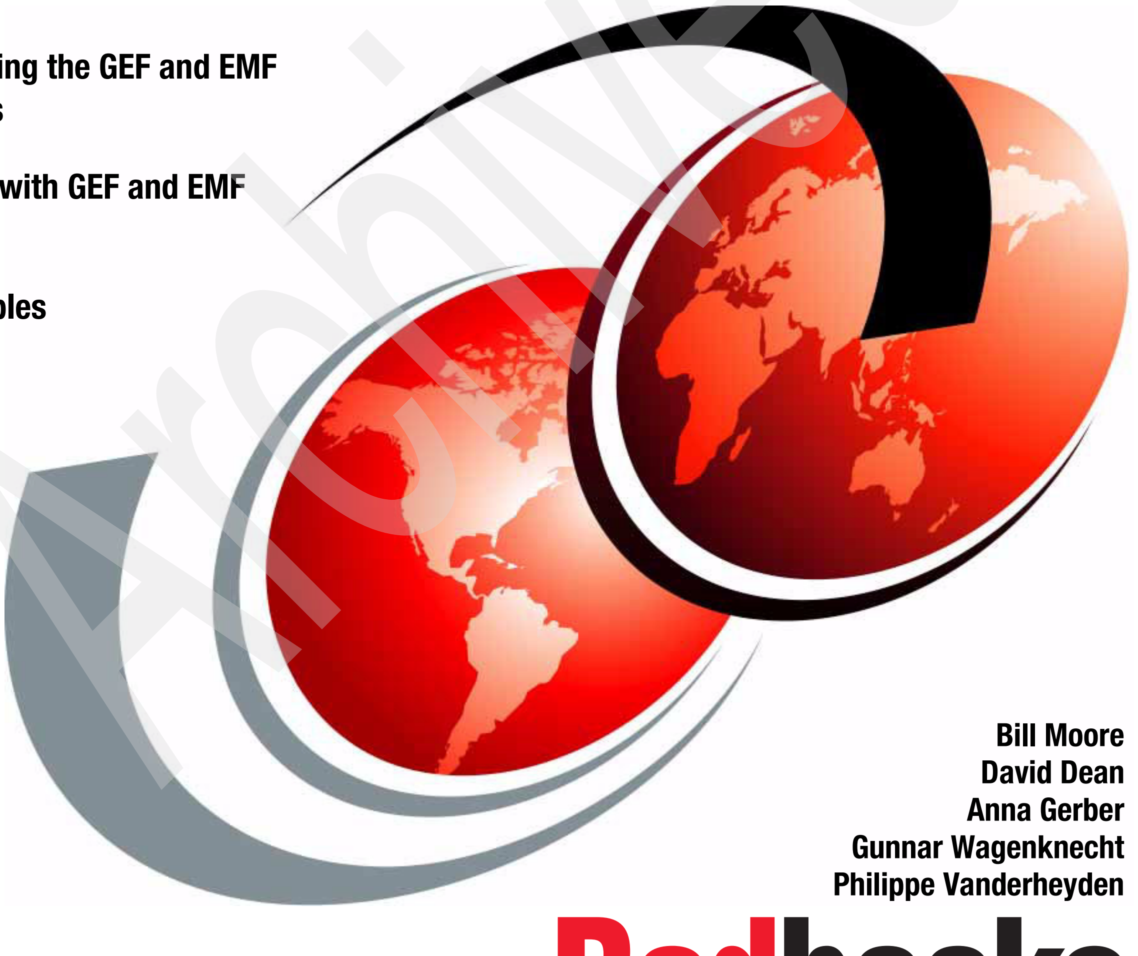
Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework

Understanding the GEF and EMF frameworks

Developing with GEF and EMF

Code examples



Bill Moore
David Dean
Anna Gerber
Gunnar Wagenknecht
Philippe Vanderheyden

International Technical Support Organization

Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework

February 2004

SG24-6302-00

Note: Before using this information and the product it supports, read the information in "Notices" on page vii.

First Edition (February 2004)

This edition applies to Version: 2.1.1 of the Eclipse Platform, Version 1.1.0 of the Eclipse Modeling Framework (EMF), and Version 2.1.1 of the Graphical Editing Framework (GEF) on Microsoft Windows.

Contents

Notices

Trademarks

Preface

This redbook is written for developers who use the Eclipse SDK to develop plug-in code. It is intended for a technical readership and for developers who already have good knowledge and experience in Eclipse plug-in development. We expect that you understand the concepts of Eclipse views and editors, and have some familiarity with Draw2D.

In this redbook, we examine two frameworks that are developed by the Eclipse Tools Project for use with the Eclipse Platform:

- The Graphical Editing Framework (GEF)
- The Eclipse Modeling Framework (EMF)

Important: This redbook covers both the Graphical Editing Framework and the Eclipse Modeling Framework, but readers should remember that these frameworks can be used separately and there is no dependency between the two frameworks. We do write about using GEF and EMF together, but please remember that this is not required, and many applications you develop will not require both GEF and EMF.

We provide a high level introduction to these frameworks so that Eclipse plug-in developers can consider whether the frameworks will be useful for the requirements of their particular development; then we provide helpful tips and techniques for writing code that uses GEF and EMF. Finally, we implement a more detailed example to illustrate a GEF editor that uses an EMF model.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Raleigh Center.

William Moore (Bill) is a WebSphere specialist at the International Technical Support Organization, Raleigh Center. He writes extensively and teaches IBM® classes on WebSphere and related topics. Before joining the ITSO, Bill was a Senior AIM Consultant at the IBM Transarc lab in Sydney, Australia. He has 18 years of application development experience on a wide range of computing platforms and using many different coding languages. He holds a Master of Arts degree in English from the University of Waikato, in Hamilton, New Zealand. His current areas of expertise include application development tools, object-oriented programming and design, and e-business application development.

David Dean is a Technical Lead at Chordiant in Cupertino, California. For the last two years he has been focused on Eclipse plug-in development and in building a GEF-based workflow editor. His twenty years of software development experience include medical imaging, process control, telephony, finance, and Web applications. David's interests include user interfaces, graphics, and software development tools. He holds a BA degree in Biology from the State University of New York at Albany, and did post-graduate studies in Historic Preservation Planning at Cornell University.

Anna Gerber is currently a Research Scientist at the Distributed Systems Technology Centre (DSTC) in Brisbane, Australia. Anna's research interests include Enterprise Modelling; in particular, model-driven development techniques and generation of tools such as domain-specific graphical editors from models.

Gunnar Wagenknecht is a software developer at Intershop AG in Jena, Germany.He has professional experience in developing Java™ Enterprise applications using the J2EE framework, and he developed a visual editor for modelling business processes during the last year. He just finished his thesis and is going to get a Bachelor's degree in Practical Computer Science from the Business Academy Thuringia in Gera, Germany after finishing the residency. His areas of expertise include object-oriented software architectures and Web application development. He has written extensively on GEF topics.

Philippe Vanderheyden is an IT Architect who has been working with object-oriented (OO) technologies for many years. Philippe has been working on a variety of projects, ranging from document publishing systems to financial application development and monitoring. His areas of interest include OO modelling, distributed enterprise systems, and Web-based application design and real-time transactional systems. Philippe has a good knowledge of the Java programming language, and Java-related technologies (JDBC, servlets, XML, JSP, etc.). His recent work has included building enterprise applications using the Enterprise Java Beans component model and the J2EE framework in WebSphere 5.0 cluster environment. Philippe is comfortable working with a diverse range of technologies and platforms. He has extensive experience of the UNIX® OS and has also worked for many years with Object Oriented languages (Java, Smalltalk, and C++).

Authors: Gunnar Wagenknecht, Anna Gerber, Philippe Vanderheyden

Author: David Dean

Thanks to the following people for their contributions to this project:

Randy Hudson
Pat McCarthy
IBM Raleigh

Jim D'Anjou
IBM San Jose

Yvonne Lyon, editor
International Technical Support Organization, San Jose Center

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks™ to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an Internet note to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HZ8 Building 662
  P.O. Box 12195
  Research Triangle Park, NC 27709-2195

Part 1

EMF and GEF introduced

In this part of the book, we describe the basics of the Graphical Editing Framework(GEF) and Eclipse Modeling Framework(EMF).

1

Introduction to EMF

In this chapter, we introduce the Eclipse Modeling Framework (EMF). We mention most of the sources of information that are available on the subject, and we build a simple model as a practical demonstration of the use of EMF.

1.1 What is the Eclipse Modeling Framework?

Application development generally starts with consideration of the design model, then moves to more user interface oriented tasks. The Eclipse Modeling Framework is designed to ease the design and implementation of a structured model. The Java framework provides a code generation facility in order to keep the focus on the model itself and not on its implementation details. The key concepts underlying the framework are: meta-data, code generation, and default serialization.

1.1.1 Positioning of the framework

EMF was started as a Meta Object Facility (MOF) of the Object Management Group (OMG) implementation and has evolved to what it is now. EMF is an enhancement of MOF2.0. EMF is open source code that enhances the MOF 2.0 Ecore model and restructures its design in a way that is easy for the user.

The Eclipse Modeling Framework is part of the Model Driven Architecture (MDA). It is the current implementation of a portion of the MDA in the Eclipse family tools. The idea behind MDA it is to be able to develop and manage the whole application life cycle by putting the focus to the model. The model itself is described in a meta-model. Then, by using mappings, the model is used to generate software artefacts, which will implement the real system.

Two types of mappings are defined: Metadata Interchange, where documents like XML, DTD, and XSD are generated; and Metadata Interfaces, which target Java or any other language and generate IDL code. MDA is currently under the standardization process at the OMG.

1.1.2 Objectives

In this section we explain the main purpose of EMF and what it can currently be used for.

The problems EMF solves

EMF can be used to describe and build a model. Based on that definition, Java code can be generated and enhanced by the addition of higher level Java code. This implemented model can be used as the basis for any Java application development.

When not to use EMF

At the moment, EMF implements a subset of the MDA approach. As such, it does not contain all the mappings we would need to make and deploy an application at a company wide level, where XML, EAI, EJBs, Web services, and other technologies have to be combined.

1.1.3 Where to find documents and resources

EMF is still under development, but several sources of information are available These include:

- **The EMF project page:**

  EMF is one project of the Eclipse Tools Project, which is part of the global Eclipse Project, `(http://www.eclipse.org)`. EMF is directly accessible at the URL:

  `http://www.eclipse.org/emf`

  Available services range from code access and documents publishing, to community support, online code access using CVS, packaged code download, articles, user guides, tutorials, mailing list, newsgroup, and more.

- **Newsgroup:**

  The newsgroup server is `news.eclipse.org`. The newsgroup name for EMF is eclipse.tools.emf. It shows EMF relationship within the Eclipse Tools project.

- **Mailing list:**

  The mailing list for the EMF project is emf-dev@eclipse.org.

Note: You should send your questions to the newsgroup rather than to the mailing list.

1.2 Framework basics

This section provides some basic information about the Eclipse Modeling Framework to help you get it up and running.

1.2.1 Prerequisites

When we wrote this redbook, the current version of EMF was v1.1.0. A valid Eclipse product installation is a prerequisite to use EMF. As of EMF v1.0.2, Eclipse v2.1 is required. For the purpose of writing the redbook, Eclipse v2.1 and EMF v1.1.0 have been used.

1.2.2 Product installation

Eclipse product installation is straightforward. You extract the content of the downloaded archive, which is platform dependent, to a folder of your choice. Depending on the operating system, double-click the Eclipse icon, or run the corresponding shell command to complete the installation process and launch the Eclipse Platform.

EMF is packaged in three parts: the first one is the runtime, the second contains the documentation, and the third contains the source code.

EMF framework installation

Download the EMF Runtime archive (for example, `emf_1.1.0_20030620_1105VL.zip`) and extract the content to the Eclipse folder.

EMF documentation installation

Download the EMF Documentation archive (`for example`, `emf.doc_1.1.0_20030620_1105VL.zip`) and extract the content to the Eclipse folder.

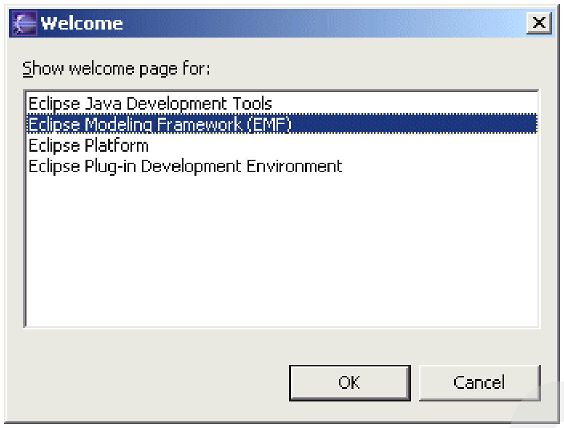
Note: If Eclipse was running, while doing EMF and document installation, Eclipse will need to be restarted for changes to take effect.

1.2.3 Getting help in Eclipse

EMF help can be found in the Eclipse help system.

The welcome page

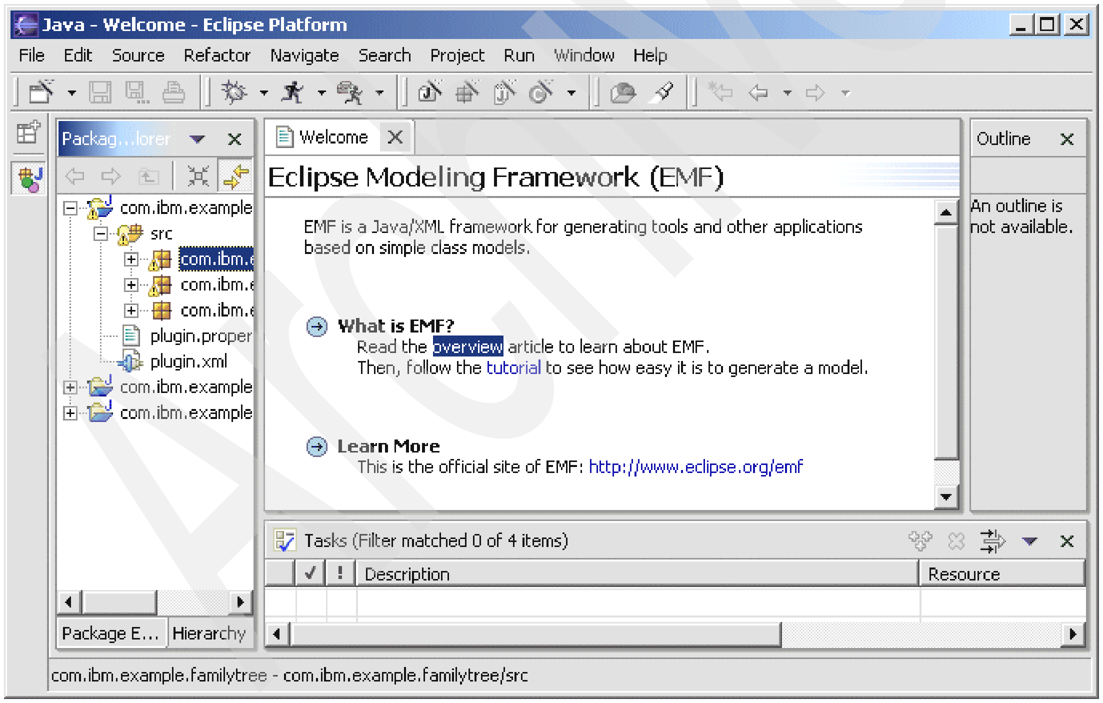
1. The welcome page is the main entry point to the EMF documentation. In Eclipse, click **Help -> Welcome..**. to list available welcome pages as shown in Figure 1-1.



Eclipse Java Development Tools
Eclipse Modeling Framework (EMF)
Eclipse Platform
Eclipse Plug-in Development Environment

Figure 1-1 Welcome page window

2. Select the Eclipse Modeling Framework welcome page from the list and click **OK**. Figure 1-2 shows the EMF welcome page that will be displayed.



Eclipse Modeling Framework (EMF)

EMF is a Java/XML framework for generating tools and other applications based on simple class models.

What is EMF?
Read the overview article to learn about EMF.
Then, follow the tutorial to see how easy it is to generate a model.

Learn More
This is the official site of EMF: http://www.eclipse.org/emf

com.ibm.example.familytree - com.ibm.example.familytree/src

Figure 1-2 EMF welcome page

Note: The EMF documentation package must be installed before the links in EMF welcome pages are clickable.

The help perspective

EMF help is also accessible directly from **Help -> Help contents**. Figure 1-3 shows the help available in the EMF Programmers Guide, which includes an EMF overview, a user guide, and an EMF.Edit section.

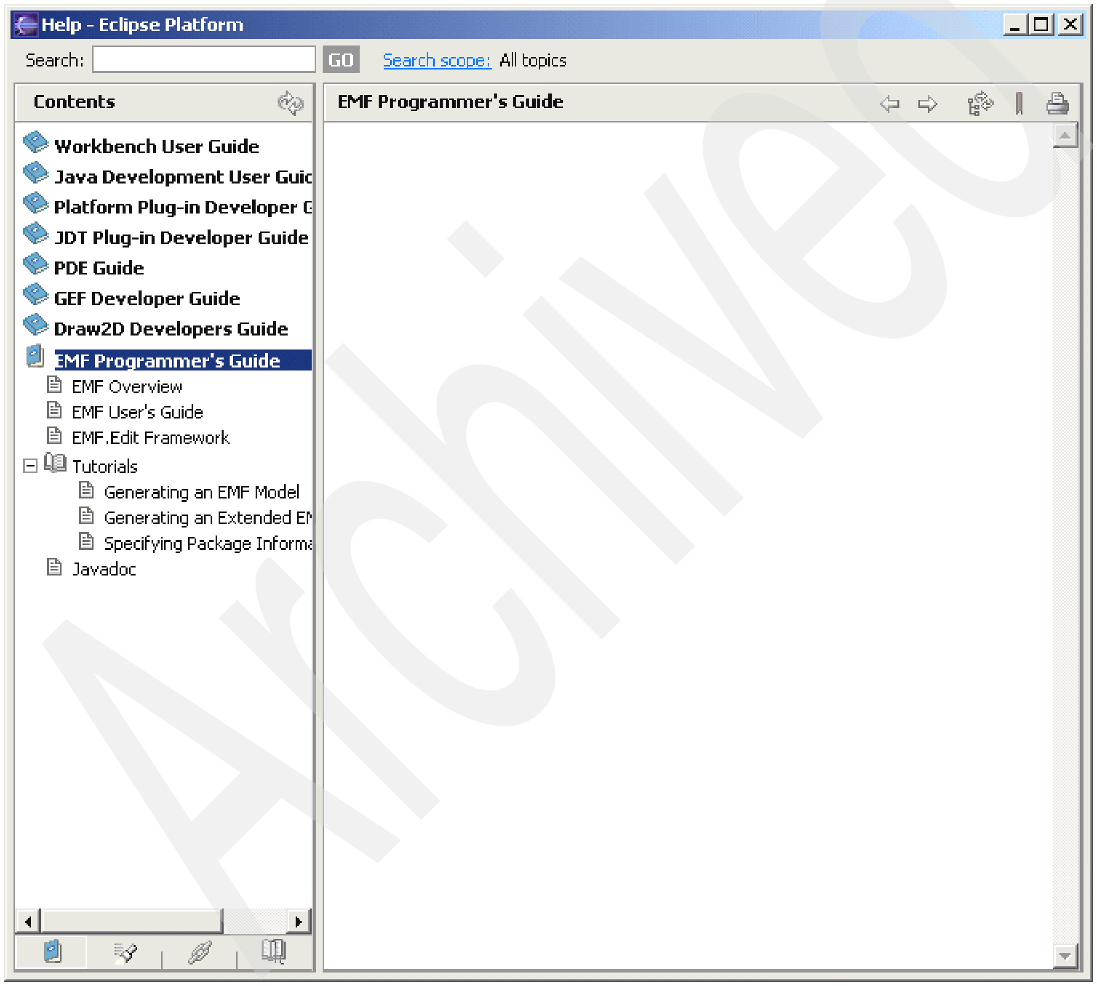


Figure 1-3 EMF help

1.3 Building a simple model

In this section, we build a simple but realistic model. The purpose is to demonstrate the main steps of the process. Later in our redbook, we use the Graphical Editing Framework (GEF), to build a workflow application on top of this model. The workflow editor will help us to create and visualize the content of the model. For more information, the application requirements and design can be found in “Sample application requirements” on page 188.

The model

Before starting to describe the modelling process using Eclipse and EMF, we need to understand the complete underlying UML model that we will build. This is shown in Figure 1-4 and discussed in more detail in “The workflow model” on page 192.

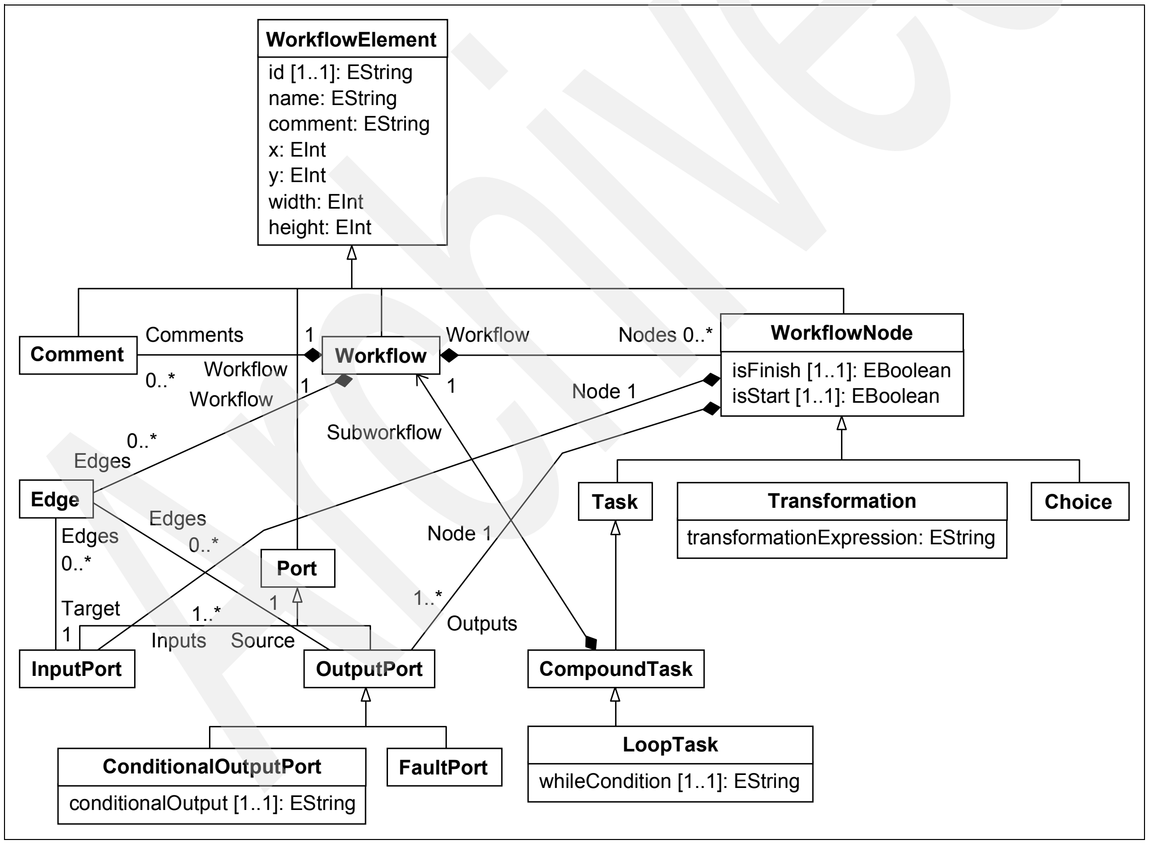


Figure 1-4 The complete UML model

1.3.1 Different ways of making the model

In EMF, the model can be created in three different ways:

- Write the XMI file directly.
- Export the XMI file, from tools like Rational® Rose® and the Omondo EclipseUML plug-in and load it into our project.
- Annotate Java interfaces with model properties.

To illustrate how to create a model, we demonstrate the use of the Omondo EclipseUML plug-in to generate the XMI and also show the use of the Java interface annotation mechanism.

1.3.2 The EclipseUML plug-in

The main advantage of UML is that it allows us to work at a very high level. In an EMF class diagram, we create classes and interfaces, we give them attributes and methods, and we set up their relationships.

plug-in installation

Omondo's EclipseUML plug-in can be downloaded from the site:

```
http://www.eclipseuml.com
```

The current version is 1.2.1. The installation is an executable jar file. On Microsoft® Windows®, double-click its icon. On other operating systems, run the following command:

```
java -jar eclipseuml-installer_1.1.4.jar
```

Install the product in the same folder you installed the Eclipse product.

Note: In our case, we did not install the versions of GEF and EMF that are provided with the EclipseUML plug-in, because we wanted to use the latest versions of GEF and EMF.

