, Rose 11

starting Visual J++ from Rose 62
static initializer 20
static modifier
  class 20
  methods 33
  variable (field) 28
Static property 91
stereotype 38
Stop on Error property 56
StopOnError property 84
String variables 22
Support Individual Change Management 47
synchronized modifier 33
Synchronized property 91
synchronizing code generation 57
syntax, checking 56

T

throws 33
transient modifier 28
Transient property 92, 95
type
  primitive 22
  return 33

U

UsePrefixes property 84
UseSpaces property 87
UseTabs property 87
using the BuiltIn Editor 63

V

variable
  arrays 29
  comments 28
  container class 28, 31
  default prefixes 29
  final 28
  modifiers 28
  primitive type 22
  reference type 23
  static 28
  transient 28
  vectors 31
  volatile 28
  with reference types 23
vectors 29, 31
version control identification tag 41
VetoableChangeSupport 50
viewing generated source 62
Virtual Machine property 5, 57
visibility
  class 28
  methods 33
visibility, constructor 20
Visual J++ 5
Visual J++, generating code 62
Visual J++, integrating with Rose 57
Visual J++, starting Rose from 5
VisualAge for Java 6
VM property 85
volatile modifier 28
Volatile property 92, 95

W

Windows Shell property 59
WindowsShell setting for editor 63

Z

zip files 40
  dragging and dropping into a model 67
  draging and dropping into a model 4