1.3.3 Initial project setup

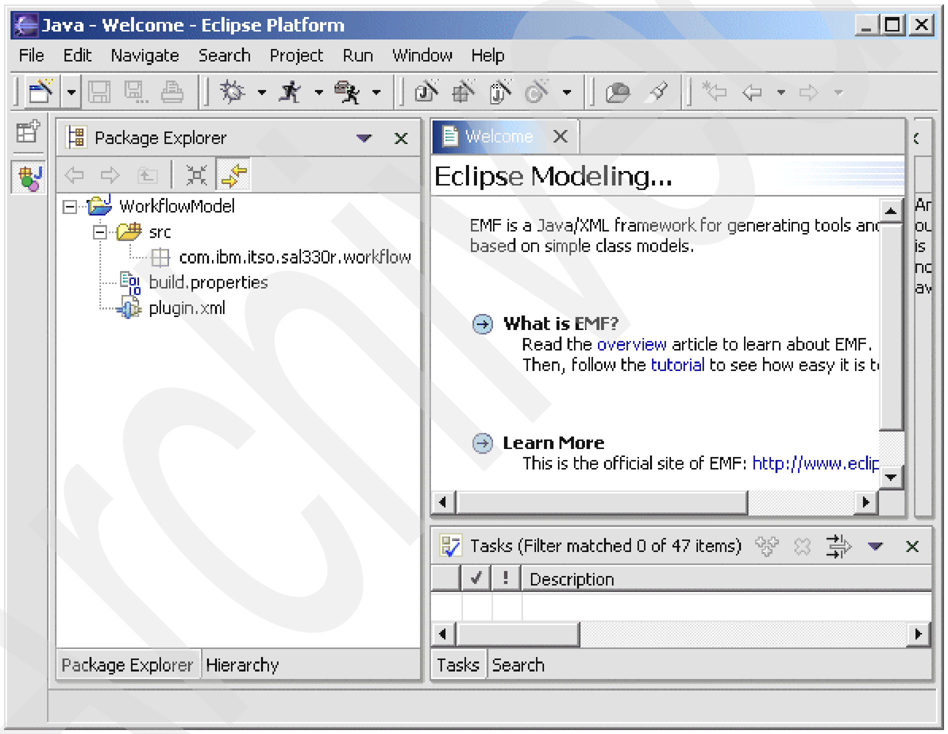
Before doing the modeling itself, we need to create an Eclipse project environment to contain all the items that we are going to produce. The steps to take are as follows:

1. Create a new project:

   a. Click **File -> New -> Other...**, select **Plug-in Development -> Plug-in Project**, and click **Next**.

   b. Enter a project name, for example, `WorkflowModel`, and click **Next**.
   c. Select **Create a Java project**, and click **Next**.
   d. Select **Create a blank plug-in project**, and click **Finish**.

2. Create a Java Package:
   a. Click **File -> New -> Other...**, select **Java -> Package**, and click **Next**.
   b. Click **Browse...** to select the `src` folder in the `WorkflowModel` project.
   c. Enter a package name, for example, `com.ibm.itso.sal330r.workflow`, and click **Finish.** Figure 1-5 shows a view of the Eclipse workbench after we have completed our initial setup tasks.


Java - Welcome - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Package Explorer
WorkflowModel
src
com.ibm.itso.sal330r.workflow
build.properties
plugin.xml
Package Explorer
Hierarchy
Welcome
Tasks (Filter matched 0 of 47 items)
Description
Tasks
Search

Eclipse Modeling...

EMF is a Java/XML framework for generating tools and based on simple class models.

What is EMF?
Read the overview article to learn about EMF. Then, follow the tutorial to see how easy it is t

Learn More
This is the official site of EMF: http://www.eclip

Figure 1-5 Initial project setup

Note: Our project must be a plug-in project, but it also needs to be a Java project, in order to allow package creation. If we had selected **Create a simple project**, package creation would not have been possible. Creating an EMF Project directly is another way to achieve the same result.

1.3.4 Modeling using the EclipseUML plug-in

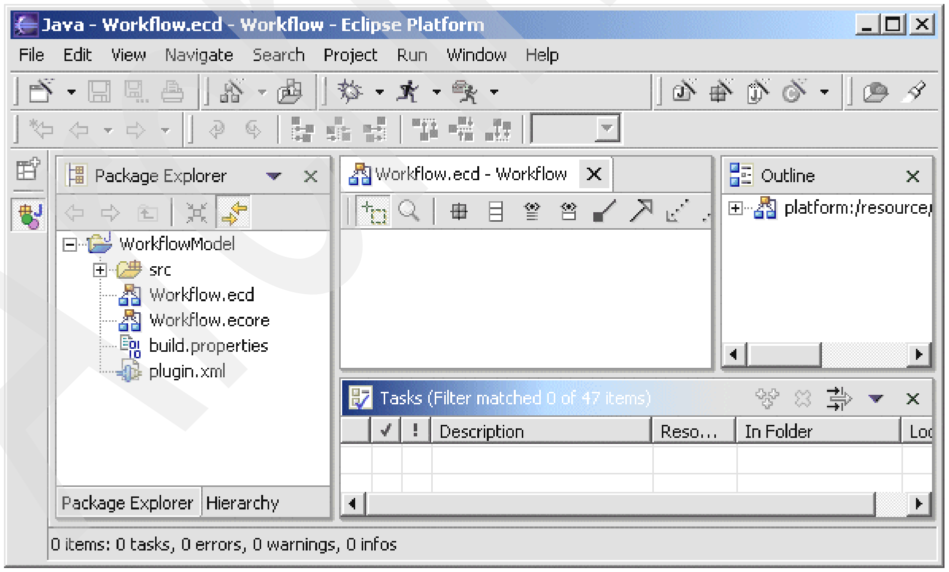
During our simple model creation, we iterate several times to achieve what we think is a good design. The graphical facilities of the EclipseUML plug-in are a great help during this process, and each intermediate diagram was used as a good start to support the next iteration of our modelling.

EMF class diagram creation

The whole model is contained in one EMF class diagram. Here are the steps to create this diagram:

1. Click **File -> New -> Other...**, select **EMF Diagrams -> EMF Class Diagram**, click **Next**:
    a. Choose the parent folder, for example, `WorkflowModel` project.
    b. Enter an EMF model file name, file extension is `ecd`, for example, `Workflow.ecd`
    c. Enter a package name, for example, `com.ibm.itso.sal330r.workflow`
    d. Check the association box, click **Finish**. See Figure 1-6.

Two files have been created: `Workflow.ecd`, which contains the class diagram; and `workflow.ecore`, which contains the core model definition.



0 items: 0 tasks, 0 errors, 0 warnings, 0 infos

Figure 1-6 EMF class diagram window

Notes:

1. In the Eclipse,new EMF class diagram dialog, the package name in the advanced section corresponds to the EMF EPackage.
2. With the EclipseUML plug-in, two types of diagram can be created: UML and EMF models. The file for UML extension is .u?? (`ex: Workflow.ucd`) while the extension for EMF is .e?? (`ex: Workflow.ecd`). Available modeling operations and data types are adapted to the type of file you are working in. Remember that if you work with an EMF model, only .e?? files and the associated editors give you access to EMF functionality.

Class diagram modeling

From the modeling point of view, the class diagram is complete, once we have defined a set of classes (EMF interfaces), and the relationships between them.

Interface design

We first create the root interface, which is called `WorkflowElement`, then we implement the `WorkflowNode` hierarchy, the `Port` hierarchy, and finally `Workflow`, `Edge`, and `Comment`.

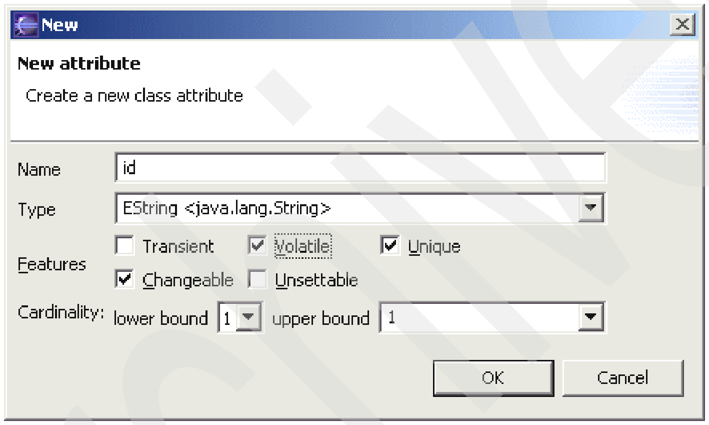
To do the WorkflowElement interface creation:

1. Open the EMF class diagram editor:
   a. Select `Workflow.ecd` in the `WorkflowModel` project in the navigator view of Eclipse.
   b. Right-click **Open with -> EMF Class Diagram Editor** — or simply double-clicking the file tree item should work fine also.
2. Create a class in the editor:
   a. Click the icon for the class creation tool on the editor tool palette, click in the working area of the editor, and a new a window opens.
   b. Enter a name, for example, `WorkflowElement`, and choose the boxes, **Is an interface** and **Is abstract**, then click **Finish**.

Attribute creation

Now we add an id attribute to the WorkflowElement interface:

1. Select the `WorkflowElement`, by clicking close to the border of the visual in the editor. A rectangle should appear; right-click and choose **New -> Attribute**.
   a. Enter the name of the attribute, for example, `id`.
   b. Select the type of the attribute, for example, `EString`. Most of the EMF types, which are equivalent to the Java basic types, are available.
   c. Choose the features you want to give to the attribute. See Figure 1-7 for an example and refer to 1.3.6, "EMF features" on page 24 for more information on the features themselves.
   d. Choose the cardinality associated to the attribute, and click **OK**.



Name: id
Type: EString <java.lang.String>
Features: [ ] Transient [✓] Volatile [✓] Unique [✓] Changeable [ ] Unsettable
Cardinality: lower bound 1 upper bound 1

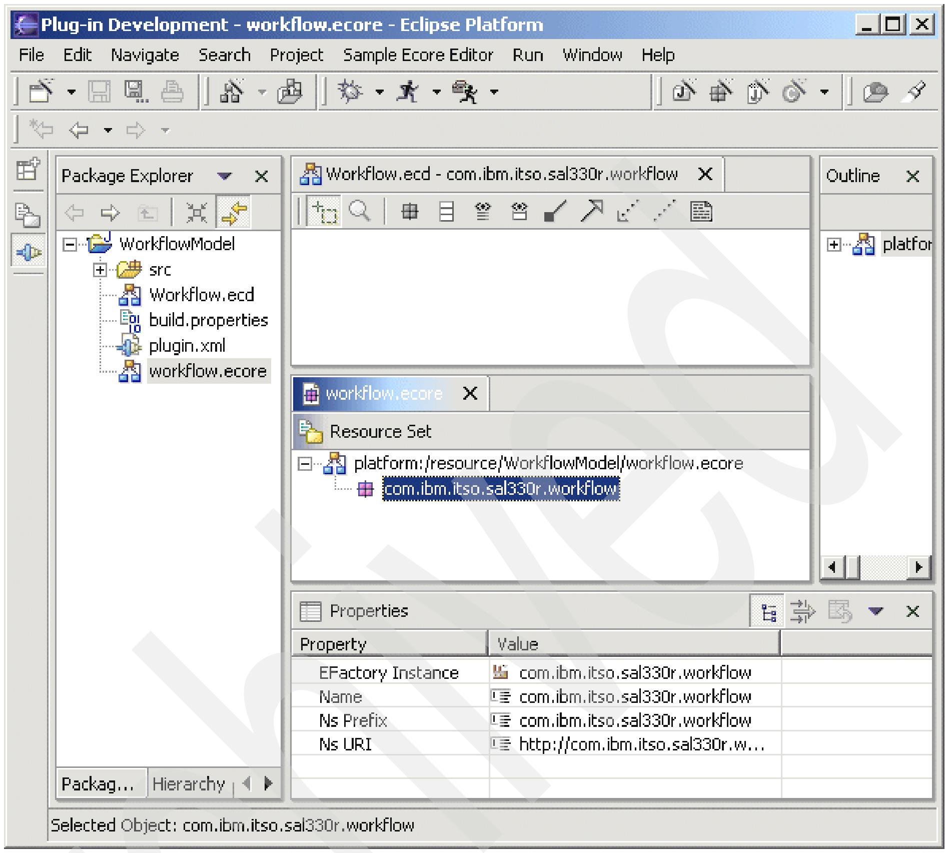
Figure 1-7 The new attribute window

At any point, if you realize that something is wrong or that you have forgotten something, do not worry; most of the time, you do not have to delete your model and start again. Simple corrections can be made in the property view, and more complex corrections can be made using either the Sample Ecore Model editor or the default text editor. These give you different ways of accessing the underlying model, and allow you to correct, enhance, or even totally redefine the model.

Available editors

To open the Sample Ecore Model or the text editor:

- Select the `Workflow.ecore` file, right-click and either choose **Open With -> Sample Ecore Model Editor** or **Open With -> Text Editor**. See Figure 1-8 for an example of using the Ecore Model Editor.


Plug-in Development - workflow.ecore - Eclipse Platform
File Edit Navigate Search Project Sample Ecore Editor Run Window Help
Package Explorer
WorkflowModel
src
Workflow.ecd
build.properties
plugin.xml
workflow.ecore
Packag... Hierarchy
Workflow.ecd - com.ibm.itso.sal330r.workflow
Outline
platfor
workflow.ecore
Resource Set
platform:/resource/WorkflowModel/workflow.ecore
com.ibm.itso.sal330r.workflow
Properties

Property	Value
EFactory Instance	com.ibm.itso.sal330r.workflow
Name	com.ibm.itso.sal330r.workflow
Ns Prefix	com.ibm.itso.sal330r.workflow
Ns URI	http://com.ibm.itso.sal330r.w...

Selected Object: com.ibm.itso.sal330r.workflow

Figure 1-8 EMF Class Diagram and Sample Ecore Model Editor together

Note: The `Workflow.ecd` file cannot be open in the EMF Class Diagram editor and the text editor at the same time. To chose the editor to open the file in, select the file, right-click **Open With -> EMF Class Diagram Editor**, or **Open With -> Text Editor.**

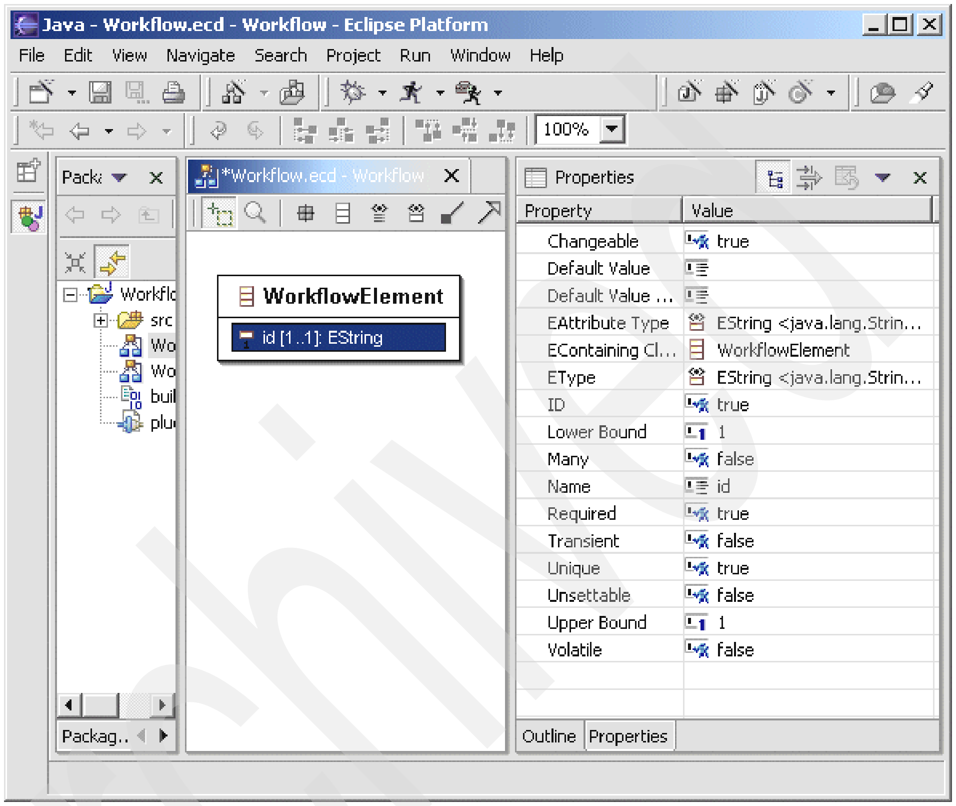
The property view

Some properties are not directly supported by the EclipseUML plug-in, but they can still be changed using the property view.

To show the property view in Eclipse:

1. Click **Window -> Show View -> Other...**
2. Select **Basic -> Properties,** and click **OK**.

We use the property view to complete the `id` attribute. We have to mention that the id is an ID, which will be used for serialization later. We set the ID property of the `id` attribute to `true` as shown in Figure 1-9.



Property	Value
Changeable	true
Default Value	
Default Value ...	
EAttribute Type	EString <java.lang.Strin...
EContaining Cl...	WorkflowElement
EType	EString <java.lang.Strin...
ID	true
Lower Bound	1
Many	false
Name	id
Required	true
Transient	false
Unique	true
Unsettable	false
Upper Bound	1
Volatile	false

Figure 1-9 The property view with the ID attribute set to true

We complete the `WorkflowElement` interface by adding all the other attributes. Each WorkflowElement in a workflow has a `name`, is located at position `x` and `y` on the canvas, and has a `height` and a `width`. Table 1-1 shows the properties of all the WorkflowElement attributes.

Table 1-1 WorkflowElement attribute properties

Name	Type	Features and properties

id	EString	volatile="true"
		lowerBound="1"
		iD="true"

name	EString	
comment	EString	
x	EInt	defaultValueLiteral="0"
y	EInt	defaultValueLiteral="0"
width	EInt	defaultValueLiteral="-1"
height	EInt	defaultValueLiteral="-1"

We repeat the same process to create the other classes of the Workflow model. See Table 1-2 for a summary of their attributes, features, and properties. For the classes which are not in the table, but are in the model, depicted in Figure 1-4 on page 9, simply create them with no attribute. Do not forget that WorkflowElement and WorkflowNode are two abstract classes.

Note: The Default Value Literal can only be set in the property editor.

Table 1-2 Interface attributes properties

Interface Name/attribute	Type	Features and properties
WorkflowNode (abstract)		

isStart	EBoolean	defaultValueLiteral="false"
		lowerBound="1"

isFinish	EBoolean	defaultValueLiteral="false"
		lowerBound="1"

Transformation		
transformExpression	EString	
LoopTask		
whileCondition	EString	lowerBound="1"
ConditionalOutputPort		
condition	EString	lowerBound="1"

Figure 1-10 shows what the model should like after these steps.

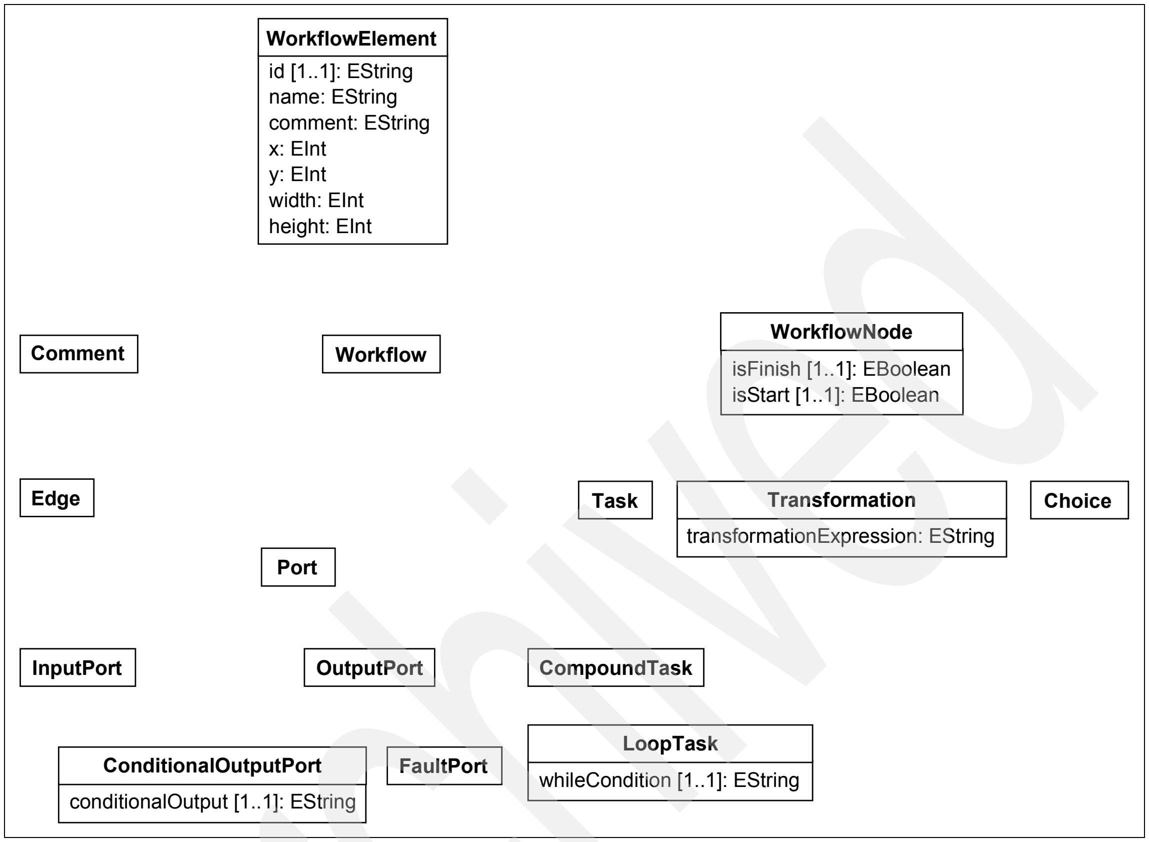


Figure 1-10 The workflow model classes, before relationship definition

Generalization definition

Generalization or inheritance links are made using the generalization tool. Select the tool by clicking its icon, which is an arrow with a big triangle at the end. Click the specialized interface, hold the mouse button down, then move to the generalized interface or connect to a generalization link going to the superclass. Figure 1-11 shows our model with the generalization relationships added.

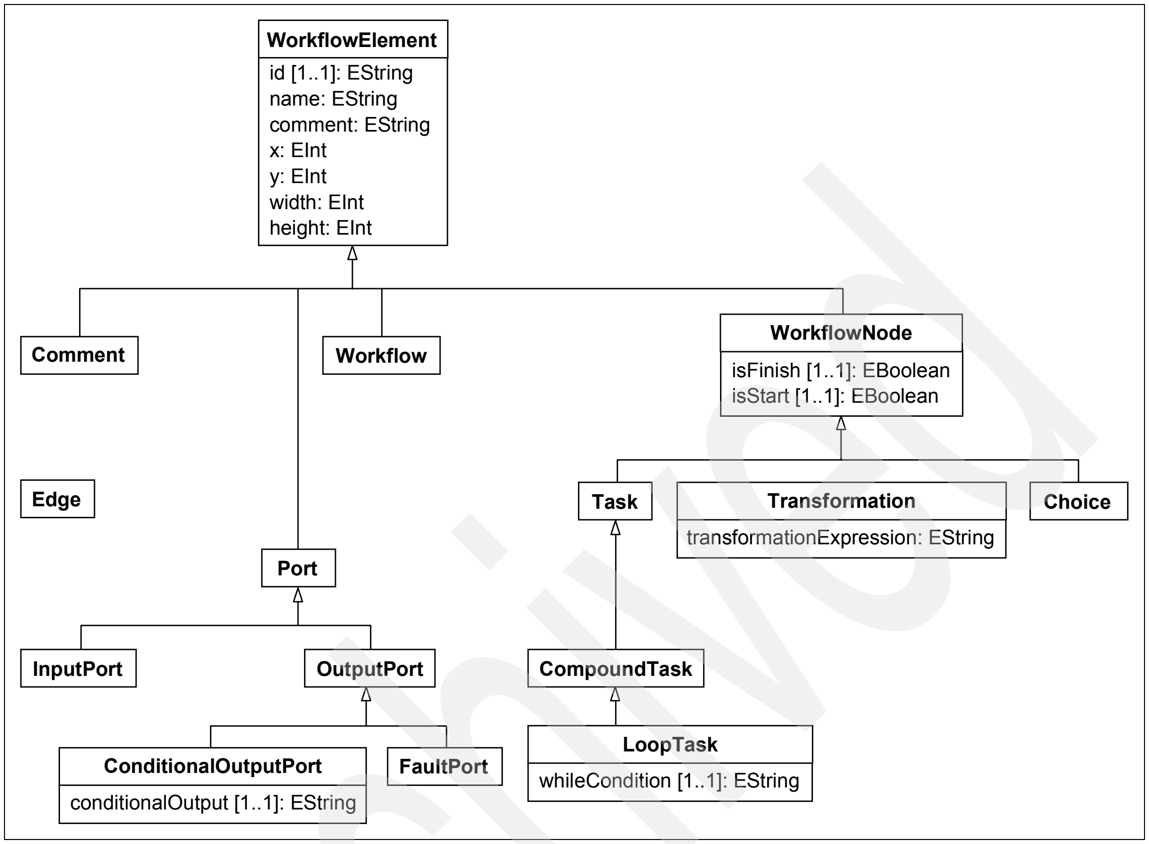


Figure 1-11 Generalization relationships

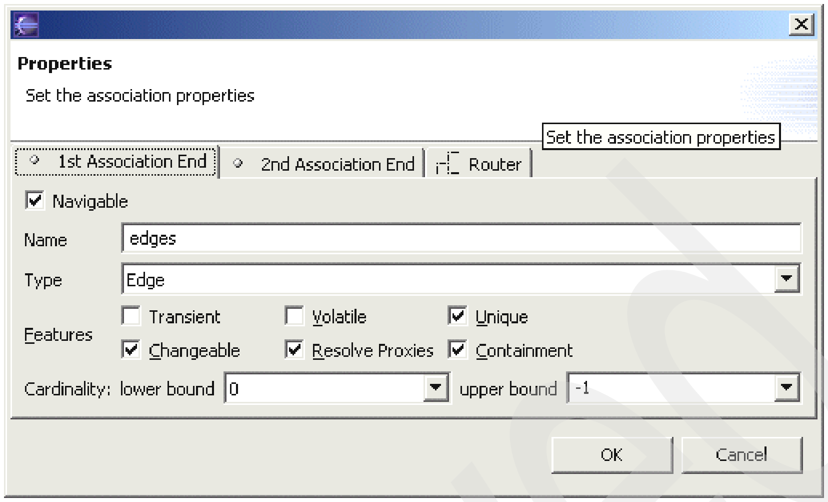
Association definition

Using the association tool, we set up the associations between the classes. We show how to set up the association between `Workflow` and `Edge`, then we provide a summary of all the other associations with their features; see Table 1-3.

Here are the steps to set up the `Workflow` to `Edge` association:

1. Click the source interface, which is `Workflow`.
2. Click the target interface, which is `Edge`.
3. Give the association properties, see Figure 1-12.

   Add `'s'` to the association name, click **Containment**, select `-1` as the upper bound cardinality, and click **OK.** See Figure 1-12.


Properties

Set the association properties

Set the association properties

☑ Navigable

Name: edges

Type: Edge

Features: ☐ Transient ☐ Volatile ☑ Unique ☑ Changeable ☑ Resolve Proxies ☑ Containment

Cardinality: lower bound 0 upper bound -1

Figure 1-12   Association property window

Each association has two endpoints. So far, we have defined the characteristics of the Workflow to Edge association, now we complete the opposite association end, which is called `Workflow`.

We complete the second association endpoint, by:

1. Clicking the **2nd Association End** tab.
2. Changing the **lower bound** cardinality to be `1`, then clicking **OK**.

We do the same for all the associations in the model. Any mistake can be corrected later, by simple double-clicking the link itself in the editor. Table 1-3 shows the associations that you should create.

Table 1-3   Association properties

Origin	End	Association end	Attributes
Workflow	Edge	edges	upperBound="-1" containment="true"
		workflow	lowerBound="1"
	WorkflowNode	nodes	upperBound="-1" containment="true"
		workflow	lowerBound="1"
	Comment	comments	upperBound="-1" containment="true"

Origin	End	Association end	Attributes
		workflow	lowerBound="1"
WorkflowNode	InputPort	inputs	lowerBound="1" upperBound="-1" containment="true"
		node	lowerBound="1"
	OutputPort	outputs	lowerBound="1" upperBound="-1" containment="true"
		node	lowerBound="1"
OutputPort	Edge	edges	upperBound="-1"
		source	lowerBound="1"
InputPort	Edge	edges	upperBound="-1"
		target	lowerBound="1"
Compound task	Workflow	subworkflow	lowerBound="1" containment="true"

Note: Take care, that:

1. The link between `CompoundTask` and `Worklow` is only a one-way link, navigable from `CompoundTask` to `Workflow`. Open the **2nd association end** of the link and unset **Navigable**.
2. The `Edge` to `OutputPort` association name is called `source` and the one from `Edge` to `InputPort` is named `target`, because an `Edge` connects an `OutputPort` to the next `InputPort`.

The model will now be like that shown in Figure 1-13.

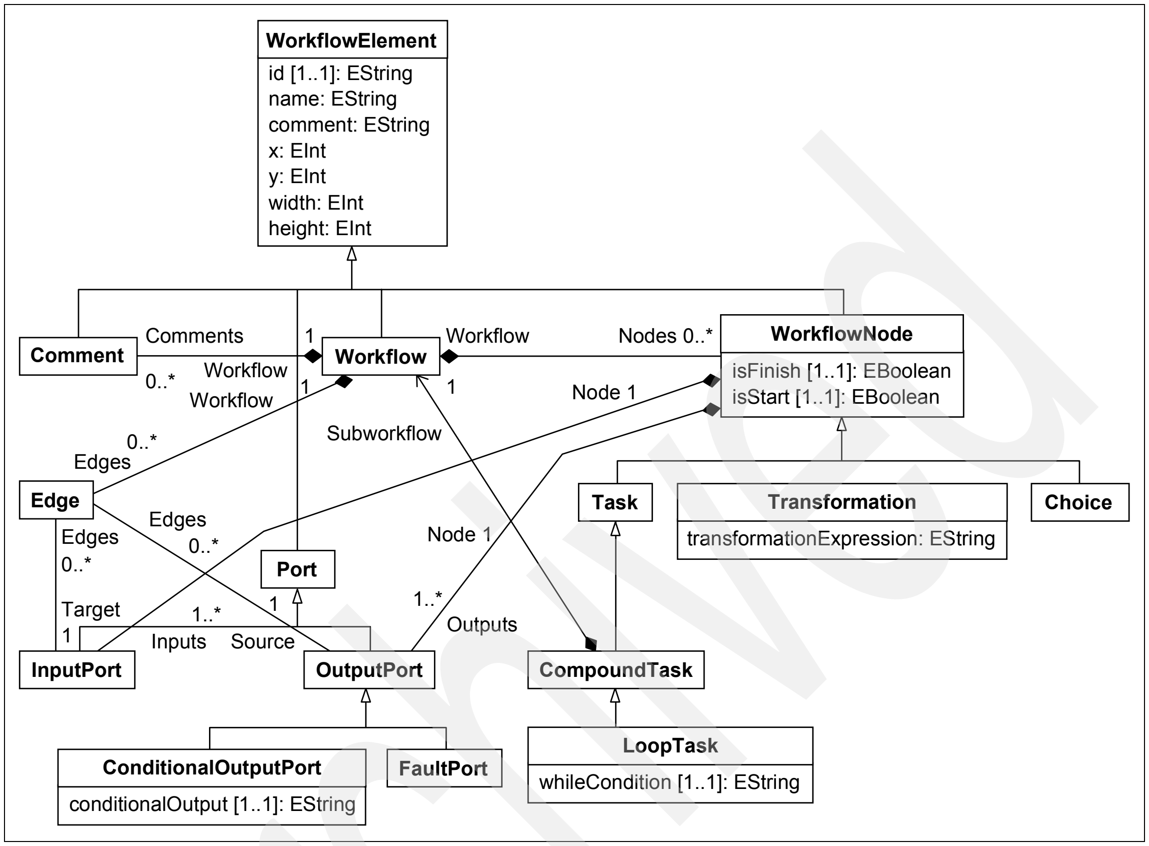


Figure 1-13 Workflow complete model

1.3.5 Modeling using Java interface annotation

To define a model by means of Java interface annotations, we need to provide the same set of information we gave during the graphical modeling. We need to create a set of interfaces, one for each of the model elements. Each interface contains methods. The annotation mechanism enhances the code by adding some @model tags in the comment of any code element.

Interface definition

The `abstract="true"` attribute is used for `WorkflowElement` and `WorkflowNode`. Example 1-1 shows the @model tag for the `WorkflowNode`. All the other interfaces use the standard @model tag to enhance the model.

Example 1-1 The WorkflowNode interface@model tag

```
package com.ibm.itso.sal330r.workflow;

import org.eclipse.emf.ecore.EObject;

/**
 * @author Vanderheyden
 *
 * @model abstract="true"
 */

public interface WorkflowElement extends EObject{

}
```

Adding attributes

An attribute is not added directly to the interface, instead, we have to define an accessor for it. Code generation completes the interface by defining the setter and provides the implementation of both the setter and the getter. Example 1-2 shows the x attribute @model tag.

Example 1-2 The x attribute @model tag

```
   /**
    * @model default="0"
    */
   int getX();
```

Adding associations

For each reference, we have to define:

- The type of object it gives access to.
- If it is a containment reference.
- The name of the second association endpoint.
- If it is required or not.

  See Example 1-3

Example 1-3 The WorkflowNode to OutputPort reference @model tag

```
package com.ibm.itso.sal330r.workflow;

import org.eclipse.emf.common.util.EList;

/**
 * @model abstract="true"
 */
public interface WorkflowNode extends WorkflowElement{

   /**
    * @model type="com.ibm.itso.sal330r.workflow.OutputPort" opposite="node"
containment="true" required="true"
    */
   EList getOutputs();

}
```

Note: The complete rebuild of the model using the Java annotation mechanism is a very long process, and there is no real added value in providing complete instructions in the context of our redbook.

Here is a short summary of what has to be done, for those who want to do it:

1. Create an EMF project.
2. Create a Java package.
3. Create a Java interface for all the model objects.
4. Add a getter method for each attribute.
5. Add a method for each association which is navigable. Two methods are added for navigation navigable from both ends.
6. Create an EMF model inside the EMF project, by using the Java annotation mechanism.

Java annotation and the code generation process

Each @model tag annotates the Java code to provide model related information. Those directive are used by the code generator in order to generate the corresponding implementation code. The code generation process is a non-destructive process. No @model annotations are lost during code generation. Generated code will contain the @generated tag to indicate that it has been generated and can be replaced again.

1.3.6 EMF features

EMF features are associated with attributes and associations. The code generator uses them to generate the implementation code.

EMF features for an attribute

Table 1-4 provides a short description of the EMF features that can be associated with an attribute.

Table 1-4 EMF features for an attribute

EMF feature	Description
Transient	Transient is the opposite to persistent. The attribute value is not supposed to be saved, persisted.
Volatile	A cache behavior is implemented for attribute value. Volatile is a way to prevent caching.
Unique	If the attribute is multi valued (upperBound=”-1”), each value must be unique in that case
Changeable	Indicates if an attribute can be modified.
Unsettable	Indicates if an attribute can be set in a state that mean it has no value.

EMF features for an association

Table 1-5 provides a short description of the EMF features that can be associated with an association.

Table 1-5 EMF features for an association

EMF feature	Description
Transient	The object referenced through this association will not get persisted.
Volatile	Prevents the object caching.
Unique	All referenced objects are unique.
Changeable	If true, the value of the attribute is not hard coded, fixed.
Resolve Proxies	Indicates whether proxy reference should be resolved automatically.
Containment	If true, it means that any object, called the containment, which is referenced by this one, called the container, are considered as being part of it.

1.3.7 EMF model creation

Once the model has been completed, by means of EMF modeling or Java interface definition, we can generate the corresponding code to implement it. We need to create a new generator model resource, which is based on our Ecore file, or our Java interfaces.

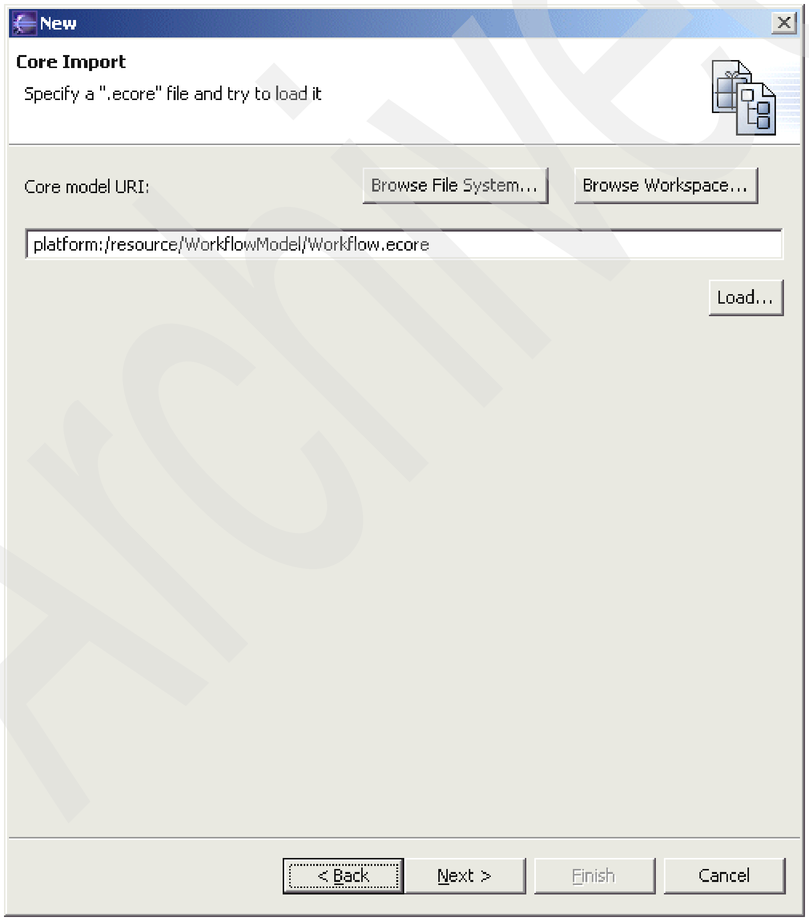
These are the steps to create an EMF model from an EMF class diagram:

1. Create the model:

   Click **File -> New -> Other...**, select **Eclipse Modeling Framework -> EMF Models**, click **Next**.

2. Choose the parent folder, for example, `WorkflowModel` project, define the EMF model file name with a `genmodel` extension, for example, `Workflow.genmodel`, and click **Next**.
3. Select **Load from an EMF core model**, and click **Next**.
4. Choose the `.ecore` file for which you want to create a model.

   Click **Browse Workspace...,** navigate to the `WorkflowModel` project, select `Workflow.ecore` file, and click **Next.** See Figure 1-14.



platform:/resource/WorkflowModel/Workflow.ecore

Figure 1-14 Ecore file selection window

5. Choose `Workflow` package selection, and click **Finish**.

These are the steps to create an EMF model from Java interface annotations:

1. Create an EMF Model:

   Click **File -> New -> Other...**, choose **Eclipse Modeling Framework** and **EMF Models** and click **Next**.

2. Choose the project and the package you want to contain the generator model resource. Define a file name for the model, for example, `Workflow.genmodel`, and click **Next**.
3. Select load from annotated Java and click **Next**.
4. Choose the package selection, and click **Finish**.

The `workflow.ecore` and `Workflow.genmodel` files have been created.

1.3.8 Code generation facility

Once the Workflow.genmodel has been created and opened in an EMF Generator editor by Eclipse, the code generation can take place:

1. Open the EMF Generator Editor:

   Select the `Workflow.genmodel` file, right-click **Open With -> EMF Generator**.

2. Generate the code:

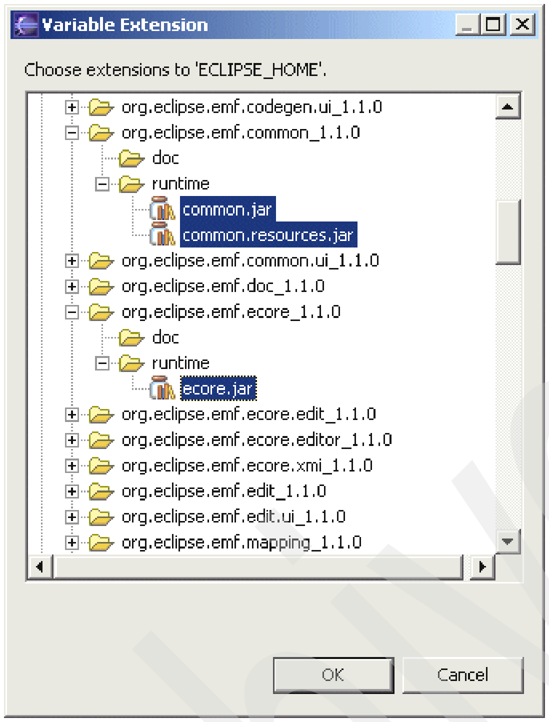
   In the editor, click **Generator -> Generate Model Code** or select the root element in the tree and right-click **Generate Model Code**.

1.3.9 Compiling the code

Before compiling, the Java build path has to be updated, in order to resolve the EMF classes.

To update the Java Build Path:

1. Open project properties:
   a. Select the WorkflowModel project, right-click **Properties,** select **Java Build Path**.
2. Open the **Libraries** tab:
   a. Click **Add Variable**.
   b. Select **ECLIPSE_HOME - C:\Program Files\eclipse**.
   c. Click **Extend...**, select `ecore.jar`, `common.jar` and `common.resource.jar`, and click **OK.** See Figure 1-15.


Variable Extension

Choose extensions to 'ECLIPSE_HOME'.

```
org.eclipse.emf.codegen.ui_1.1.0
org.eclipse.emf.common_1.1.0
    doc
    runtime
        common.jar
        common.resources.jar
org.eclipse.emf.common.ui_1.1.0
org.eclipse.emf.doc_1.1.0
org.eclipse.emf.ecore_1.1.0
    doc
    runtime
        ecore.jar
org.eclipse.emf.ecore.edit_1.1.0
org.eclipse.emf.ecore.editor_1.1.0
org.eclipse.emf.ecore.xmi_1.1.0
org.eclipse.emf.edit_1.1.0
org.eclipse.emf.edit.ui_1.1.0
org.eclipse.emf.mapping_1.1.0
```

Figure 1-15 EMF jar files

3. Click the **Order and Export** tab
   a. Select the three jars, click **Up** to move them to the correct position in the path, click **OK**.
4. Compile the code, select **Project -> Rebuild All**.

1.3.10 Conclusion

We have demonstrated how to create an EMF model, which can be used directly as the model for our application. For more information on the Object, View, and Interaction Diagram (OVID) vocabulary used, see:

`http://www-3.ibm.com/ibm/easy/eou_ext.nsf/Publish/589`

Accordingly, our model contains all the model objects that we need and all the object relationships and navigation paths to easily move from one object to the next. The model needs to be enhanced with some convenience methods, for example, the `connectTo()` method in the `Workflow` object, that will even encapsulate more of the model specifics and give a higher level model entry point.

2

EMF examples

In this chapter we discuss Eclipse Modeling Framework (EMF) modeling techniques and provide examples of creating models with EMF. We also cover the EMF.Edit framework and provide tips and techniques for generating and customizing EMF-based editors. Finally, we outline how to use Java Emitter Templates (JET) to customize code generation from EMF models.

Note: The sample code we describe in this chapter is available as part of the redbook additional material. See Appendix A, “Additional material” on page 225 for details on how to obtain and work with the additional material. The sample code for this chapter is provided as Eclipse projects that can be imported into your Eclipse workbench. Each major section of this chapter has a matching Eclipse project in the additional material. The projects are cumulative and they also depend on your having completed the modelling and code generation described in Chapter 1, “Introduction to EMF” on page 3. You will need to make sure that you have created the Java build path variables described in 1.3.9, “Compiling the code” on page 27, otherwise you may get classpath errors when importing the sample projects.

2.1 EMF modeling techniques

In this section, we focus on techniques for modeling with EMF. We begin by exploring examples to illustrate how to define new models using EMF. Then, we discuss the mapping between EMF and XML Schema and describe how a model expressed in XML Schema is migrated to EMF.

2.1.1 Creating new models

In this section we illustrate how to use EMF's Ecore model concepts to create new models. We begin by creating a naive model of Workflow, and then refactor that model based on modeling tips that we provide. We discuss the motivation for each change to the model and describe how to generalize the refactorization to other models.

Note: For a handy overview of the Ecore model concepts, consult the JavaDoc for the org.eclipse.emf.ecore package. Aside from the APIs for each model object, you will also find a class diagram of the Ecore model as well as a list of the EMF Datatypes and their corresponding Java types.

Creating a simple Workflow model

The model that we create in this section is a simplified version of the WorkflowModel used in the sample application and described in Chapter 6, "Sample requirements and design" on page 187. For our example, we only concern ourselves with modeling basic tasks and dataflow between those tasks. Figure 2-1 shows a model that we might create to describe this domain.

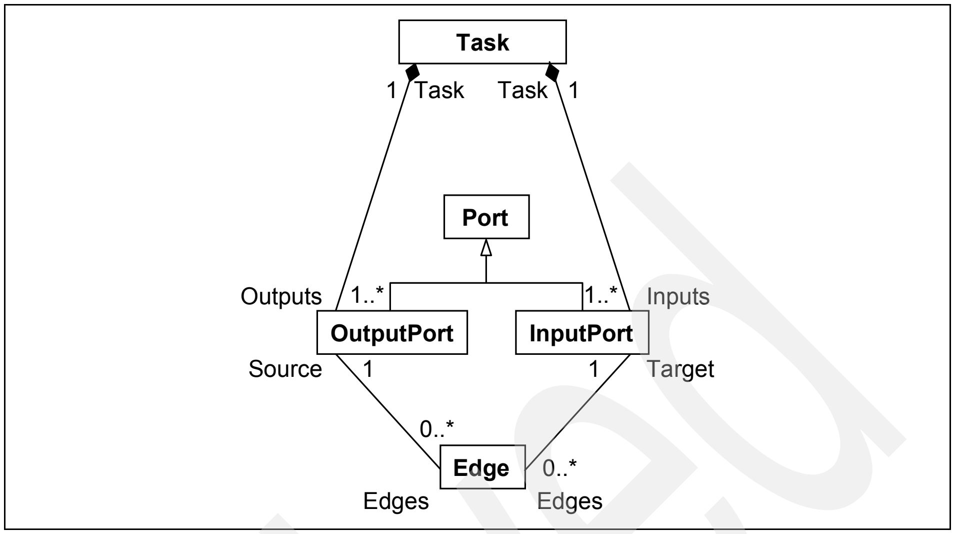


Figure 2-1 Native model of Workflow

In our model, Tasks represent units of work, and Edges represent the connections (flows of control and data) between them. Each Edge flows from an OutputPort on a Task to an InputPort on another Task, indicating that data resulting from the completion of the source's Task becomes the input of the target's Task. We have used the multiplicity of the references from Task to InputPort and OutputPort, to express the constraint that each Task must have at least one InputPort and at least one OutputPort.

We construct our model as described in Chapter 1, "Introduction to EMF" on page 3. We use the Sample Ecore Model Editor, but you may choose to edit the XMI directly, or use the Omondo EclipseUML plug-in. We create an EPackage named workflow, and within it, create EClasses to represent Task, Edge, Port, OutputPort, and InputPort.

Tip: If you are using the model to drive code generation, we suggest that you follow Java conventions for naming model elements:

- Heed Java case conventions:
  - Use lower case for package names.
  - Use lower case for the initial letter of feature and operation names.
  - Begin class names with an upper case letter.
- Use the plural form for names of multi-valued features and the singular form for single-valued features.

Example 2-1 shows the XML Metadata Interchange (XMI) that represents the workflow EPackage. Each EClass is represented as an `eClassifiers` element nested within the workflow `EPackage` element.

Example 2-1 XMI for model of Workflow

```
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" name="workflow"
   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" nsPrefix="workflow"
   nsURI="http://www.redbooks.ibm.com/sal330r/example/workflow1">
   <eClassifiers xsi:type="ecore:EClass" name="Task">
      <eReferences name="inputs" eType="#//InputPort" lowerBound="1"
         upperBound="-1" containment="true" eOpposite="#//InputPort/task"/>
      <eReferences name="outputs" eType="#//OutputPort" lowerBound="1"
         upperBound="-1" containment="true" eOpposite="#//OutputPort/task"/>
   </eClassifiers>
   <eClassifiers xsi:type="ecore:EClass" name="Port" abstract="true"/>
   <eClassifiers xsi:type="ecore:EClass" name="Edge">
      <eReferences name="target" eType="#//InputPort" lowerBound="1"
         eOpposite="#//InputPort/edges"/>
      <eReferences name="source" eType="#//OutputPort" lowerBound="1"
         eOpposite="#//OutputPort/edges"/>
   </eClassifiers>
   <eClassifiers xsi:type="ecore:EClass" name="InputPort"
      eSuperTypes="#//Port">
      <eReferences name="edges" eType="#//Edge" upperBound="-1"
         eOpposite="#//Edge/target"/>
      <eReferences name="task" eType="#//Task" lowerBound="1"
         eOpposite="#//Task/inputs"/>
   </eClassifiers>
   <eClassifiers xsi:type="ecore:EClass" name="OutputPort"
      eSuperTypes="#//Port">
      <eReferences name="edges" eType="#//Edge" upperBound="-1"
         eOpposite="#//Edge/source"/>
      <eReferences name="task" eType="#//Task" lowerBound="1"
         eOpposite="#//Task/outputs"/>
   </eClassifiers>
</ecore:EPackage>
```

Although we have shown associations in the Class Diagram in Figure 2-1, the Ecore model does not represent associations explicitly. Instead, we use an EReference to represent each navigable end of an association. An association that is navigable in both directions is represented by two EReferences, one on each associated class, with eOpposites that refer to each other. For example, the association between Edge and InputPort is navigable from both ends, and so we see the edges EReference in InputPort and the target EReference in Edge.
It is important to make sure that the eOpposites of a pair of corresponding EReferences match, and that both EReferences have their eOpposite set.

An association that is navigable in one direction only is represented as a single EReference, with no eOpposite. The multiplicity of the association ends is represented by the `upperBound` and `lowerBound` attributes on the eReferences elements representing each EReference.

As we can see from our example, associations that represent containment, such as the associations between Task and Ports, are represented by an EReference where containment is true, on the containing class. The containment of the InputPorts and OutputPorts within Tasks is represented by the `inputs` and `outputs eReferences` inside the Task `eClassifiers` element.

The inheritance of ports is represented by the `eSuperTypes` attribute on the InputPort and OutputPort elements. The EClass Port is an abstract class, which is indicated by the value of the `abstract` attribute on the `eClassifiers` element representing Port.

When we generate an EMF.Edit-based editor from our model, as described in the EMF documentation, and use it to create Tasks and Edges, we can immediately see a problem with this model. Using the generated editor, we can only create Tasks and Edges separately; we are missing a class that we could instantiate to contain all of the tasks and edges in our workflow. The solution is to add an additional class, Workflow, that contains both Tasks and Edges.

Tip: It is often useful to design models around a containment-based hierarchy rooted at a single class. This approach can make it easier to work with instances, as you have a single entry point from which you can access all of the other objects in the instance (directly or indirectly), and it means that all of the objects will be serialized into a single XMI document by default. We discuss this in more detail in 2.3.2, “Default serialization of model instances” on page 66.

If you wish to have the flexibility of choosing whether or not to contain instance objects in the top-level container, make sure that any references back to the container have a lowerBound of zero.

Figure 2-2 shows the model with the additional Workflow class.

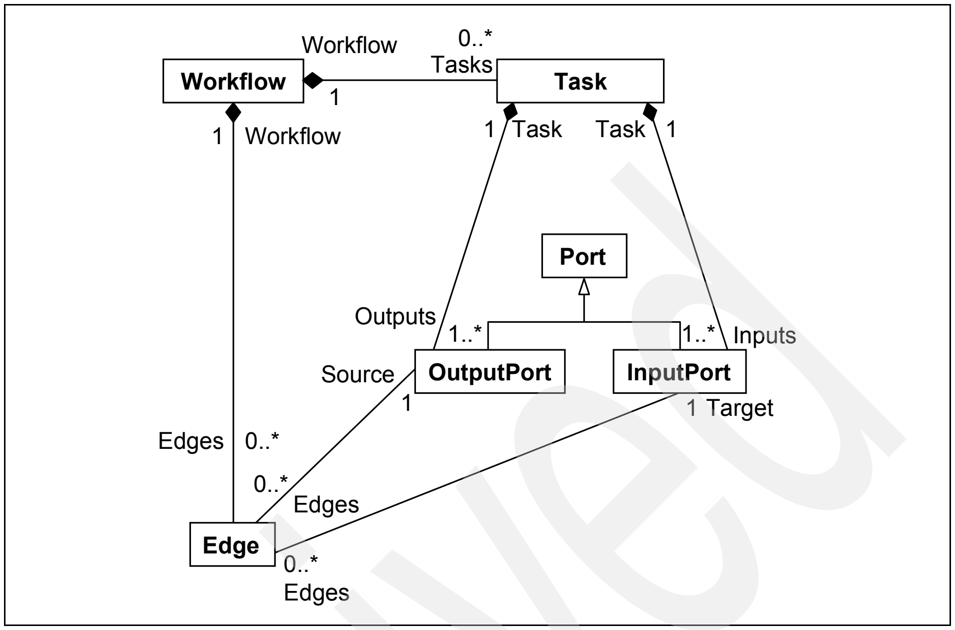


Figure 2-2 Model of Workflow with additional Workflow class

Example 2-2 shows the XMI fragment that represents the Workflow class. The `eClassifiers` element is added to the contents of the workflow EPackage. References to the Workflow are also added to Task and Edge as `eReferences` elements within the `eClassifiers` representing each class, for example:
`<eReferences name="workflow" eType="#//Workflow" lowerBound="1" eOpposite="#//Workflow/edges"/>`

Example 2-2 XMI fragment for Workflow class

```
<eClassifiers xsi:type="ecore:EClass" name="Workflow">
   <eReferences name="tasks" eType="#//Task" upperBound="-1"
      containment="true" eOpposite="#//Task/workflow"/>
   <eReferences name="edges" eType="#//Edge" upperBound="-1"
      containment="true" eOpposite="#//Edge/workflow"/>
</eClassifiers>
```

When we start adding detail to the classes that we use to model workflow, we notice that many of the elements share common features, such as name. This is often the case when modelling, and it is usual to create a common supertype that represents an abstraction of all objects in the model, and which provides these common features. When you are using such a model, you have the benefit of knowing that all objects in the model are of that type, which can be useful when you are working with the objects reflectively. For EMF models, this is less of an issue, as all model elements already have a common supertype, EObject, and a rich reflective API is provided to allow you to work with your model objects in this way.

Figure 2-3 shows the model with the added WorkflowElement class.

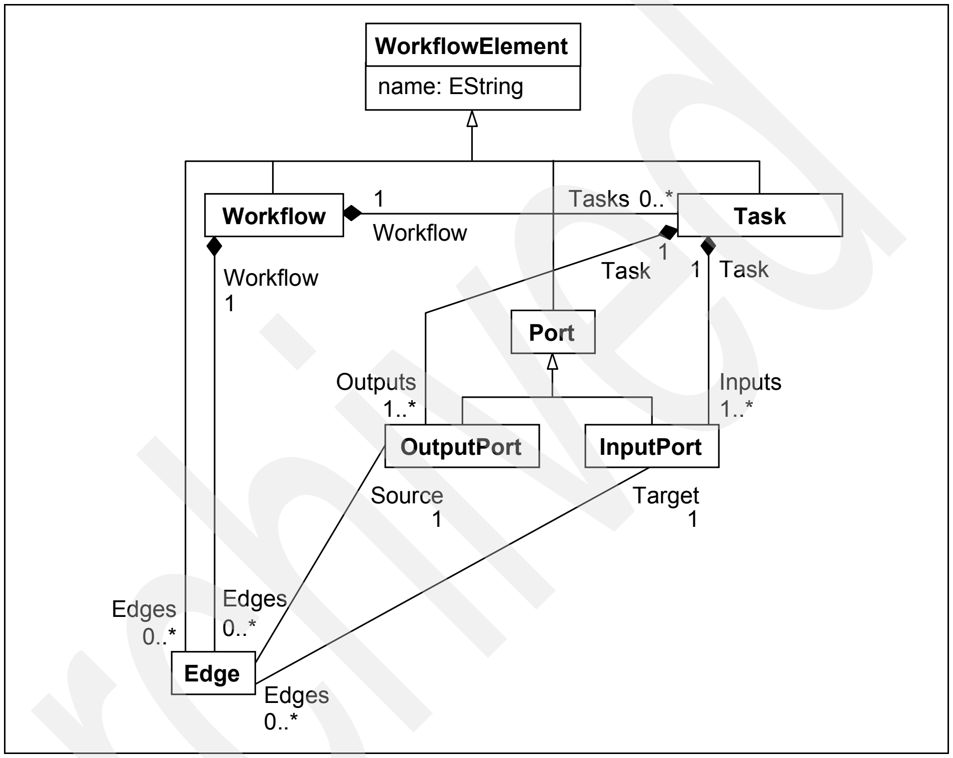


Figure 2-3 Model of Workflow with additional common supertype

Working with packages

EPackages are used to collect EClasses and EDataTypes together in much the same way that packages are used in Java. In this section, we discuss models that span multiple packages.

Typically packages are used to group related concepts into reusable modules. When creating an editor for a model, it is often necessary to store additional information about model objects, such as layout information or display properties. For the sample application described in Chapter 7, “Implementing the sample” on page 203, we add this information directly to the WorkflowModel; however, another approach is to use a separate package to represent the information about each diagram.

We create an EPackage Diagram, and within it, classes to represent connected and contained nodes within that diagram, as Figure 2-4 shows. Display properties such as the x and y co-ordinates, width and height, are represented by EAttributes belonging to DiagramNode.

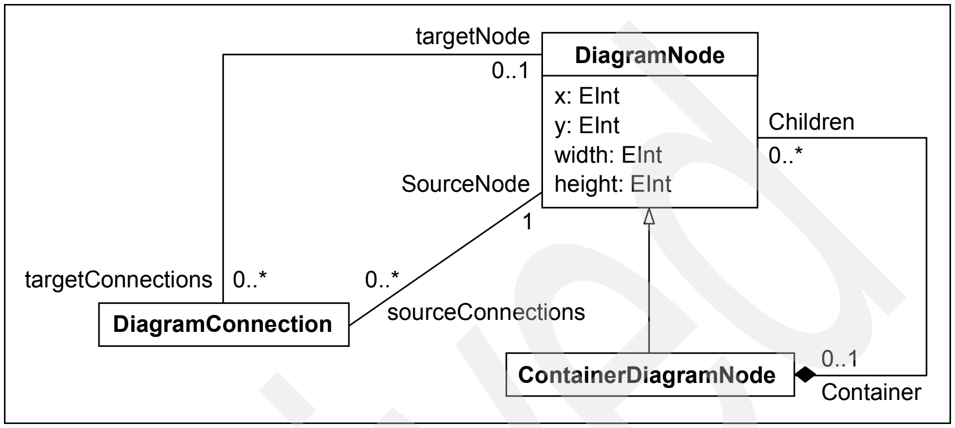


Figure 2-4 DiagramModel

The following examples illustrate two ways of using our DiagramModel and WorkflowModel together:

- ► We construct a new package WorkflowDiagram, which merges concepts from the two packages using inheritance.
- ► We store the diagram information separately from the workflows, using references between DiagramNode and DiagramConnection and the appropriate classes from the WorkflowModel to maintain the relationship between the two models.

For the first approach, we create a new package WorkflowDiagramPackage, which contains classes that combine concepts from the WorkflowModel and the DiagramModel. For example, a Task in a Diagram is represented by a WorkflowDiagramTask, which inherits from both Task and DiagramNode. Notice that we identify types defined in another package by the Ecore file that contains the type, followed by the usual reference to the type itself. Also notice that the multiple inheritance is represented by a space separated list within the eSuperTypes attribute. We choose to specify the corresponding classes from the WorkflowModel as the primary supertypes of the classes in the WorkflowDiagram model, and so they appear in the eSuperTypes list first.

Example 2-3 Importing the DiagramModel and WorkflowModel

```
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" nsPrefix="wfDiagram"
   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="WorkflowDiagram"
   nsURI="http://www.redbooks.ibm.com/sal330r/example/workflowdiagram">
   <eClassifiers xsi:type="ecore:EClass" name="WorkflowDiagramTask"
      eSuperTypes="workflowWithSupertype.ecore#//Task
         Diagram.ecore#//DiagramNode"/>
   <eClassifiers xsi:type="ecore:EClass" name="WorkflowDiagramWorkflow"
      eSuperTypes="workflowWithSupertype.ecore#//Workflow
         Diagram.ecore#//DiagramNode"/>
   <eClassifiers xsi:type="ecore:EClass" name="WorkflowDiagramEdge"
      eSuperTypes="workflowWithSupertype.ecore#//Edge
         Diagram.ecore#//DiagramNode"/>
</ecore:EPackage>
```

In the second approach, the diagram and the workflow are more loosely coupled. We add references to the classes in the DiagramModel to represent the linkage between the two models, as shown in Example 2-4.

Example 2-4 DiagramModel with references to WorkflowModel objects

```
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" nsPrefix="diagram"
   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="diagramwithrefs"
   nsURI="http://www.redbooks.ibm.com/sal330r/example/diagram">
   <eClassifiers xsi:type="ecore:EClass" name="DiagramNode">
      <eReferences name="container" eType="#//ContainerDiagramNode"
         eOpposite="#//ContainerDiagramNode/children"/>
      <eReferences name="model"
         eType="ecore:EClass WorkflowWithCommonSupertype.ecore#//Task"/>
      <eReferences name="sourceConnections" eType="#//DiagramConnection"
         upperBound="-1" eOpposite="#//DiagramConnection/sourceNode"/>
      <eReferences name="targetConnections" eType="#//DiagramConnection"
         upperBound="-1" eOpposite="#//DiagramConnection/targetNode"/>
      <eAttributes name="x"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
      <eAttributes name="y"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
      <eAttributes name="width"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
      <eAttributes name="height"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
   </eClassifiers>
   <eClassifiers xsi:type="ecore:EClass" name="DiagramConnection">
      <eReferences name="sourceNode" eType="#//DiagramNode" lowerBound="1"
         eOpposite="#//DiagramNode/sourceConnections"/>
      <eReferences name="targetNode" eType="#//DiagramNode"
         eOpposite="#//DiagramNode/targetConnections"/>
      <eReferences name="model"
      eType="ecore:EClass WorkflowWithCommonSupertype.ecore#//Edge"/>
   </eClassifiers>
   <eClassifiers xsi:type="ecore:EClass" name="ContainerDiagramNode"
      eSuperTypes="#//DiagramNode">
      <eReferences name="children" eType="#//DiagramNode" upperBound="-1"
         containment="true" eOpposite="#//DiagramNode/container"/>
      <eReferences name="model"
      eType="ecore:EClass WorkflowWithCommonSupertype.ecore#//Workflow"/>
   </eClassifiers>
</ecore:EPackage>
```

Notice that because we are referencing classes from another package, we have to be explicit about the type. For example, we refer to the Task class as follows:

```
   <eReferences name="model"
   eType="ecore:EClass WorkflowWithCommonSupertype.ecore#//Task"/>
```

Notice also that the references are one-way references, as we do not wish to pollute the WorkflowModel with references to the DiagramModel.

The Ecore model also allows us to define nested packages, which are represented in the XMI as eSubpackages elements. For example, we could package the DiagramModel and the WorkflowModel together as sub-packages of a new package NestedWorkflowDiagram. Example 2-5 shows the XMI for our NestedWorkflowDiagram package, with some details omitted for brevity. Notice that reference strings also now include the subpackage, such as:

```
   <eReferences name="model" eType="#//workflowsupertype/Edge"/>
```

Example 2-5   Using nested packages

```
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" nsPrefix="nested"
   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="nested"
   nsURI="http://www.redbooks.ibm.com/sal330r/example/nested" >
   <eSubpackages name="workflowsupertype" nsPrefix="workflow"
   nsURI="http://www.redbooks.ibm.com/sal330r/example/workflow3">
   ... contents of workflow package ...
   </eSubpackages>
   <eSubpackages name="diagram" nsPrefix="diagram"
   nsURI="http://www.redbooks.ibm.com/sal330r/example/diagram">
      ... DiagramNode class ...
      <eClassifiers xsi:type="ecore:EClass" name="DiagramConnection">
         <eReferences name="model" eType="#//workflowsupertype/Edge"/>
         <eReferences name="sourceNode" eType="#//diagram/DiagramNode"
         lowerBound="1" eOpposite="#//diagram/DiagramNode/sourceConnections"/>
         <eReferences name="targetNode" eType="#//diagram/DiagramNode"
         eOpposite="#//diagram/DiagramNode/targetConnections"/>
      </eClassifiers>
      <eClassifiers xsi:type="ecore:EClass" name="ContainerDiagramNode"
      eSuperTypes="#//diagram/DiagramNode">
```

```
        <eReferences name="model" eType="#//workflowsupertype/Workflow"/>
        <eReferences name="children" eType="#//diagram/DiagramNode"
        upperBound="-1" containment="true"
        eOpposite="#//diagram/DiagramNode/container"/>
      </eClassifiers>
   </eSubpackages>
</ecore:EPackage>
```

Tip: When using nested sub-packages, be sure that each package has a unique nsURI.

Declaring datatypes

EMF provides datatypes such as EString and EInt, which represent the basic Java types that you can use for simple attributes. If you need to use a different Java type, you need to create an EDataType to represent it. For example, we use EString to represent attributes such as condition of ConditionalOutputPort and whileCondition for LoopTask from the WorkflowModel for the sample application. If we wanted to represent these conditions with a specific existing Java type instead, we would declare an EDataType corresponding to that type, as follows:

```
   <eClassifiers xsi:type="ecore:EDataType" name="Condition"
   instanceClassName="com.example.Condition"/>
```

Adding operations

We can augment the classes in our model by adding operations to them. Aside from the convenience of having the signatures and skeletons generated into the code, there is little difference between adding the operations directly to the code as methods and adding the operations to the model. In both cases you will need to implement the methods in the generated code. A good approach is to define the signatures of the methods that you want to be public in your model, then complete the generated skeletons to implement them.

Annotating the model

The Ecore model includes an EAnnotation object that can be added to any model element. EAnnotations represent additional information that is associated with a model object, and they take the form of key and value pairs. You may choose to use EAnnotations to provide hints or additional information about how to use or represent model objects in an application, to represent additional constraints that are evaluated using another tool, or you may choose simply to use these annotations to document your model. An example of using EAnnotations to provide additional information about a model is described in 2.3.3, “Using the XSD plug-in to customize serialization” on page 70. The XSD plug-in uses EAnnotations to map model objects to XML.

2.1.2 Migrating existing models

The EMF documentation describes how to import from models expressed using annotated Java interfaces, models created using Rational Rose®, and models represented by an XML Schema. In this section, we discuss migrating existing models, focusing on migrating an XML Schema to EMF as an example. We provide examples to illustrate the correspondences between concepts from XML Schema and concepts provided by EMF Ecore.

For information about migrating models expressed using other frameworks, please refer to the following documents, which are linked from the documents section of the EMF project site at:

http://www.eclipse.org/emf/:

UML:

- *Tutorial: Generating an EMF model*
- *Specifying Package Information in Rose*

Annotated Java interfaces:

- *Tutorial: Generating an EMF model*
- *Using EMF* (Catherine Griffin's Eclipse Corner article)

Migrating from XML Schema to EMF is described in the *Tutorial: Generating an EMF Model using XML Schema.* The first page of the tutorial briefly outlines the mapping used to create EMF models from an XML Schema. In this section, we provide examples that illustrate this mapping. We use the purchase order XML Schema shown in Example 2-6 as the source for our new EMF model. Notice that this schema is taken from the XML Schema Part 0: Primer W3C Recommendation, 2 May 2001.[1] The examples for this section can be found in the MigrateFromXMLSchema project, in the examples provided with this book.

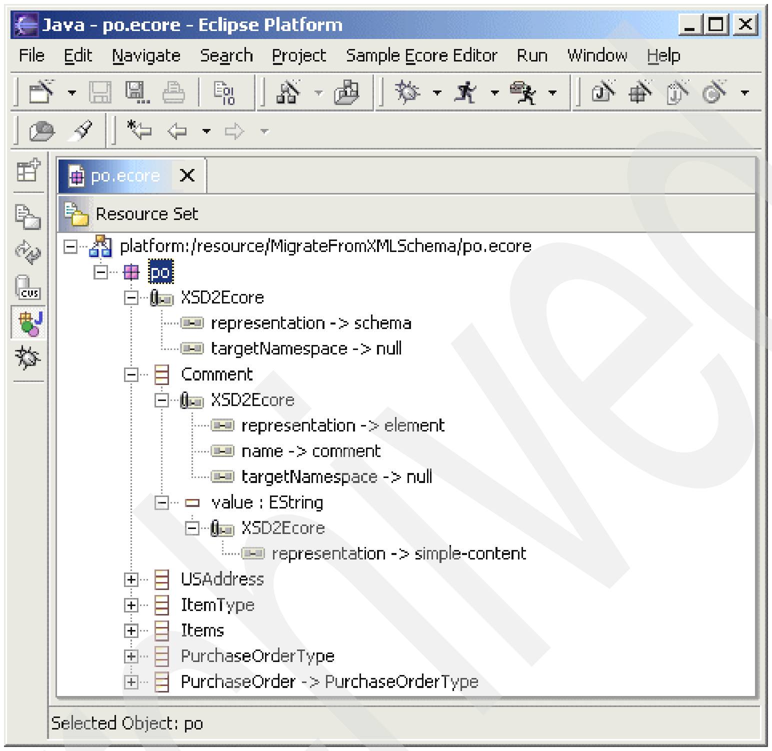
Example 2-6 Example XML Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:annotation>
      <xsd:documentation xml:lang="en">
         Purchase order schema for Example.com.
         Copyright 2000 Example.com. All rights reserved.
      </xsd:documentation>
   </xsd:annotation>
   <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
   <xsd:element name="comment" type="xsd:string"/>
   <xsd:complexType name="PurchaseOrderType">
      <xsd:sequence>
```

[1] Copyright ©2001 W3C® (MIT, INRIA, Keio), All Rights Reserved. W3C liability, trademark, document use and software licensing rules apply. http://www.w3.org/Consortium/Legal/

```
         <xsd:element name="shipTo" type="USAddress"/>
         <xsd:element name="billTo" type="USAddress"/>
         <xsd:element ref="comment" minOccurs="0"/>
         <xsd:element name="items"  type="Items"/>
      </xsd:sequence>
      <xsd:attribute name="orderDate" type="xsd:date"/>
   </xsd:complexType>
   <xsd:complexType name="USAddress">
      <xsd:sequence>
         <xsd:element name="name"   type="xsd:string"/>
         <xsd:element name="street" type="xsd:string"/>
         <xsd:element name="city"   type="xsd:string"/>
         <xsd:element name="state"  type="xsd:string"/>
         <xsd:element name="zip"    type="xsd:decimal"/>
      </xsd:sequence>
      <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
   </xsd:complexType>
   <xsd:complexType name="Items">
      <xsd:sequence>
         <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="productName" type="xsd:string"/>
                  <xsd:element name="quantity">
                     <xsd:simpleType>
                        <xsd:restriction base="xsd:positiveInteger">
                           <xsd:maxExclusive value="100"/>
                        </xsd:restriction>
                     </xsd:simpleType>
                  </xsd:element>
                  <xsd:element name="USPrice"  type="xsd:decimal"/>
                  <xsd:element ref="comment"   minOccurs="0"/>
                  <xsd:element name="shipDate" type="xsd:date"
                     minOccurs="0"/>
               </xsd:sequence>
               <xsd:attribute name="partNum" type="SKU" use="required"/>
            </xsd:complexType>
         </xsd:element>
      </xsd:sequence>
   </xsd:complexType>
   <!-- Stock Keeping Unit, a code for identifying products -->
   <xsd:simpleType name="SKU">
      <xsd:restriction base="xsd:string">
         <xsd:pattern value="\d{3}-[A-Z]{2}"/>
      </xsd:restriction>
   </xsd:simpleType>
</xsd:schema>
```

When we import our model, as the tutorial describes, each namespace declared as a targetNamespace of an XML Schema is represented in EMF as an EPackage. In our case, we only have one targetNamespace, so a single EPackage is created, as shown in Figure 2-5.



```
platform:/resource/MigrateFromXMLSchema/po.ecore
  po
    XSD2Ecore
      representation -> schema
      targetNamespace -> null
    Comment
      XSD2Ecore
        representation -> element
        name -> comment
        targetNamespace -> null
      value : EString
        XSD2Ecore
          representation -> simple-content
    USAddress
    ItemType
    Items
    PurchaseOrderType
    PurchaseOrder -> PurchaseOrderType
```

Figure 2-5 EMF model from XML Schema

If the schema that you are importing from has a targetNamespace, then the nsURI of the generated EPackage is set to that URI, and the name and nsPrefix are derived from that URI. For example, if the targetNamespace is `http://www.example.com`, then the nsPrefix is com.example, and the name is example. If the targetNamespace is `http://www.example.com/foo`, then the name is foo and the nsPrefix is com.example.foo.

Example 2-7 shows how the features of the EPackage created from po.xsd are populated by the mapping. The purchase order schema did not have a targetNamespace, so the URI to the schema file is used as the nsURI instead, and the name of the file is used for the nsPrefix and name.

Example 2-7 EPackage from XML Schema

```
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="po"
   nsURI="platform:/resource/MigrateFromXMLSchema/po.xsd" nsPrefix="po">
   ...
</ecore:EPackage>
```

You may notice that the XSD plug-in generates EAnnotations for each of the objects in the model. These annotations describe how the model maps to the schema, and is used to serialize model instances so that they conform to the XML Schema from which the EMF model was generated. We discuss how to modify these annotations to control serialization in 2.3.3, "Using the XSD plug-in to customize serialization" on page 70.

Types from the XML Schema become EClassifiers: complex types, which represent types that contain elements or attributes, are represented by EClasses in EMF. Example 2-8 shows the EClass mapped from the USAddress complex type. Notice that the representation of this EClass is `type`, because it has been generated from a type.

Example 2-8 EClass for the USAddress type

```
<eClassifiers xsi:type="ecore:EClass" name="USAddress">
   <eAnnotations source="http:///org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
      <details key="representation" value="type"/>
      <details key="name" value="USAddress"/>
      <details key="targetNamespace"/>
   </eAnnotations>
   ...
</eClassifiers>
```

Elements of this type are mapped to EReferences within the EClass representing the containing type. For example, the USAddress type is the type of the shipTo element, contained within the PurchaseOrderType. Hence, as shown in Example 2-9, shipTo is represented as an EReference within the EClass created for PurchaseOrderType. Notice that the representation is element, because the EReference was mapped from an element declaration in the XML Schema.

Example 2-9 EReference for element of complex type

```
<eReferences name="shipTo" eType="#//USAddress" lowerBound="1"
   containment="true">
   <eAnnotations source="http:///org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
      <details key="representation" value="element"/>
      <details key="name" value="shipTo"/>
      <details key="targetNamespace"/>
   </eAnnotations>
</eReferences>
```

EDataTypes are used to represent simple types that represent atomic values. For example, SKU is represented by EString in the model. For XML elements that are of a simple type, such as Comment from the purchase order schema, an EClass representing the element is created, and an EAttribute is used to represent the content. Example 2-10 shows the Comment EClass. Notice that the representation of the value attribute is `simple-content`, that is, it provides the actual content of the `comment` element.

Example 2-10 EClass from simple-typed element

```
<eClassifiers xsi:type="ecore:EClass" name="Comment">
   <eAnnotations source="http:///org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
      <details key="representation" value="element"/>
      <details key="name" value="comment"/>
      <details key="targetNamespace"/>
   </eAnnotations>
   <eAttributes name="value"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString">
      <eAnnotations
      source="http:///org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
         <details key="representation" value="simple-content"/>
      </eAnnotations>
   </eAttributes>
</eClassifiers>
```

Every simple-typed attribute in the XML Schema maps to an EAttribute belonging to the EClass mapped from the containing XML element. When the type of the XML Schema attribute has been mapped to an EClass (which is true for types such as anyURI), then the attribute is mapped to an EReference instead. We see an example in Example 2-11. The representation is `attribute` to indicate that it was mapped from an XML attribute.

Example 2-11 EAttribute from XML attributes

```
<eAttributes name="orderDate" eType="ecore:EDataType
   http://www.eclipse.org/emf/2002/Ecore#//EString">
   <eAnnotations
      source="http:///org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
      <details key="representation" value="attribute"/>
      <details key="name" value="orderDate"/>
      <details key="targetNamespace"/>
   </eAnnotations>
</eAttributes>
```

2.2 EMF.Edit-based editors and code generation

The *Tutorial: Generating an EMF Model* describes how to use the GenModel wizard to create a GenModel for the WorkflowModel, and how to generate plug-ins that can be used to create and edit WorkflowModel instances. In this section, we describe the correspondences between generated plug-ins and the model from which they are generated, by examining the code produced for the model, edit, and editor plug-ins generated from the WorkflowModel. We then discuss how to customize these generated plug-ins using code generation properties.

2.2.1 The generated plug-ins

In this section we describe the model, edit, and editor plug-ins generated for the WorkflowModel, and discuss the correspondences between the model and the generated code. The generated plug-ins are provided in the sample code provided with this book.

Note: The JET framework is used to generate model, edit, and editor plug-ins from EMF models. The templates that are used to generate these plug-ins are located in:

<ECLIPSEHOME>/plugins/org.eclipse.emf.codegen.ecore_<EMFVERSION>
/templates

Where <ECLIPSEHOME> is the location where you installed Eclipse, and <EMFVERSION> is the version of the EMF plug-in that you have installed.

We discuss the JET framework and how to customize code generation using templates in 2.4, “Using JET to customize code generation” on page 79.

The model plug-in

In this section, we describe the code in the model plug-in generated from the WorkflowModel.

Packages

For each EPackage, two or three Java packages are generated. For the WorkflowModel, these packages are workflow, workflow.impl, and workflow.util. Notice that there may be a prefix used to generate package names, as discussed in “Package-level GenModel properties” on page 54; however, for the purposes of describing the generated plug-ins in this section, we will ignore it. The util package is optional: Its presence will depend on the code generation properties. The util package is generated when the code generation properties are set to their defaults.

Classes

For each EClass in the EPackage, an interface is generated in the base package, and a Java class that implements it is generated in the impl package. If the EClass inherits from another EClass, then the generated interface and implementation extend the interface and implementation generated for the supertype.

If a class has multiple supertypes, then the first supertype in the eSuperTypes list is considered to be the primary supertype. The generated implementation for a subclass extends the corresponding implementation class of the primary supertype, and implements methods defined in the interfaces generated for any other supertypes. For example, for the WorkflowDiagram model from “Working with packages” on page 35, WorkflowDiagramTask extends TaskImpl (the primary supertype), and implements the methods from DiagramNode.

Features

For each feature, getter and setter methods are generated in the class and interface generated from their containing class. A field to cache the value of the feature is also generated if the feature is not volatile. If a feature is not changeable, then only getter methods are generated.

For multi-valued attributes and references, an EList is used to represent the feature, while single valued attributes are represented by the type of that attribute, for example EString, or the instanceClass of a user-defined EDataType. The type of the EList used to represent features will depend on the constraints in the model, for example, a non-containment reference is represented by an EObjectWithInverseResolvingEList while a containment reference is represented by an EObjectContainmentWithInverseEList.

The reflective methods such as eSet() are generated from all features for the containing class.

Operations

For each EOperation, a public method signature is generated in the interface of the containing class, and a skeleton implementation is generated in the corresponding implementation.

DataTypes

For each EEnum, an implementation is generated that extends org.eclipse.common.util.AbstractEnumerator. For other EDataTypes, there are no interfaces or implementations generated; instead, their instanceClass is used directly for EAttributes of that type.

The edit plug-in

ItemProviders are generated for each class in the edit plug-in in the provider package. In addition, an EMFPluginClass is generated for the entire plug-in. The ItemProviders extend org.eclipse.emf.edit.provider.ItemProviderAdaptor and adapt the corresponding EObject from the model. For example, WorkflowElementItemProvider adapts a WorkflowElement. The ItemProvider forwards some notifications received when the model object changes via fireNotifyChanged(), and filters the rest. You can control which notifications are filtered when you generate the plug-in, as described in 2.2.2, “Customizing code generation through GenModel properties” on page 47.

ItemProviders also manage property descriptors for all features of the class, as well as an icon and description for the class, returned by the getImage() and getText() methods.

An ItemProviderAdaptorFactory is also generated for all of the generated ItemProviders. For the WorkflowModel, it is WorkflowItemProviderAdaptorFactory. Refer to *The EMF.Edit framework and code generator overview* for more information about these factories.

The editor plug-in

For each model, three classes are generated in the editor plug-in, in the presentation package. There is a multi-page editor, which creates several different JFace viewers for the model, including a TreeViewer which use the ItemProviders from the edit plug-in as their content and label providers. The editor also creates an outline view and property sheet page that displays the properties for the selected object from the viewers.

An ActionBarContributor is also generated, that is used to create the context menu to add children or siblings to selected items from the viewers in the editor.

The final class generated in the editor plug-in is a wizard, which allows you to create a new resource containing an instance of one of your model objects.

2.2.2 Customizing code generation through GenModel properties

The EMF Users’ Guide describes the properties defined for each of the Ecore objects in a model. Some of these properties affect the way in which code is generated from the model and are duplicated in the GenModel for that model. In 2.2.1, “The generated plug-ins” on page 45, we examine the code generated for the WorkflowModel’s model, edit, and editor plug-ins. For any Ecore model, the generation of the model, edit, and editor plug-ins is driven by the properties represented in the GenModel created for that model. In this section, we examine those properties, and discuss the effect that changing them has on code generation.

If we examine the GenModel for the WorkflowModel using a text editor, we can see that it is described using XMI. This is because the GenModel is itself an EMF model, so its instances are serialized by default according to the XMI 2.0 production rules[2], as described in 2.3.2, “Default serialization of model instances” on page 66.

Example 2-12 shows the top-level XMI element from the GenModel for the WorkflowModel. As we can see, the GenModel has properties (represented in the XMI as attributes) that identify the model from which the edit, and editor plug-ins are generated, and that specify the name and package of the generated plug-ins. We provide details of the effect that these properties have on code generation in “Top-level GenModel properties” on page 52.

Example 2-12 Top-level element for WorkflowModel GenModel

```
<genmodel:GenModel
   xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
   xmlns:genmodel="http://www.eclipse.org/emf/2002/GenModel"
   modelDirectory="/WorkflowModel/src"
   editDirectory="/WorkflowModel.edit/src"
   editorDirectory="/WorkflowModel.editor/src"
   modelPluginID="WorkflowModel" modelName="Workflow"
   editPluginClass="WorkflowPackage.provider.WorkflowEditPlugin"
   editorPluginClass="WorkflowPackage.presentation.WorkflowEditorPlugin">
   ...
</genmodel:GenModel>
```

Nested within the top level element of the WorkflowModel’s GenModel XMI, we find elements corresponding to each object from the WorkflowModel, with attributes representing the properties specific to each object. The nesting of the contents of a GenModel XMI matches the nesting within the source Ecore model, with elements corresponding to classes, data types, and sub-packages nested within the element corresponding to their containing package; and elements corresponding to references, attributes, and operations nested within the element corresponding to their containing class.

In Example 2-13, we see a fragment of the GenModel for the WorkflowModel that corresponds to the Workflow class.The `genClasses` element corresponding to the Workflow class contains `genFeatures` elements that correspond to the name attribute, and to the task and edge references of the Workflow class. The effect on code generation of the properties represented for each class is discussed in “Class-level GenModel properties” on page 55.

[2] For more information, refer to the *XML Metadata Interchange (XMI) Specification, Version 2.0*, which can be found at: http://www.omg.org/technology/documents/formal/xmi.htm

Aside: If you have installed the org.eclipse.emf.source plug-in, you can take a look at the file GenModel.ecore, which describes the GenModel. The zip file containing the file is usually installed at the following location:

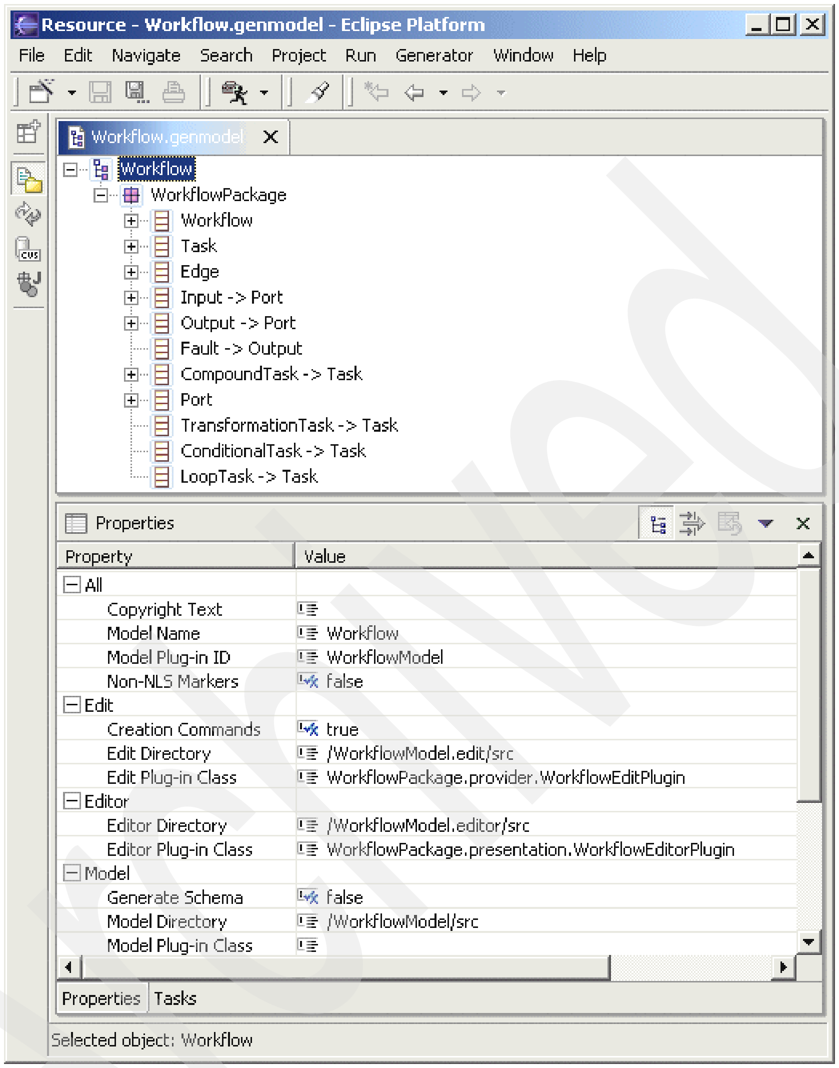
<ECLIPSEHOME>/plugins/org.eclipse.emf.source_<EMFVERSION>/src/org.eclipse.emf.codegen.ecore_<EMFVERSION>/runtime/codegen.ecoresrc.zip

Where <ECLIPSEHOME> is the location where you installed Eclipse, and <EMFVERSION> is the version of the EMF plug-in that you have installed.

Example 2-13 GenModel fragment for EClass Workflow

```
<genClasses ecoreClass="Workflow.ecore#//Workflow">
   <genFeatures
      ecoreFeature="ecore:EAttribute Workflow.ecore#//Workflow/name"/>
   <genFeatures property="None" children="true"
      ecoreFeature="ecore:EReference Workflow.ecore#//Workflow/task"/>
   <genFeatures property="None" children="true"
      ecoreFeature="ecore:EReference Workflow.ecore#//Workflow/edge"/>
</genClasses>
```

While you can control the generation of plug-ins by editing the GenModel XMI directly, you can also edit the GenModel properties with the editor provided by the GenModel plug-in. Figure 2-6 shows the GenModel editor displaying the top-level properties from the GenModel for the WorkflowModel. As the figure shows, the tree view provided by the GenModel editor mirrors the containment hierarchy from the GenModel XMI and the source model, and displays the properties for the selected item in the Properties view. If you do not see the properties, select **Window -> Show View -> Other** and then select **Properties** from the **Basic** item in the tree.


Resource - Workflow.genmodel - Eclipse Platform
File Edit Navigate Search Project Run Generator Window Help
Workflow.genmodel
Properties
Properties Tasks
Selected object: Workflow

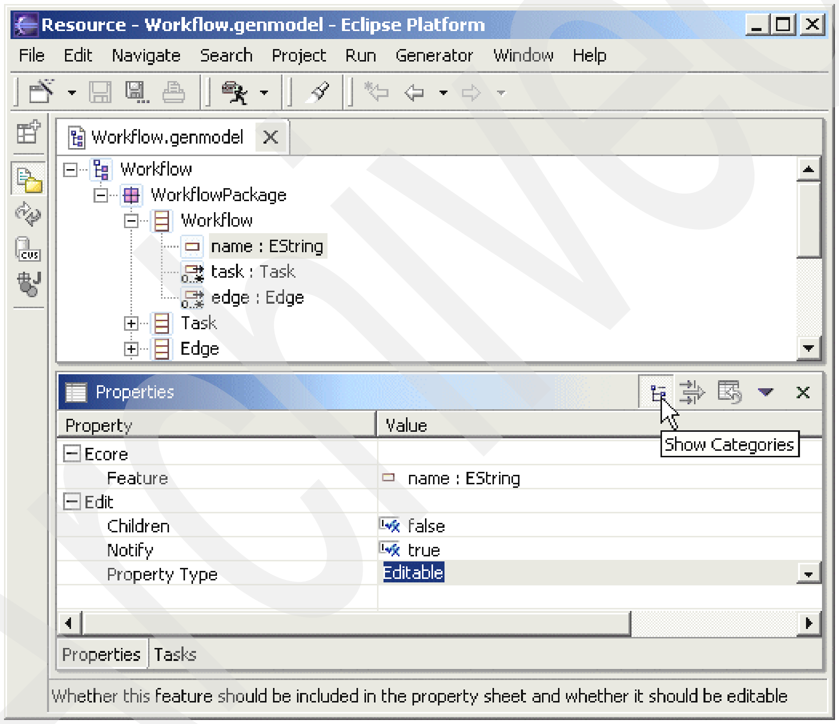
```
Workflow
  WorkflowPackage
    Workflow
    Task
    Edge
    Input -> Port
    Output -> Port
    Fault -> Output
    CompoundTask -> Task
    Port
    TransformationTask -> Task
    ConditionalTask -> Task
    LoopTask -> Task
```

Property	Value
All	
Copyright Text	
Model Name	Workflow
Model Plug-in ID	WorkflowModel
Non-NLS Markers	false
Edit	
Creation Commands	true
Edit Directory	/WorkflowModel.edit/src
Edit Plug-in Class	WorkflowPackage.provider.WorkflowEditPlugin
Editor	
Editor Directory	/WorkflowModel.editor/src
Editor Plug-in Class	WorkflowPackage.presentation.WorkflowEditorPlugin
Model	
Generate Schema	false
Model Directory	/WorkflowModel/src
Model Plug-in Class	

Figure 2-6 Top-level GenModel properties for WorkflowModel

An advantage of using the GenModel editor over editing the XMI directly is that the properties are organized by category, as we see in Figure 2-6, where the properties for the WorkflowModel GenModel are categorized according to whether they relate to the generation of the model, edit, or editor plug-ins. There is also a Templates & Merge category that is not shown in the figure, which we discuss in 2.4, "Using JET to customize code generation" on page 79.

You can toggle between the categorized view and a flat view of the properties by clicking the button identified by the tree icon, as shown in Figure 2-7, with the tool tip Show Categories. Here we see the properties for the name attribute of the class Workflow. The property view allows us to view and edit all of the properties associated with each object in the model. The editor provides a brief description of each of the properties, which is displayed in the status bar whenever a property is selected, as shown for the Property Type property. For properties that have a fixed set of values, such as Property Type here, the editor provides a pull-down list from which you may select an alternate value. Notice that the XMI file representing the model may not explicitly persist a property that is unchanged from its default value, as shown in Example 2-13 on page 49, where none of the properties in the Edit category for the name attribute are present in the XMI.



Property	Value
Ecore	
Feature	name : EString
Edit	
Children	false
Notify	true
Property Type	Editable

Whether this feature should be included in the property sheet and whether it should be editable

Figure 2-7 Using the GenModel editor to edit properties

In addition to specifying code generation properties using the GenModel editor, you may provide values for some of these properties when you initially create or import your model from XMI or annotated Java interfaces. When you use the GenModel wizard to create a GenModel from your model, the values that you supply in your model are used to populate the GenModel, and for any properties for which you do not supply a value, a default value is used instead.

In the following sections, we detail the GenModel properties, organizing them according to the GenModel hierarchy. At each level, we provide a table that outlines the name of the property as it appears in the GenModel editor, the category to which the property belongs in the GenModel editor, the attribute used to represent the property in the GenModel XMI and the default value provided by the GenModel wizard for that property. We also discuss the effect that changing each property from its default has on the generation of the model, edit, and editor plug-ins.

Top-level GenModel properties

The properties represented at the top level for each GenModel are described in Table 2-1.

Table 2-1 Top-level GenModel properties

Property	Category	XMI attribute	Default Value
Copyright Text	All	copyrightText	
Creation Commands	Edit	creationCommands	true
Edit Directory	Edit	editDirectory	<PROJECT>.edit/src
Editor Directory	Editor	editorDirectory	<PROJECT>.editor/src
Editor Plug-in Class	Editor	editorPluginClass	<basePackage of top-level EPackage><modelName> EditorPlugin
Edit Plug-in Class	Edit	editPluginClass	<basePackage of top-level EPackage><modelName> EditPlugin
Generate Schema	Model	generateSchema	false
Model Directory	Model	modelDirectory	<PROJECT>/src
Model Name	All	modelName	GenModel base filename
Model Plug-in Class	Model	modelPluginClass	
Model Plug-in ID	All	modelPluginID	<PROJECT>
Non-NLS Markers	All	nonNLSMarkers	false

The copyrightText property provides the value for the final static field copyright in every generated Java class in the model and edit plug-ins. By default, the copyright field is set to be the empty string. Notice that this field is not generated in the classes for the editor plug-in.

The creationCommands property controls whether or not the generated edit plug-in includes support for creating new objects. If creationCommands is false, the generated editor only allows properties of existing objects to be modified, and the menu options for creating new child or sibling objects are not present. If creationCommands is true, in the edit plug-in, each ItemProvider generated from each class in the model contains a method collectNewChildDescriptors, which constructs a list of the types of children objects that can be created. These lists are used by the editor plug-in to construct actions that can be used to create children and sibling objects.

The modelName property is used to construct the default names of the edit, and editor plug-ins. The values of the modelDirectory, editDirectory and editorDirectory properties determine the projects, and path within those projects, into which the plug-ins are generated, while the modelPluginClass, editPluginClass and editorPluginClass properties determine the Java package of each generated plug-in.

The modelPluginID property is used as the plug-in ID of the model plug-in and is referenced from the edit plug-in, which depends on the model plug-in. If you change the value of this property, you need to delete the plugin.xml files from the model and edit plug-ins before regenerating the code, to ensure that they are updated.

Setting the generateSchema property to true means that the XML Schema for the model is generated whenever the model plug-in is generated. When you generate the schemas, you will notice new schema files appear in the project; XMI.xsd and <PackageName>XMI.xsd, where <PackageName> is the name of the top-level package from your model. For example, the XML Schema files generated from the WorkflowModel are XMI.xsd and WorkflowXMI.xsd. XMI.xsd declares XMI elements and attributes that are common to all models, while WorkflowXMI.xsd contains the only element and attribute declarations specific to serializing WorkflowModel instances.

Tip: For the XML Schema generation to succeed, you must have installed the XSD plug-in available from `http://www.eclipse.org/xsd/`.

Setting nonNLSMarkers controls whether National Language Support (NLS) comment markers, marking particular strings as non-translatable, are generated in the source of the plug-ins. Example 2-14 shows a code fragment from the `toString` method from the class PortImpl, generated as part of the model plug-in from the WorkflowModel. We see that the strings `name` and `condition` are marked as NON-NLS, that is, that they are not translatable.

When nonNLSMarkers is true, strings that are used as keys to lookup resource bundles and strings based on the names of model objects (such as `name` and `condition` in this example), are marked as NON-NLS. However, some strings, such as those that represent default values for EString-typed attributes, remain unmarked when this property is true. For more information about internationalizing Eclipse plug-ins, see:

http://www.eclipse.org/articles/Article-Internationalization/how2I18n.html

Example 2-14 Generated NON-NLS markers

```
public String toString() {
   ...
   result.append(" (name: "); //$NON-NLS-1$
   result.append(name);
   result.append(", condition: "); //$NON-NLS-1$
   result.append(condition);
   ...
}
```

Package-level GenModel properties

For each EPackage in the model, there is a corresponding genPackages element in the GenModel XMI, which is represented in the GenModel editor as an item in the tree view. The properties represented for each package are presented in Table 2-2.

Table 2-2 Package-level GenModel properties

Property	Category	XMI attribute	Default Value
Adaptor Factory	Model	adapterFactory	true
Base Package	All	basePackage	
Package	Ecore	ecorePackage	EPackage name
Prefix	All	prefix	EPackage name (capitalized)
Resource Type	Model	resource	None

The ecorePackage property identifies the corresponding EPackage from the source model. The prefix property is used to generate the names for the AdapterFactory, Package, Factory and Switch classes generated for the EPackage. The prefix should begin with an upper-case character so that the generated classes have upper-case names.

If a value is specified for basePackage, that value is used as the package prefix for the generated plug-ins. For example, to generate the model interfaces for our WorkflowModel plug-in into the package `com.ibm.itso.workflow`, we set basePackage for the Workflow package to `com.ibm.itso` and check that the name of the Workflow package in the WorkflowModel is lower case to ensure that we follow java package naming convention. If you generate the GenModel from a model where the top-level package has a fully qualified Java name, the wizard fills in the basePackage property with the prefix from that package.

Note, if your model contains sub-packages, these are represented in the GenModel as nestedGenPackages elements. The default basePrefix for each nestedGenPackages element is the package name constructed from the basePrefix and ecorePackage properties of the containing genPackages element. If you change the basePrefix for a sub-package, the code generated for the objects directly contained by that sub-package is generated into the package specified by the sub-package's basePrefix and the sub-package name.

The value of adaptorFactory indicates whether an AdapterFactory and Switch is generated for the EPackage, in the corresponding util package.

The resource property indicates whether to generate a Resource and ResourceFactory implementation for the model, and the type of Resource to subclass when doing so. When this property is set to None, as it is by default, the generated editor uses an XMIResource to serialize model instances, as described in 2.3.2, "Default serialization of model instances" on page 66. If resource is set to Basic, a subclass of ResourceImpl is generated in the util package, which can then be modified to customize serialization to any format. Similarly, setting resource to XML or XMI means that the generated ResourceImpl is a subclass of XMLResourceImpl or XMIResourceImpl, respectively, and you can customize these serializations as described in 2.3.5, "Providing a custom resource implementation" on page 75.

Class-level GenModel properties

Classes are represented in the GenModel as genClasses elements. The properties for each class are shown in Table 2-3.

Table 2-3 Class-level GenModel properties

Property	Category	XMI attribute	Default Value
Class	Ecore	ecoreClass	EClass name
Image	Edit	image	true
Label Feature	Edit	labelFeature	
Provider Type	Edit	provider	Singleton

The ecoreClass property identifies the corresponding EClass from the source model.

The provider property indicates which item provider pattern is used to generate the ItemProvider for this class in the editor plug-in; Singleton, Stateful, or None. Refer to the Item provider implementation classes section, from *The EMF.Edit framework and code generator overview*, in the Documents section of the EMF project site at: `http://www.eclipse.org/emf/` for details of the Singleton and Stateful pattern. If the property is set to None, no Item Provider is generated for the class.

The image property indicates whether an icon is generated for the class in the corresponding ItemProvider, which is returned by the getImage() method.

The labelFeature property identifies the attribute that is used to provide the default label for objects of this type, which is returned by the getText() method in the generated ItemProvider. If this property is not set, then the code generation will look for an attribute called name (or with name as a substring in its name) to use instead, and if that does not exist, it will use the first simple attribute (that is, an attribute that is of a simple type such as EString) from the class.

Feature-level GenModel properties

The GenModel represents each attribute and reference as a genFeatures element in the XMI. The properties for each feature are listed in Table 2-4.

Table 2-4 Feature-level GenModel properties

Property	Category	XMI attribute	Default Value
Children	Edit	children	true for containment references, otherwise false
Feature	Ecore	ecoreFeature	The name of the EAttribute or EReference
Notify	Edit	notify	true for attributes and containment references
Property Type	Edit	property	None for containment/container references, Editable for normal references and attributes, Readonly for features where changeable is false

The corresponding feature (EAttribute or EReference) from the source model is identified by the ecoreFeature property.

The children property indicates whether this feature is considered to be a child of the containing class, for the purposes of constructing the tree view in the editor, and whether the context menu for the parent provides an option to create this feature as a child. Most features are represented as properties, and so children is usually false, however for containment references, children is true by default.

The notify property indicates whether the ItemProvider forwards notifications indicating that the feature has changed. By default, the code generated in the model code notifies whenever any feature changes, however the generated ItemProviders filter these notifications. By default, notifications of changes to attributes and containment references are forwarded, while non-containment reference changes are not.

The property indicates whether this feature is represented as a property on the property sheet, and whether its value can be edited via the property sheet. Features represented as children are usually not included in the property sheet, so it is usual to see containment references with None as the value for property, and attributes and non-containment references with this value set to Editable. Features that have changeable set to false in the Ecore model will be Readonly by default.

GenModel properties for DataTypes

EDataTypes are represented in the GenModel as genDataTypes elements. The ecoreDataType property identifies the associated EDataType from the model. As EDataTypes already reference their implementation class, the GenModel does not represent any other code generation properties for them. Similarly, EEnums are represented in the GenModel by genEnums elements, with an ecoreEnum property referring to the EEnum from the model. The literals are represented by genEnumLiterals elements nested within the corresponding genEnums element, again with a single property, ecoreEnumLiteral, that refers to the EEnumLiteral from the model.

GenModel properties for operations and parameters

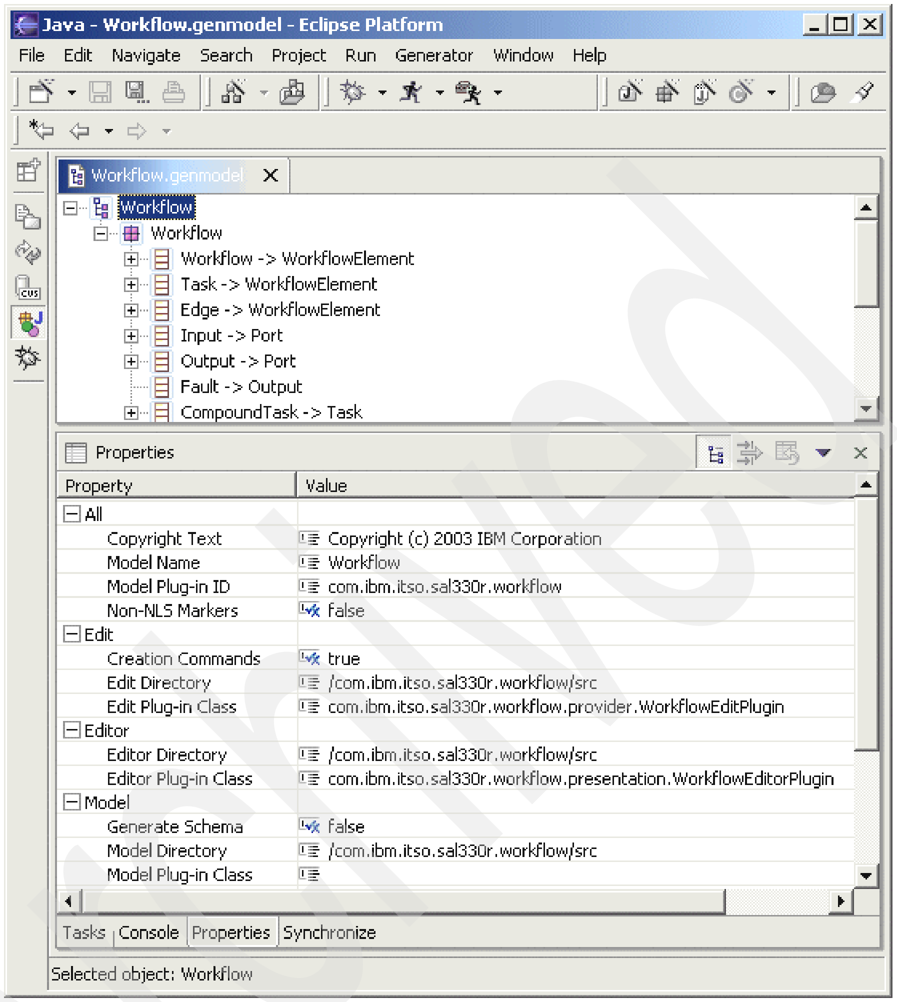
Operations are represented as genOperations elements, which contain genParameters elements for each parameter. Apart from the ecoreOperation property of genOperations, and the ecoreParameter property of genParameters identifying the associated model objects, there are no other GenModel properties associated with operations or parameters. If you want to add methods to the generated code, it makes little difference whether you add them to the model first and generate skeletons, or simply add them directly to the generated code. If you do generate them from the model, make sure that you remove the @generated tag when you implement them so that your implementation is not overwritten if you regenerate the code.

Customization example

We can see that the properties at the top-level of the GenModel generally affect the names, packages, and locations of the generated model, edit, and editor plug-ins, while the GenModel properties at the package, class, and feature level affect only the types generated from those model elements. The default values generate three separate plug-ins, such as the plug-ins that we examined in “The generated plug-ins” on page 45. In the following example, we change some of the top-level GenModel properties in order to customize the generated plug-ins.

Our example customizes the code generation so that the model, edit, and editor are generated into a single plug-in, to make it easier to package for distribution. These are the steps that we take to generate a single plug-in called com.ibm.itso.sal330r.workflow, for our WorkflowModel editor:

1. In our existing WorkflowModel project, we generate the GenModel for the WorkflowModel and open it using the GenModel editor.
2. We change the modelPluginID to `com.ibm.itso.sal330r.workflow`, as shown in Figure 2-8. This is the identifier that is used for the plug-in containing the model, edit, and editor code.
3. We edit modelDirectory, editDirectory, and editorDirectory properties so that they are all set to `/com.ibm.itso.sal330r.workflow/src`. When we generate the plug-ins, the com.ibm.itso.sal330r.workflow project is created if it does not already exist. The code for all three plug-ins is generated to the src directory of this project, and a single plugin.xml is generated to describe the plug-in containing the model, edit, and editor code.
4. Edit the editPluginClass and editorPluginClass properties, as shown in Figure 2-8, so that they have the same package prefix.


Java - Workflow.genmodel - Eclipse Platform
File Edit Navigate Search Project Run Generator Window Help
Workflow.genmodel
Properties
Tasks Console Properties Synchronize

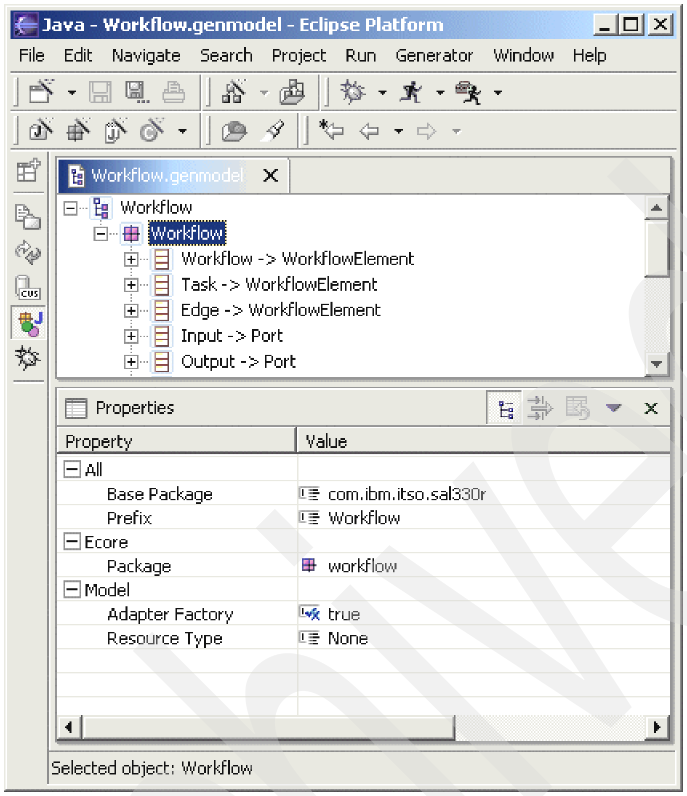
```
Workflow
  Workflow
    Workflow -> WorkflowElement
    Task -> WorkflowElement
    Edge -> WorkflowElement
    Input -> Port
    Output -> Port
    Fault -> Output
    CompoundTask -> Task
```

Property	Value
All	
Copyright Text	Copyright (c) 2003 IBM Corporation
Model Name	Workflow
Model Plug-in ID	com.ibm.itso.sal330r.workflow
Non-NLS Markers	false
Edit	
Creation Commands	true
Edit Directory	/com.ibm.itso.sal330r.workflow/src
Edit Plug-in Class	com.ibm.itso.sal330r.workflow.provider.WorkflowEditPlugin
Editor	
Editor Directory	/com.ibm.itso.sal330r.workflow/src
Editor Plug-in Class	com.ibm.itso.sal330r.workflow.presentation.WorkflowEditorPlugin
Model	
Generate Schema	false
Model Directory	/com.ibm.itso.sal330r.workflow/src
Model Plug-in Class	

Selected object: Workflow

Figure 2-8 Changing the top-level GenModel properties to generate a single plug-in

5. In order to generate the model code into the com.ibm.itso.sal330r.workflow package, we also edit the basePackage property on the workflow package, setting it to `com.ibm.itso.sal330r`, as shown in Figure 2-9. Notice that the basePackage does *not* include the package name workflow. When the model code is generated, the name of the EPackage from the model is used to construct the last part of the Java package name.


Java - Workflow.genmodel - Eclipse Platform

File Edit Navigate Search Project Run Generator Window Help

Workflow.genmodel

```
Workflow
  Workflow
    Workflow -> WorkflowElement
    Task -> WorkflowElement
    Edge -> WorkflowElement
    Input -> Port
    Output -> Port
```

Properties

Property	Value
All	
Base Package	com.ibm.itso.sal330r
Prefix	Workflow
Ecore	
Package	workflow
Model	
Adapter Factory	true
Resource Type	None

Selected object: Workflow

Figure 2-9 Changing the basePackage property

6. Select **Generate All** from the context menu of the top-level GenModel element to generate the model, edit, and editor code into the com.ibm.itso.sal330r.workflow plug-in. It is important to select **Generate All** the first time you generate the code**,** rather than choosing the model, edit, or editor options individually, so that plugin.xml contains all of the required entries for the combined plug-in.

The resulting plug-in is located in the com.ibm.itso.sal330r.workflow project.

Tip: Be sure to generate from the context menu of the top-level element in the GenModel. Generating from any item lower down in the tree will only generate code associated with that item and its children.

2.2.3 Modifying the generated code

Once you have generated the code for the model, edit, and editor plug-ins, there may still be some customizations that you need to make before you can use it. Common additions that you may make to the model code include implementing methods generated from EOperations, implementing getter and setter methods for volatile features, or adding methods that were not represented in the model.

Important: Whenever you modify part of the generated code, be sure to remove the @generated tag, or change it to read @generated NOT in the comment that describes the method, type, or field that you are modifying. If you fail to do this, your changes will be discarded next time you regenerate the code from the model.

Modifying the model plug-in

In the following example, we modify the model code generated from the WorkflowModel to implement the getter and setter methods generated for our volatile attribute id, in WorkflowElementImpl. The id attribute is volatile because we want to generate its value to ensure that it is unique. When we generate the model code, skeletons are generated for the getId() and setId() methods, as shown in Example 2-15.

Example 2-15 Methods generated for volatile feature

```
/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
public String getId() {
   // TODO: implement this method to return the 'Id' attribute
   // Ensure that you remove @generated or mark it @generated NOT
   throw new UnsupportedOperationException();
}
/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
public void setId(String newId) {
   // TODO: implement this method to set the 'Id' attribute
   // Ensure that you remove @generated or mark it @generated NOT
   throw new UnsupportedOperationException();
}
```

We modify WorkflowElementImpl to add a method to generate the id, add a field to cache the generated id, and use the method to set the value from within the getId() and setId() method, as shown in Example 2-16. We generate the id in both methods so that the id is never null when it is used.

Example 2-16 Modifying the getID() method

```
public abstract class WorkflowElementImpl extends EObjectImpl implements 
WorkflowElement {
   /**
    * Prefix used for generated ids
    */
   protected static final String idPrefix = "w";
   /**
    * The cached value of the '{@link #getId() <em>id</em>}' attribute.
    * <!-- begin-user-doc -->
    * <!-- end-user-doc -->
    * @see #getId()
    * 
    */
   protected String id;

   protected static int counter = 0;
   ...
   /**
    * Generate (and cache) an id as needed
    */
   public String getId() {
      if (id == null){
         id = generateId();
      }
      return id;
   }
   /**
    * Generate a random id based on the current time
    * @return the generated id
    */
   public synchronized String generateId(){
      long current= System.currentTimeMillis();
      return idPrefix + current + counter++;
   }
   /**
    * Set or generate an Id
    */
   public void setId(String newId) {
      if (newId == null && id == null){
         id = generateId();
      }
      else {
            id = newId;
        }
    }
}
```

We remove the @generated tag from the comments to ensure our methods are not overwritten. Notice that volatile attributes are quite commonly not changeable, and are usually also transient. This means that usually you would not need to cache the value of the attribute or provide a setter implementation. In our case, although the ids are generated, and we don't care what the value of the ids are while the objects are in memory, we use them in the serialization to make the XMI references more readable, which means that the id attribute has to be non-transient and changeable.

Modifying the edit plug-in

A common modification that you might want to make to an ItemProvider generated from a model object is to customize the getText() method. By default, this method returns the type of the object, followed by the value of the label feature for that type, and is used by the generated editor to label each item in the tree view displaying a model. For our WorkflowModel example, although Edges have a name and id, it is more useful to label them by the names of their source and target nodes. We modify the getText() method of EdgeItemProvider as shown in Example 2-17 to provide this functionality.

Example 2-17 The getText() method of EdgeItemProvider

```
   /**
    * This returns the label text for the adapted class.
    * <!-- begin-user-doc -->
    * <!-- end-user-doc -->
    * @generated NOT
    */
   public String getText(Object object) {
      Edge e = (Edge)object;
      String label = getString("_UI_Edge_type");
      String fromNode = e.getSource().getNode().getName();
      String toNode = e.getTarget().getNode().getName();
      if (!(fromNode == null || fromNode.length() == 0))
         label += " from " + fromNode;
      if (!(toNode == null || toNode.length() == 0))
         label += " to " + toNode;
      return label;
   }
```

In this case, we must take care, because the features that we are using to label the Edge are non-containment references. Remember from 2.2.2, “Customizing code generation through GenModel properties” on page 47, that notifications of changes to non-containment references are filtered by the ItemProvider and not passed to the editor by default. This means that the label will not be updated to reflect new values for the source or target if they change. We can override this behavior by setting the notify property for the source and target references of Edge in the GenModel to true, and then regenerating the edit code.

You can see the result of the changes in the default WorkflowModel editor in Figure 2-10.

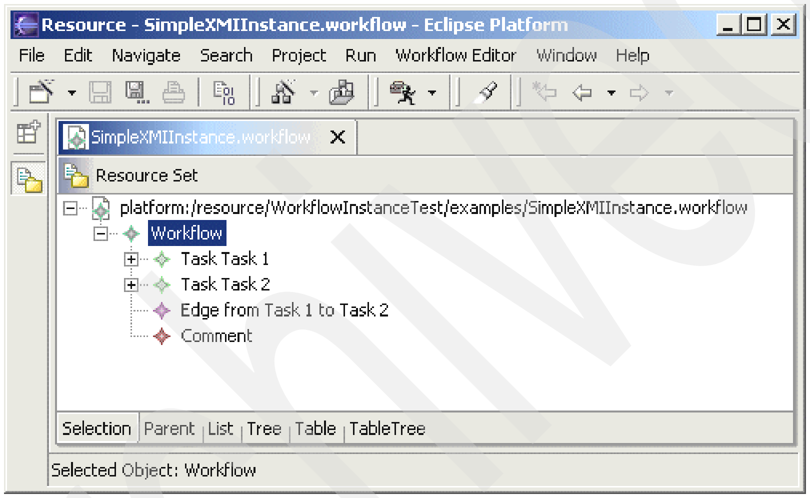

Resource - SimpleXMIInstance.workflow - Eclipse Platform
File Edit Navigate Search Project Run Workflow Editor Window Help
SimpleXMIInstance.workflow
Resource Set
platform:/resource/WorkflowInstanceTest/examples/SimpleXMIInstance.workflow
Workflow
Task Task 1
Task Task 2
Edge from Task 1 to Task 2
Comment
Selection Parent List Tree Table TableTree
Selected Object: Workflow

Figure 2-10 Editor using modified EdgeItemProvider

2.3 Model instances and serialization

In this section we examine how to create and serialize model instances. We provide examples that illustrate how to customize serialization and discuss the effect that different modeling techniques can have on the way in which instances are serialized.

2.3.1 Creating model instances

We can use the code generated for the model plug-in from our model to create instances of that model.

Example 2-18 shows how we create a Workflow instance and a Task instance using the WorkflowFactory. The example also demonstrates how we use the methods from the generated code to set properties such as the name on the Task, and maintain references, in this case adding the Task to the nodes of the Workflow, and adding an InputPort and OutputPort to the Task.

Example 2-18 Creating instances

```
Map registry = EPackage.Registry.INSTANCE;
String workflowURI = WorkflowPackage.eNS_URI;
WorkflowPackage wfPackage = (WorkflowPackage) registry.get(workflowURI);
WorkflowFactory wfFactory = wfPackage.getWorkflowFactory();
Workflow workflow = wfFactory.createWorkflow();
// add a Task to the workflow
Task t1 = wfFactory.createTask();
workflow.getNodes().add(t1);
t1.setName("Task 1");
// add an input port and an output port to the Task
t1.getInputs().add(wfFactory.createInputPort());
t1.getOutputs().add(wfFactory.createOutputPort());
...
```

If we were using the reflective API to manipulate our instance objects, we would replace methods such as setName() and getNodes() that are specific to the WorkflowModel with generic eSet() and eGet() methods, for example:
`t1.eSet(WorkflowPackage.eINSTANCE.getTask_Name(), "Task 2")`
to set the name of the Task.

An interesting application of using the reflective APIs is to work with dynamic models, that is, to work with Ecore models as in-memory objects rather than generating code from the model and using the generated classes. Example 2-19 shows an sample of how we can create instances of the Ecore model to represent a very basic model of Workflow.

Example 2-19 Creating a dynamic model

```
// Create the Workflow Package
EPackage workflowPackage = EcoreFactory.eINSTANCE.createEPackage();
// create the Port class
EClass portClass = EcoreFactory.eINSTANCE.createEClass();
portClass.setName("Port");
EClass inputPortClass = EcoreFactory.eINSTANCE.createEClass();
inputPortClass.setName("InputPort");
// set up inheritance
inputPortClass.getESuperTypes().add(portClass);
// create the Task class
EClass taskClass = EcoreFactory.eINSTANCE.createEClass();
taskClass.setName("Task");
```

```
// add name attribute to Task
EAttribute taskNameAttr = EcoreFactory.eINSTANCE.createEAttribute();
taskNameAttr.setName("name");
taskNameAttr.setEType(EcorePackage.eINSTANCE.getEString());
taskClass.getEAttributes().add(taskNameAttr);
// set up the reference between Task and InputPort
EReference taskToInputPortRef = EcoreFactory.eINSTANCE.createEReference();
taskToInputPortRef.setUpperBound(-1);
taskToInputPortRef.setLowerBound(1);
taskToInputPortRef.setEType(inputPortClass);
taskClass.getEReferences().add(taskToInputPortRef);
...
// add the classes to the package
workflowPackage.getEClassifiers().add(taskClass);
workflowPackage.getEClassifiers().add(portClass);
workflowPackage.getEClassifiers().add(inputPortClass);
...
```

We can create instances of this model using the reflective API, as Example 2-20 demonstrates.

Example 2-20 Using the reflective API to create dynamic model instances

```
EFactory wfFactory = workflowPackage.getEFactoryInstance();
EObject task1 = wfFactory.create(taskClass);
task1.eSet(taskNameAttr, "Task 1");
```

Obviously, this dynamic approach only works for some applications — if you are using a model where you would normally customize the code generated from the model, for example, to implement EOperations, then this approach is not suitable.

2.3.2 Default serialization of model instances

When you create and serialize instances of an Ecore model, they are serialized by default as XMI 2.0. This section provides examples illustrating how EMF objects are represented in the XMI.

For a more complete description of XMI 2.0, please refer to the *XML Metadata Interchange (XMI) Specification, Version 2.0*, found at:

http://www.omg.org/technology/documents/formal/xmi.htm

Note: We have already seen several examples of XMI representing model instances. The Ecore model is itself an EMF model, so the Ecore documents describing the models that we created in 2.1, “EMF modeling techniques” on page 30, were all examples of the default serialization of Ecore model instances to XMI.

All of the example serializations discussed in this section can be found in the Serializations directory in the examples provided for this section.

Example 2-21 shows XMI representing a Workflow instance, representing a Workflow containing a Comment and two Tasks, each with an input, output, and fault port, and an Edge connecting them. You can load this example from the file SimpleXMIInstance.workflow.

Example 2-21 Default XMI serialization of a Workflow instance

```
<workflow:Workflow xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="w105966221898456"
   xmlns:workflow="http://www.redbooks.ibm.com/sal330r/workflow">
   <nodes xsi:type="workflow:Task" name="Task 1" x="31" y="82"
      id="w105966222103157" isStart="true">
      <outputs xsi:type="workflow:FaultPort" name="fault"
         id="w105966226568760"/>
      <outputs name="output" id="w105966226568761" edges="w105966226568762"/>
      <inputs name="input" id="w105966226568763"/>
   </nodes>
   <nodes xsi:type="workflow:Task" name="Task 2" x="265" y="81"
      id="w105966222451558" isFinish="true">
      <outputs xsi:type="workflow:FaultPort" name="fault"
         id="w105966226568764"/>
      <outputs name="output" id="w105966226568765"/>
      <inputs name="input" id="w105966226568766" edges="w105966226568762"/>
   </nodes>
   <edges id="w105966226568762" target="w105966226568766"
      source="w105966226568761"/>
   <comments comment="This is a sample Workflow instance" x="24" y="20"
      id="w105966223168759"/>
</workflow:Workflow>
```

As the example shows, the Workflow object is serialized to an element in the XMI, with attributes representing its EAttributes and non-containment EReferences. Containment EReferences are represented as elements, with the content of the contained object contained inline, as we see for the nodes elements from the example. When the reference can be to objects of different types (that is, to different subtypes of the referenced class), the xsi:type attribute is also serialized to identify the type of the object represented by the element.

Non-containment EReferences, such as the references between each edge and its target and source Ports, are represented as strings, that identify the object being referenced. By default, the strings used to identify other objects are based on the containing resource, type and position of the referenced object. Example 2-22 shows how positional references are used to serialize a Workflow. In Example 2-21, the id attribute is used instead of positional references. This is because the id property for that attribute it set to true in the model. If the id property is true for one of the attributes, references will refer to objects using the value of that attribute, if it is set, or use a positional reference if the id attribute is not set. If you are using an id attribute, it is important to ensure that its values are unique within a resource, so that the ids in the XMI are unique within the document, as required by the standard.

Example 2-22 Positional references

```
<workflow:Workflow xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:workflow="http://www.redbooks.ibm.com/sal330r/workflow">
   <nodes xsi:type="workflow:Task" name="Task 1">
      <outputs name="output" edges="//@edges.0"/>
      <inputs name="input"/>
   </nodes>
   <nodes xsi:type="workflow:Task" name="Task 2">
      <outputs name="output1"/>
      <inputs name="input1" edges="//@edges.0"/>
   </nodes>
   <edges target="//@nodes.1/@inputs.0" source="//@nodes.0/@outputs.0"/>
</workflow:Workflow>
```

You can customize the way that references are represented in the XMI by overriding the eURIFragmentSegment() and eObjectForURIFragmentSegment() methods in your model implementation classes. The default positional references are provided by these methods in EObjectImpl, which is a superclass of all the implementation classes generated from a model.

When the references are to objects contained by another resource, then the scheme for finding the file that is the serialization of the resource (for example, http) and the name of the file is also added to the reference. An example of this is when we use cross-package references and serialize the containing Ecore EPackages into separate files, such as the following snippet taken from Example 2-4 on page 37:

```
   <eReferences name="model"
   eType="ecore:EClass WorkflowWithCommonSupertype.ecore#//Task"/>
```

Each model instance that is created by the generated editor plug-in is added to a Resource, which can later be used to serialize that instance. Within any EMF-based application, we can use an XMIResource to serialize or deserialize instance objects. In the sample application discussed in Chapter 7, “Implementing the sample” on page 203, we use XMIResources to contain WorkflowModel instances. To create or get each resource, we first create a ResourceSet, as Example 2-23 shows.

Example 2-23 Set up the ResourceSet

```
// Initialize the workflow package
WorkflowPackageImpl.init();

// Register the XMI resource factory for the .workflow extension
Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
Map m = reg.getExtensionToFactoryMap();
m.put("workflow", new XMIResourceFactoryImpl());

// Obtain a new resource set
ResourceSet resSet = new ResourceSetImpl();
```

ResourceSets are used to group related Resources. A Resource can contain instances of any object from a model, and a ResourceSet is used to group related Resources, that is, Resources that contain objects that reference each other. For the Workflow example, we use Resources to contain Workflow instances. We can use the ResourceSet created in Example 2-23 to create a new Resource as shown in Example 2-24.

Example 2-24 Create an XMIResource

```
// Create a resource
Resource resource =
   resSet.createResource(URI.createPlatformResourceURI(path.toString()));
```

If we want to load from an existing resource, we use the getResource() method instead, as shown in Example 2-25.

Example 2-25 Load an XMIResource

```
// Get a resource
Resource resource =
   resSet.getResource(URI.createPlatformResourceURI(path.toString()),true);
```

Once we have the resource, we can add objects to the contents of the resource. Objects contained by the same resource will be serialized to the same file. Example 2-26 shows how we create and add a Workflow object to a resource.

Example 2-26 Add a model object to a resource

```
Workflow workflow = wfFactory.createWorkflow();
resource.getContents().add(workflow);
```

Many models are based on an inheritance hierarchy, with a single top-level container. One of the advantages of this approach is that you need only add the top-level object to the resource explicitly. All of the other objects contained in the hierarchy will be serialized as elements within that top-level element. If you are using a model without a top-level container, then any objects that are not contained need to be added to the Resource explicitly.

2.3.3 Using the XSD plug-in to customize serialization

In this section, we demonstrate how to use a custom serialization to XML, by annotating our model with information used by the XSD plug-in.

Whenever we create an Ecore model from an XML Schema using the XSD plug-in, annotations that describe how each object is serialized to XML are added to the model, so that serialized model instances conform to the source schema. Use of these annotations is not limited to models imported from XML Schema; we show how to use the same annotations on any Ecore model to control how its instances are serialized.

Note: Although the XML produced by using techniques described in this section may look very similar to the XMI described in 2.3.2, “Default serialization of model instances” on page 66, it is important to notice that it does not conform to the XMI 2.0 standard.

We make the following changes to improve the readability of XML representing WorkflowModel instances:

- Use an XML element instead of an XML attribute to represent EAttributes that potentially have lengthy values, including:
  - comment from WorkflowElement
  - transformExpression from Transformation
  - condition from ConditionalOutputPort
  - whileCondition from LoopTask
- Use the singular form of the name of multi-valued containment EReferences to prevent the plural form being used for an elements that represent single objects. Notice that we do not make this change for non-containment EReferences, as the default serialization is to an XML attribute that represents the entire list of values, and so using the plural form of the name is appropriate.

- For example, within Workflow:
  - comments becomes comment
  - edges becomes edge
  - nodes becomes node
- Change the Workflow element to be lower case, to provide consistent capitalization throughout the document.

We begin by annotating the workflow EPackage, as shown in Example 2-27, indicating to the XSD plug-in that instances of this package use elements and attributes from the namespace `http://www.redbooks.ibm.com/sal330r/workflowXSD`. The annotations on the objects contained by the package indicate how each object maps to elements and attributes from this namespace.

Example 2-27 XSD annotation on workflow EPackage

```
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" nsPrefix="workflow"
   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="workflow"
   nsURI="http://www.redbooks.ibm.com/sal330r/workflowXSD">
   <eAnnotations source="http:///org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
      <details key="representation" value="schema"/>
      <details key="targetNamespace"
         value="http://www.redbooks.ibm.com/sal330r/workflowXSD"/>
   </eAnnotations>
   ... existing content ...
</ecore:EPackage>
```

Example 2-28 shows the `eAnnotations` element we use to annotate the comment EAttribute of WorkflowElement, to indicate that it should be represented as an element, rather than as an attribute. This is achieved by setting the value of the `representation` key to `element`. To force serialization as an attribute, we would use the value `attribute` instead. We add similar eAnnotations to the `eAttributes` elements representing transformExpression, condition and whileCondition so that they are also represented as XML elements.

Example 2-28 XSD annotation on comment EAttribute

```
<eAttributes name="comment" eType="ecore:EDataType
   http://www.eclipse.org/emf/2002/Ecore#//EString">
   <eAnnotations
      source="http:///org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
      <details key="representation" value="element"/>
      <details key="name" value="comment"/>
      <details key="targetNamespace"/>
   </eAnnotations>
</eAttributes>
```

We use a similar annotation to change the names of elements used to represent EReferences with pluralized names. For example, to use an element called node instead of nodes to represent each node contained by a Workflow, we add the EAnnotation shown in Example 2-29. We provide the new element name `node` as the value of the `name` key. We add similar annotations for all of the other multi-valued containment EReferences in our model.

Example 2-29 XSD annotation on nodes EReference

```
<eReferences name="nodes" eType="#//WorkflowNode" upperBound="-1"
   containment="true" eOpposite="#//WorkflowNode/workflow">
   <eAnnotations source="http:///org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
      <details key="representation" value="element"/>
      <details key="name" value="node"/>
      <details key="targetNamespace"/>
   </eAnnotations>
</eReferences>
```

We use the same technique to ensure that a lower-case element name is used for Workflow, however we have to be careful to make sure we specify the targetNamespace correctly, as the workflow element is the top-level element of our XML instance documents, so we cannot rely on XML namespace scoping for this value. Because we have constraints in the WorkflowModel that mean that all other objects are contained either directly or indirectly by a Workflow, we do not have to specify targetNamespace for any other elements, however, if you are using this technique to customize serialization for other models, make sure you specify this value for any elements that could appear as the top-level element in a serialized instance. Example 2-30 shows how we annotate the `eClassifiers` element representing the Workflow class. The targetNamspace that we specify in this annotation should match the nsURI of the containing package exactly.

Example 2-30 XSD annotation on Workflow EClass

```
<eClassifiers xsi:type="ecore:EClass" name="Workflow"
   eSuperTypes="#//WorkflowElement">
   <eAnnotations source="http:///org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
      <details key="representation" value="element"/>
      <details key="name" value="workflow"/>
      <details key="targetNamespace"
         value="http://www.redbooks.ibm.com/sal330r/workflowXSD"/>
   </eAnnotations>
   ... existing content ...
</eClassifiers>
```

Having annotated our model, we re-generate the model plug-in as follows:

1. Create or reload the GenModel from the annotated WorkflowModel, as described in “Java annotation and the code generation process” on page 24. To reload, select **Reload...** from the context menu that appears when you right-click the GenModel file in the Navigator or Package Explorer view, and then open the GenModel file.
2. We modify the GenModel so the regenerated code supports our customizations. Refer to 2.2.2, “Customizing code generation through GenModel properties” on page 47 for more information about setting properties in the GenModel. Select the workflow package from the GenModel tree and set Resource Type to `XML`.
3. Save the GenModel and then select **Generate Model Code** from the right-click context menu of the top-level element in the GenModel. You may wish to select **Generate All** instead if you do not have an up-to-date editor generated from your model. If adding these annotations is the only change that you have made to the model since generating the edit, and editor plug-ins, you do not need to regenerate them.

The model plug-in now includes code that supports serializing to our custom XML syntax. When we run our editor and create new model instances as described in Chapter 1, “Introduction to EMF” on page 3, the object instances are represented as elements or attributes according to the annotations that we added to the model. If we take a look at a new instance in a text editor, such as the instance shown in Example 2-31, and compare this to the default serialization shown in Example 2-21, we can see evidence of the changes that we have made to the serialization format.

Example 2-31 Custom serialization of a Workflow instance

```
<workflow:workflow xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:workflow="http://www.redbooks.ibm.com/sal330r/workflowXSD"
   id="w105966221898456">
   <node xsi:type="workflow:Task" name="Task 1" x="31" y="82"
      id="w105966222103157" isStart="true">
      <output xsi:type="workflow:FaultPort" name="fault"
         id="w105966226568760"/>
      <output name="output" id="w105966226568761" edges="w105966226568762"/>
      <input name="input" id="w105966226568763"/>
   </node>
   <node xsi:type="workflow:Task" name="Task 2" x="265" y="81"
      id="w105966222451558" isFinish="true">
      <output xsi:type="workflow:FaultPort" name="fault"
         id="w105966226568764"/>
      <output name="output" id="w105966226568765"/>
      <input name="input" id="w105966226568766" edges="w105966226568762"/>
   </node>
   <edge id="w105966226568762" target="w105966226568766"
```

```
      source="w105966226568761"/>
   <comment x="24" y="20" id="w105966223168759">
      <comment>This is a sample Workflow instance</comment>
   </comment>
</workflow:workflow>
```

You can find the completed annotated model in the WorkflowXSD folder in the examples for this section. This example demonstrates how to control whether model objects are serialized as XML elements or attributes, and allows us to provide names of our choosing for those elements and attributes. There are other annotations that are used by XSD to control feature-order and map XML Schema types to model objects, as discussed in 2.1.2, "Migrating existing models" on page 40, which you may also use to customize the serialization further.

2.3.4 Customizing XMI serialization using an XMLMap

When we customize the serialization using XSD annotations, we are using the XSD plug-in to generate an XMLMap that specifies the mapping between model objects and their serialization. We can perform similar customizations when we serialize without annotating the model.

The XMI 2.0 production rules allow features to be serialized either as elements or attributes. We can control whether each feature is serialized as an element or as an attribute by creating an XMLMap and adding appropriate mappings, as the following example illustrates. You can find the code for this example in the XMLMapExample directory in the examples for this section.

In this example, we change the serialization of the comment attribute of WorkflowElement, so that an XML element rather than an attribute is used to represent the value. We modify the execute() method within the WorkspaceModifyOperation in the doSave() method of the generated editor as shown in Example 2-32, to customize the serialization.

Example 2-32 Using an XMLMap to customize serialization of XMI

```
XMLMapImpl map = new XMLMapImpl();
XMLInfoImpl x = new XMLInfoImpl();
x.setXMLRepresentation(XMLInfoImpl.ELEMENT);
map.add(WorkflowPackage.eINSTANCE.getWorkflowElement_Comment(), x);
Map options = new HashMap();
options.put(XMLResource.OPTION_XML_MAP, map);
Resource savedResource =
   (Resource)editingDomain.getResourceSet().getResources().get(0);
savedResources.add(savedResource);
savedResource.save(options);
```

We perform the following steps to customize the serialization of a type or feature from the model:

1. Create an XMLMap to store the information about mapping the model to XML.
2. For each model object with a custom serialization:
   a. Create an XMLInfo and use the setName(), setTargetNamespace() and setXMLRepresentation() method to specify the how the object is represented in the XML.
   b. Add the XMLInfo to the XMLMap, using the object for which you want to customize serialization as the key. We do this in the example with:
   `map.add(WorkflowPackage.eINSTANCE.getWorkflowElement_Comment(), x);`
3. Put the XMLMap as the OPTION_XML_MAP in the options map that you use to save the resource.

Use the setName() method on the XMLInfoImpl to customize the name of the element or attribute tag used in the XML, and setTargetNamespace() to set the namespace for that element or attribute.

Use setXMLRepresentation() to specify whether the object is represented as an ELEMENT, ATTRIBUTE or CONTENT. Specifying CONTENT results in the value of object being contained directly by its parent. For example, we might use CONTENT to represent the condition attribute of a ConditionalOutputPort so that serialized instances look something like this:

```
<outputs xsi:type="workflow:ConditionalOutputPort" id="w1">
This is the condition
</outputs>
```

instead of looking like this:

```
<outputs xsi:type="workflow:ConditionalOutputPort" id="w1" condition="This
is the condition"/>
```

2.3.5 Providing a custom resource implementation

When we use the XSD plug-in to customize serialization, we are using an XMLResource to contain our model objects rather than an XMIResource. If we set the Resource Type property of a package to Basic, XMI, or XML in the GenModel, when we generate the model plug-in from the model, a ResourceImpl and ResourceFactoryImpl are generated for our model in the util package. By modifying the implementation of the ResourceImpl generated for our model, we can customize the serialization. If we have chosen XMI or XML as the Resource Type, the generated ResourceImpl will be a subclass of XMIResource or XMLResource, respectively. We can override methods in that subclass to customize serialization to XMI or XML. If we have chosen to use a Basic Resource Type, then we can serialize to any format by providing the necessary methods to implement ResourceImpl.

Customizing XMI serialization

When customizing XMI serialization, it is important to remember that if you customize the serialization format too much, it will no longer be standard XMI. However, there are some customizations that you can make without breaking conformance to the XMI 2.0 standard. One such customization is to use an element instead of an attribute to represent features, as we demonstrate in 2.3.4, “Customizing XMI serialization using an XMLMap” on page 74.

Another change that you can make while still complying with the standard is to modify how ids are generated in the serialization. Instead of using an attribute with the id property set to true in the model, you may wish to generate ids for all elements in the serialization. Although the ids are not accessible from the model, the advantage of generating them is that you can ensure that they remain unique. Be aware, however, that generating ids means that elements can potentially have a different id each time they are serialized.

The ids are mapped to objects from the model by the resource, which uses a map to map ids to each EObject. You can assign ids specifically to particular objects before you serialize by using the setID() method on the resource, as shown in Example 2-33.

Example 2-33   Set object ids via setID()

```
Resource resource = ...
EObject myobject = ....
resource.setId(myobject, “id1”);
```

If you want ids to be generated automatically for your objects, you can override the getID() in your resource implementation to do this, as Example 2-34 shows.

Example 2-34   Override getId() to generate ids

```
public String getId(EObject obj){

   String id = super.getID(obj);
   if (id == null){
      id = generateID();
      setID(obj,id);
   }
   return id;
}
```

Customizing XML serialization

The XML that we generated in 2.3.3, “Using the XSD plug-in to customize serialization” on page 70, used the names of references to contained objects for the XML elements representing those objects. A different representation would be to use a name that identifies the type of the reference, particularly in a model where there is usually only a single reference between objects of each type.

The mapping of element and attribute names to model objects is taken care of by an XMLHelper. We provide our own custom XMLHelper, to override the default names for references, replacing reference names with the name of the type instead. Example 2-35 shows how we override this method in our XMLHelper implementation.

Example 2-35 Customized XMLHelper

```
public class WorkflowXMLHelperImpl extends XMLHelperImpl implements XMLHelper {
   ...
   public String getName(ENamedElement obj) {
      XMLResource.XMLInfo info = null;
      if (xmlMap != null) {
         info = xmlMap.getInfo(obj);
      }
      if (info != null && info.getName() != null) {
         return info.getName();
      } else {
         if (obj instanceof EReference
         && ((EReference) obj).getEType() != null)
            return ((EReference) obj).getEType().getName();
         else
            return obj.getName();
      }
   }
}
```

Note: When there are multiple references to a type from the same object, we have to be careful, for example in the case of CompoundTask, because it has two references to Workflow, we have to be able to distinguish between the two, so we might need to add additional information to the serialization to do this. Generally you would only want to use a serialization such as this one if the type implied the reference.

To use the XMLHelper from our XMLResourceImpl, we simply override the method that creates the helper, to create an instance of our WorkflowXMLHelper instead, as Example 2-36 shows.

Example 2-36 Overriding the createXMLHelper() method

```
protected XMLHelper createXMLHelper()
{
  return new WorkflowXMLHelperImpl(this);
}
```

The file XMLResourceCustomization.workflow contains an example of a Workflow serialized using WorkflowXMLResource.

So far we have only dealt with serializing to the custom format, we would also have to override the getFeatureWithoutMap() method to map the types back to features, however we leave this an exercise for the reader.

Customizing XMLHelper allows us to use the names of the types of the references for element names, however because XMLHelper is creating names from the model itself, rather than from instances, it cannot create specific type names for subtypes. An example of using more specific type names for the WorkflowModel would be to use element names FaultPort or ConditionalOutputPort instead of just using OutputPort for those types, or to use Task, Comment or Choice instead of WorkflowNode, such as the fragment shown in Example 2-37.

Example 2-37 A more readable representation of contained objects

```
<Task name="Task 2" x="265" y="81" id="w105966222451558" isFinish="true">
      <FaultPort name="fault" id="w105966226568764"/>
      <OutputPort name="output" id="w105966226568765"/>
   ...
</Task>
```

The serialization would produce such elements if you added explicit references to each class with specific names for each reference, in the same way that we already have specific references to OutputPort and InputPort rather than a general reference to Port.

However, then you would need to maintain all of these references separately or lose the benefits of polymorphism, and your model would be cluttered. However, we could implement such a serialization by providing our own subclasses of XMLSaveImpl and XMLLoadImpl and use them within WorkflowResourceImpl, as these are the classes that actually serialize our instances, and override methods such as saveElement() to provide a more specific name.

These examples are provided to give you an idea of the types of things you can customize by providing your own XMLHelper, XMLSave or XMLLoad implementations. You may choose to override the methods from those classes to produce any XML serialization.

Other serializations

To serialize to other formats, all you need to do is to implement your own versions of the doSave() and doLoad() methods in your ResourceImpl subclass.

2.4 Using JET to customize code generation

In this section we provide examples that illustrate how to use the Java Emitter Templates (JET) framework provided with EMF to customize code generation. We describe how JET is used to generate the model, edit, and editor plug-ins that we examine in "The generated plug-ins" on page 45, as well as how to approach customizing this code generation.

For an introduction to JET in general, refer to the two-part JET Tutorial by Remko Popma, available from Eclipse Corner, at:

http://eclipse.org/articles/Article-JET/jet_tutorial1.html
http://eclipse.org/articles/Article-JET/jet_tutorial2.html

2.4.1 .JET-related GenModel properties

In 2.2.1, "The generated plug-ins" on page 45, we described the model, edit, and editor plug-ins that are generated from EMF models. These plug-ins are generated using JET, and we can control this generation by setting the GenModel properties of the model from which we are generating the plug-ins.

The JET-related GenModel properties are described in Table 2-5. All of these properties are represented at the top-level of the GenModel, and are grouped by the GenModel editor into the Templates & Merge category. Setting these properties allows us to override the default JET templates used to generate the model, edit, and editor plug-ins. Descriptions of the properties are provided by the GenModel editor in the status bar whenever you select one of the properties.

Table 2-5 Templates and Merge GenModel properties

Property	XMI attribute	Default
Dynamic Templates	dynamicTemplates	false
Force Overwrite	forceOverwrite	false
Redirection Pattern	redirection	
Runtime Jar	runtimeJar	false
Template Directory	templateDirectory	
Update Classpath	updateClasspath	true

The most interesting of these to us are the dynamicTemplates and template Directory properties:

The dynamicTemplates property indicates that the precompiled templates provided by org.eclipse.emf.codegen.ecore.genmodel should be ignored, and that the template implementation should be translated and compiled from dynamic templates.

The templateDirectory indicates the location to look for new templates. A template placed in this location will override the default template with the same name from org.eclipse.emf.codegen.ecore.genmodel.

2.4.2 Writing JET templates

In this section, we customize the generation of the plug-ins described in 2.2.1, "The generated plug-ins" on page 45. By default, these plug-ins are generated from templates located in:

<ECLIPSEHOME>/plugins/org.eclipse.emf.codegen.ecore_<EMFVERSION>/templates

Where <ECLIPSEHOME> is the location where you installed Eclipse and <EMFVERSION> is the version of the EMF plug-in that you have installed.

The org.eclipse.emf.codegen.ecore templates directory contains sub-directories for the model, edit, and editor plug-ins. The files with the extension javajet are the templates. The file extension follows the JET convention of using the extension of the file that is generated by the template concatenated with jet. In this example, we customize the Java code generated for the model plug-in by providing our own version of some of the model templates.

The header template provides the comment that is located at the head of each generated class file. We begin by creating our own templates directory, and by supplying a new Header.javajet. To do this, perform the following steps:

1. Add a directory called templates to the WorkflowModel project.
2. Create a new text file called Header.javajet in the templates directory. If you prefer, you can copy the existing Header.javajet file as a basis for your template.
3. Edit Header.javajet to contain the comment that is to be included at the top of every generated class file. We edit the file to read as shown in Example 2-38:

Example 2-38 Our version of Header.javajet

```
/**
 * WorkflowModel
 *
 * Copyright (c) 2000, 2003 IBM Corporation and others.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Common Public License v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/cpl-v10.html
 *
 */
```

4. Generate the GenModel for the WorkflowModel. This step may be skipped if you already have a GenModel for the WorkflowModel.
5. Edit the GenModel properties:
   a. Set the dynamicTemplates property to true.
   b. Set the templateDirectory property to the location of your templates directory, for example, `/WorkflowModel/templates`.

By default, the header is only generated the first time the code is generated from your model, so if you already have a version of the model plug-in in your project, you will need to override this behavior. The merging of existing content with new content is handled by EMF's jmerge. The rules for merging the model, edit, and editor code generated from EMF models are expressed in the file emf-merge.xml. Copy emf-merge.xml into your templates directory from the org.eclipse.emf.ecore.codegen plug-in's templates directory and modify the file so that it includes an additional rule to set the header each time the code is generated, as shown in Example 2-39.

Example 2-39   Merge rules for code generation from WorkflowModel

```
<merge:options ... >
   ... existing content ...
   <merge:pull sourceGet="CompilationUnit/getHeader"
      targetPut="CompilationUnit/setHeader"/>
</merge:options>
```

Now when you generate the model plug-in and take a look at the generated code, the contents of Header.javajet should appear in place of the default header.

Tip: Whenever you modify a template, you may need to close and then re-open the GenModel file before regenerating code so that the new version of the template is used.

JET templates use a simplified Java Server Pages (JSP) syntax. You can get a feel for how JET templates work by examining and modifying the templates used to generate the interface and implementation corresponding to each class in a model. Begin by making a copy of the templates into the WorkflowModel project's templates directory:

1. Create a model sub-directory within the templates directory in the WorkflowModel project.
2. Copy the files Interface.javajet Class.javajet and from the model directory in the org.eclipse.emf.codegen.ecore plug-in's templates directory to the directory created in the previous step.

We are mirroring the templates directory structure used by the org.eclipse.emf.codegen.ecore plug-in, as we are essentially replacing its templates with our own versions.

At the first line in Interface.javajet, we see the tag shown in Example 2-40.

Example 2-40   The jet directive

```
<%@ jet package="org.eclipse.emf.codegen.ecore.templates.model"
imports="java.util.* org.eclipse.emf.codegen.ecore.genmodel.*"
class="Interface" %>
```

The tags used within JET templates are identified by an opening `<%` and a closing `%>`. Inside the tags, you can use Java code to script what is generated from the template, or you can use special tags to represent JET directives or expressions. The `jet` tag shown in Example 2-40 is a directive. Expression tags are used to create values based on expressions in the files generated from the templates.

Directives start with `<%@` and a name that identifies them, and expressions start with `<%=`. We can see examples of each of these types of tags just a few lines further down in Interface.javajet, as shown in Example 2-41.

Example 2-41 JET scriptlet, directive and expression tags

```
<%GenClass genClass = (GenClass)argument; GenPackage genPackage =
genClass.getGenPackage(); GenModel genModel=genPackage.getGenModel();%>
<%@ include file="../Header.javajet"%>
package <%=genPackage.getInterfacePackageName()%>;
```

The first tag shown in Example 2-41, is a scriptlet that declares and initializes variables that can be referenced from other tags in the rest of the template. In this case, we see `genClass`, which represents the class for which the interface is being generated using this template, `genPackage`, which represents its containing package, and `genModel`, which is the model that contains `genPackage`.

The second tag shown in Example 2-41 is another directive; this one indicates that the code produced from the Header.javajet template is included at this point in the code generated from this Interface template. The `include` directive has a single attribute, `file`, that indicates the location of the file to be included. There are two directives that can be used within JET templates; the `include` directive, seen in this example, and the `jet` directive seen in Example 2-40. The `jet` directive may appear only on the first line of a template, and every template must have a `jet` directive. The attributes of the `jet` directive are described in the JET Tutorial part one, (Introduction to JET). Note Header.javajet did not have a `jet` directive, because it is just a fragment included into other templates.

The third tag from Example 2-41 is an expression tag, which in this case provides the expression used to get the package name for the interface that is generated using the template.

You may notice that the names of most of the types and methods that end up in the generated code come from expression tags that call methods provided by the GenModel. The reason for this is that the code is generated from GenModel objects that are provided as arguments to each template. We can change the structure or the literal content of the generated code by editing the templates, however changing the names of the methods and types in the generated code would require providing our own implementation of the interfaces in org.eclipse.emf.codegen.ecore.genmodel and then providing those objects as arguments to the templates. For our additions to the generated code, we edit the templates only. If you would like to find out more about providing different objects as arguments to a JET template, please refer to the JET Tutorial.

We modify the templates to add additional methods for multi-valued features to get an element from the list of values by position. Because WorkflowModelElement is the supertype of every other class in the WorkflowModel, and it has an name attribute, we know that every object in the model can have a name, hence we also add template methods to get list members by name.

In Interface.javajet, we can see that the section of the template that generates accessor methods for features is contained within a for-loop that iterates over the features of the class. We are adding template to generate additional methods for some features, so we make our additions within this loop.

Example 2-42 shows concrete examples of the method signatures that we are adding to the generated interfaces. In this case, the methods are for the inputs feature of the class WorkflowNode.

Example 2-42 Concrete example of additional method signatures

```
InputPort getInputs(int index);
InputPort getInputs(String name);
```

To generate similar methods for all multi-valued features using the templates, we substitute types and methods specific to the inputs reference with expression tags. We use method calls on `genFeature`, which represents each feature, to provide the values. We also add `@generated` to indicate that these methods are now generated. Example 2-43 shows the code that we add to Interface.javajet to generate the extra methods in the interface for each generated type. We wrap the template for the new methods inside conditions, to make sure that we only generate these methods for multi-valued features, that is, features that are regular list types.

Example 2-43 Interface template fragment for additional methods

```
<%for (Iterator i=genClass.getGenFeatures().iterator(); i.hasNext();)
{GenFeature genFeature = (GenFeature)i.next();%>
   ... existing content ...
<%if (genFeature.isListType()) {%>
<%if (!genFeature.isMapType()){%>
   /**
    * Get an item from the list by position
    * @generated
    */
   <%=genFeature.getQualifiedListItemType()%>
<%=genFeature.getGetAccessor()%>(int index);
   /**
    * Get an item from the list by name
    * @generated
    */
   <%=genFeature.getQualifiedListItemType()%>
<%=genFeature.getGetAccessor()%>(String name);
<%}//if%>
<%}//if%>
<%}//for%>
```

We use a similar process to template the implementation of the additional methods in Class.javajet. Example 2-44 shows concrete examples of the implementation of the methods that we wish to add.

Example 2-44 Concrete example of additional methods

```
public InputPort getInputs(int index) {
   return (InputPort) this.getInputs().get(index);
}

public InputPort getInputs(String name) {
   Iterator i = this.getInputs().iterator();
   while (i.hasNext()) {
      InputPort input = (InputPort) i.next();
      if (true == name.equals(input.getName()))
         return input;
   }
   return null;
}
```

Again, we generalize by substituting expression tags for the parts of the method implementations that are specific to the feature. Example 2-45 shows the code that we add to Class.javajet. We add the method templates to the existing for-loop that iterates over all of the implemented features. Notice that because we introduce the class java.util.Iterator into the generated code in the second additional method, we need to use the `getImportedName` method from the GenModel to make sure it is added to the imports in the generated class.

Example 2-45 Class template fragment for additional methods

```
<%for (Iterator i=genClass.getImplementedGenFeatures().iterator();
i.hasNext();) {GenFeature genFeature = (GenFeature)i.next();%>
   ... existing content ...
<%if (genFeature.isListType()) {%>
<%if (!genFeature.isMapType()){%>
   /**
    * Get an item from the list by index
    * @generated
    */
   public <%=genFeature.getQualifiedListItemType()%>
<%=genFeature.getGetAccessor()%>(int index){
      return (<%=genFeature.getQualifiedListItemType()%>)
this.<%=genFeature.getGetAccessor()%>().get(index);
   }

   /**
    * Get an item from the list by name
    * @generated
    */
   public <%=genFeature.getQualifiedListItemType()%>
<%=genFeature.getGetAccessor()%>(String name){
      <%=genModel.getImportedName("java.util.Iterator")%> i =
this.<%=genFeature.getGetAccessor()%>().iterator();
      while (i.hasNext()) {
         <%=genFeature.getQualifiedListItemType()%> l =
(<%=genFeature.getQualifiedListItemType()%>) i.next();
         if (name.equals(l.getName()))
            return l;
      }
      return null;
   }
<%}//if%>
<%}//if%>
<%}//for%>
```

Now, when you generate the model plug-in, you should see the additional methods in the generated interfaces and implementation classes. You may also notice a new project .JETEmitters appear in your workspace in the resource view. This project is created by default when the templates are translated, as described in the JET Tutorial, Part Two, and it contains the actual implementations of our templates.

3

Introduction to GEF

In this chapter, we provide an introduction to GEF and Draw2D. You can read about the basics of the frameworks and get some first tips about using them.

After the introduction, we show you how to build a graphical editor skeleton using our step-by-step instructions, and then explain how to map your model into GEF edit parts.

Note: The sample code we describe in this chapter is available as part of the redbook additional material. See Appendix A, “Additional material” on page 225 for details on how to obtain and work with the additional material. The sample code for this chapter is provided as Eclipse projects that can be imported into your Eclipse workbench. Each major section of this chapter has a matching Eclipse project in the additional material.

3.1 What is the Graphical Editing Framework?

The Graphical Editing Framework allows us to easily develop graphical representations for existing models. It is possible to develop feature rich graphical editors using GEF.

All graphical visualization is done via the Draw2D framework, which is a standard 2D drawing framework based on SWT from eclipse.org.

The editing possibilities of the Graphical Editing Framework allow you to build graphical editors for nearly every model. With these editors, it is possible to do simple modifications to your model, like changing element properties or complex operations like changing the structure of your model in different ways at the same time.

All these modifications to your model can be handled in a graphical editor using very common functions like drag and drop, copy and paste, and actions invoked from menus or toolbars.

For our demonstration code and for explanations of the GEF API, we used the latest code releases that were available during the creation of this redbook: Eclipse 2.1.1 and GEF 2.1.1.

3.1.1 Additional documents and resources

Basically there are two kinds of additional resources available — one that ships with the Graphical Editing Framework and other freely available on the Internet.

Integrated Eclipse help

The Graphical Editing Framework SDK provides online help that is integrated into Eclipse. This should be used as a starting point. It is available by clicking **Help -> Help Contents** and then clicking the topic **Draw2D Developers Guide** or **GEF Developer Guide** on the left side of the new window.

Note: Only the GEF SDK is shipped with the developer documentation of GEF and Draw2D.

Resources on the Web

The GEF Web site provides access to a wide range of resources related to the Graphical Editing Framework, including code releases, examples, and documentation:

http://www.eclipse.org/gef

Any questions and topics not answered by the frequently asked questions (FAQ) feature, available at the GEF Web site, can be discussed in the GEF newsgroup (`eclipse.tools.gef`), which is available at the Eclipse news server (`news.eclipse.org`).

A public community driven pool is available at:

`http://eclipsewiki.swiki.net`

The Eclipse Wiki also has a section for GEF related topics, which provides an additional list of answers for frequently asked questions and additional examples and other resources.

3.1.2 Applications suitable for GEF

We found numerous applications developed with GEF. Thanks to the authors of these applications, we are able to show you the following screen captures of sample applications using GEF. As you will see, there is no limit on using a graphical editor for nearly every case.

The most common case might be a modelling application. You can build graphical editors for modelling nearly every kind of situation (for example, business processes, application models, or even UI screens).

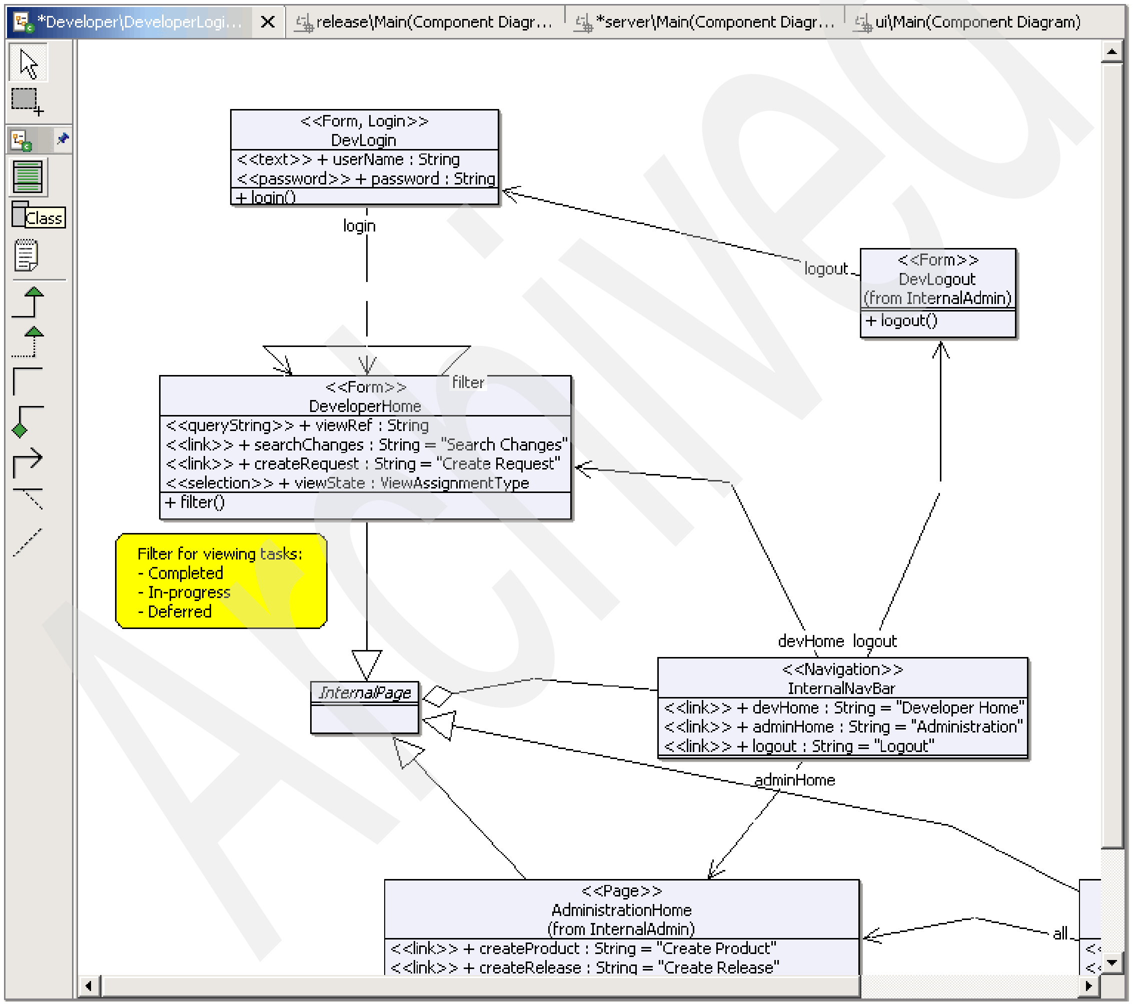
There are also graphical editors available for designing documents such as reports, Web sites, or forms. You can develop graphical editors for modifying environments (for example, configuration files of applications, servers, or deployment descriptors for enterprise applications — or even for routing trains).

The possibilities are only limited by your imagination!

MDE for Struts

Available as an Eclipse-based IDE or plug-in, MDE for Struts (Figure 3-1) enables model-driven development of Struts 1.1 applications using standard UML. From simple class diagrams, MDE for Struts creates JSPs, Java classes, struts-config.xml, validator.xml, Application Resource, ANT build scripts and J2EE deployment files. You can take control of the architecture by changing Java MetaPrograms that translate the model to code. A free evaluation version is available at:

http://www.metanology.com



*Developer\DeveloperLogi...
release\Main(Component Diagr...
*server\Main(Component Diagr...
ui\Main(Component Diagram)
Class
<<Form, Login>>
DevLogin
<<text>> + userName : String
<<password>> + password : String
+ login()
login
logout
<<Form>>
DevLogout
(from InternalAdmin)
+ logout()
filter
<<Form>>
DeveloperHome
<<queryString>> + viewRef : String
<<link>> + searchChanges : String = "Search Changes"
<<link>> + createRequest : String = "Create Request"
<<selection>> + viewState : ViewAssignmentType
+ filter()
Filter for viewing tasks:
- Completed
- In-progress
- Deferred
devHome
logout
InternalPage
<<Navigation>>
InternalNavBar
<<link>> + devHome : String = "Developer Home"
<<link>> + adminHome : String = "Administration"
<<link>> + logout : String = "Logout"
adminHome
<<Page>>
AdministrationHome
(from InternalAdmin)
<<link>> + createProduct : String = "Create Product"
<<link>> + createRelease : String = "Create Release"
all

Figure 3-1 Struts MDE

AcmeStudio

AcmeStudio (Figure 3-2) is a customizable editing environment and visualization tool for software architecture designs based on the Acme architectural description language (ADL). With AcmeStudio, you can define new Acme families for specific domains and customize the environment to work with those families by defining new diagram styles. AcmeStudio is an adaptable front-end that may be used in a variety of modeling and analysis applications. Written as an Eclipse plug-in, AcmeStudio provides the opportunity to integrate third-party architectural analysis tools.

AcmeStudio is being developed at the School of Computer Science at Carnegie Mellon University. This work is supported in part by DARPA under Grants N66001-99-2-8918 and F30602-00-2-0616, and by the High Dependability Computing Program from NASA Ames, cooperative agreement NCC-2-11298.

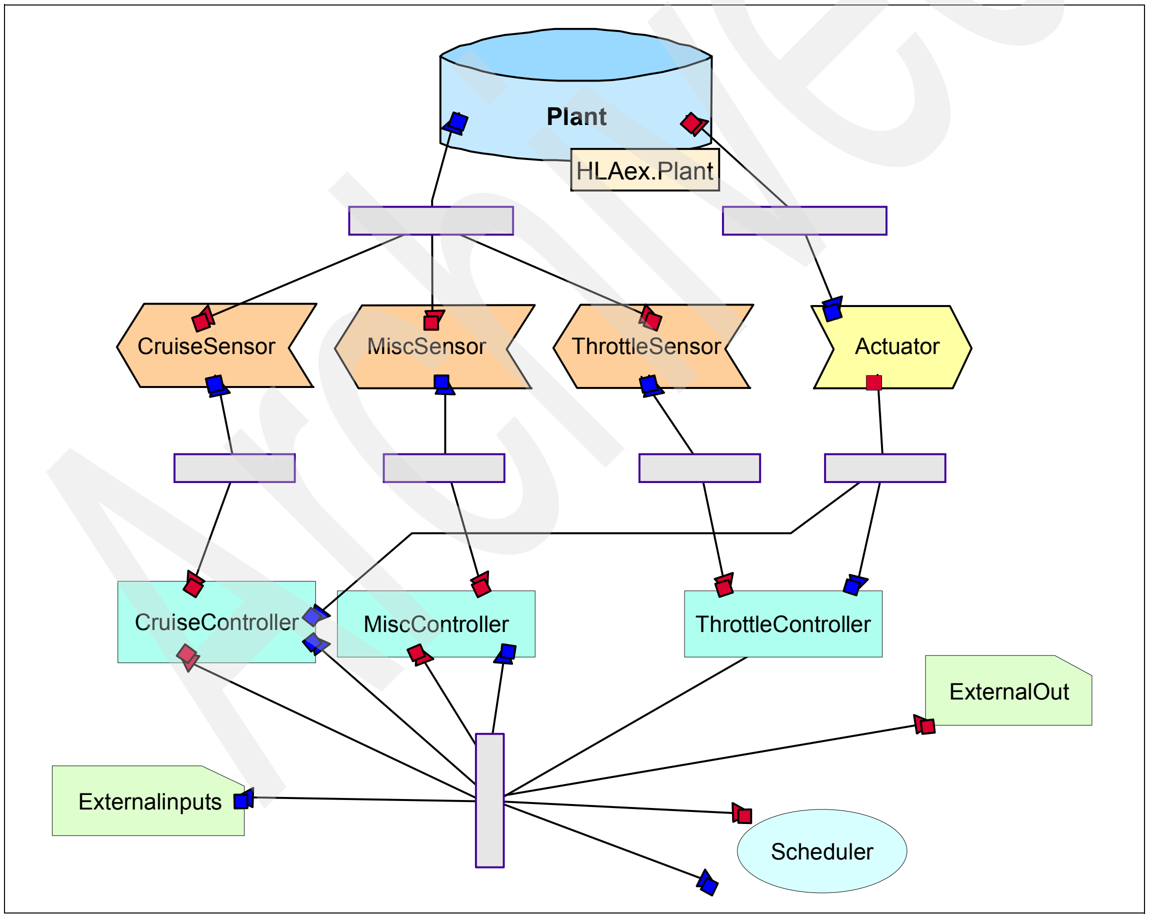
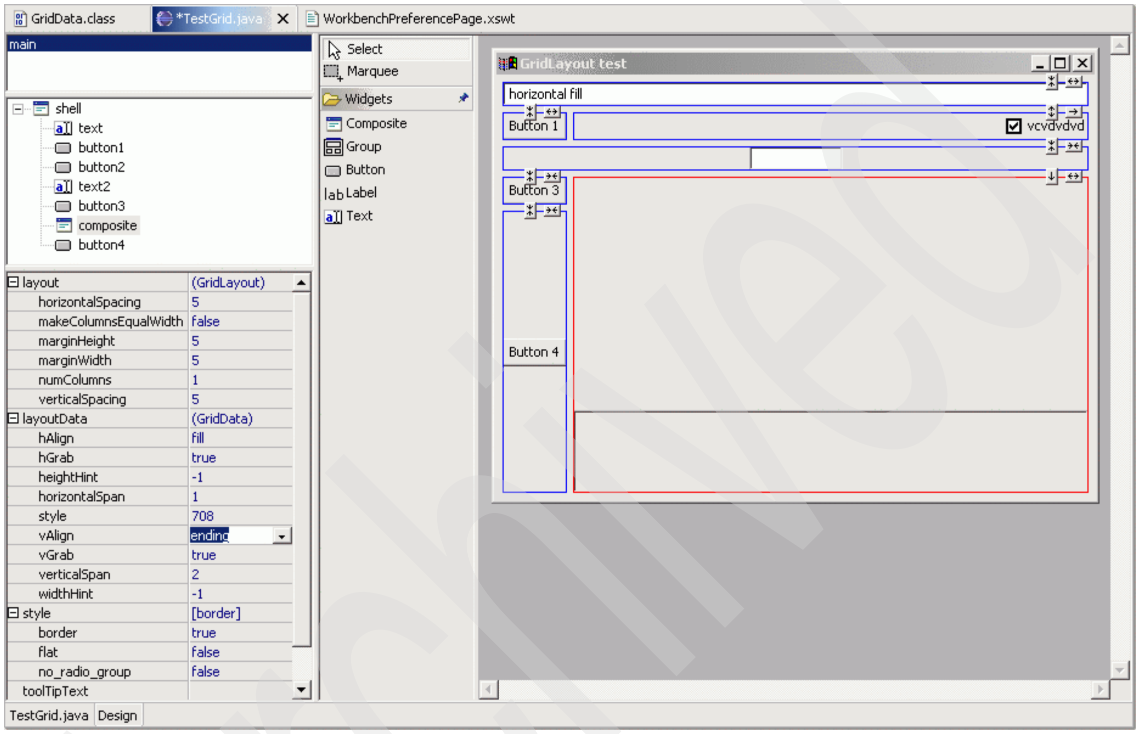


Figure 3-2 AcmeStudio

EclipseDesigner

EclipseDesigner (Figure 3-3) is a two-way visual designer for SWT. Design editing can be done in Java editor or visually on a design page using property tables and mouse manipulations in a GEF editor. It is freely available at:

```
http://eclipsedesigner.sourceforge.net
```



layout	(GridLayout)
horizontalSpacing	5
makeColumnsEqualWidth	false
marginHeight	5
marginWidth	5
numColumns	1
verticalSpacing	5
layoutData	(GridData)
hAlign	fill
hGrab	true
heightHint	-1
horizontalSpan	1
style	708
vAlign	ending
vGrab	true
verticalSpan	2
widthHint	-1
style	[border]
border	true
flat	false
no_radio_group	false
toolTipText	

Figure 3-3 EclipseDesigner

Jeez Report Designer

Jeez Report Designer (Figure 3-4) allows visual designing of reports that can be executed using a report engine. It is freely available at:

http://jeez.sourceforge.net

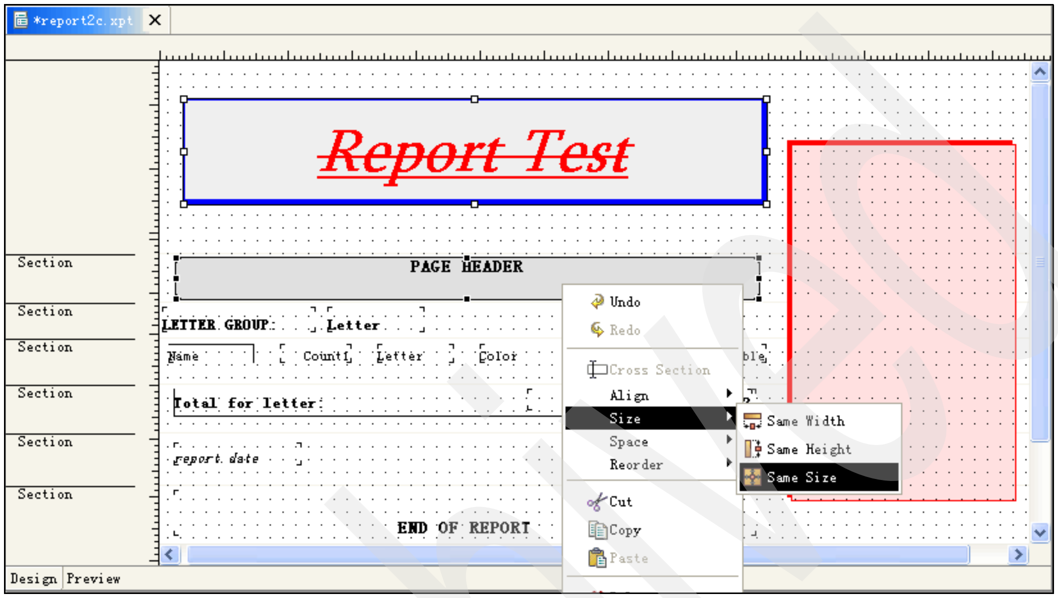


Figure 3-4 Jeez Report Designer

3.2 Introduction to Draw2D

Draw2D provides the lightweight graphical system that GEF depends on for its display. It is packaged in Eclipse as a separate plug-in, org.eclipse.draw2d. Draw2D is hosted in a SWT canvas heavyweight control and manages the painting and mouse events that occur in the host canvas by delegating them to Draw2D figures. Figures are analogous to windows in a heavyweight graphics system. They can have arbitrary, nonrectangular shapes and can be nested in order to compose complex scenes or custom controls.

Figures can be transparent or opaque, and can be ordered into layers, thus allowing parts of a diagram to be hidden or excluded from certain operations. Draw2D is a standalone graphics library that can be used by itself to create graphical views in Eclipse. A complete coverage of Draw2D in depth is beyond the scope of this book. Instead, we discuss some key Draw2D concepts and focus on the Draw2D features and classes that are most important to GEF developers.

3.2.1 What is a lightweight system?

A lightweight system is a graphics systems that is hosted inside a single heavyweight control. The graphics objects in the lightweight system, known as figures in Draw2D, are treated as if they are normal windows. They can have focus and selection, get mouse events, have their own coordinate system, and have a cursor. They each get a graphics context for rendering. The advantage of lightweight systems is that they are much more flexible than the native windowing system, which is generally composed of rectangular components. They allow you to create and manipulate arbitrarily shaped graphics objects. Because they simulate a heavyweight graphics system within a single heavyweight window, they allow you to create a graphically complex display without consuming a lot of system resources.

3.2.2 Architectural overview

As we said earlier, Draw2D is a self-contained graphics library and can be used independently of GEF or even of Eclipse. You can see the basic structure of a standalone Draw2D application in Example 3-1.

Example 3-1 A standalone Draw2D application

```
Shell shell = new Shell();
shell.open();
shell.setText("A Draw2d application");
LightweightSystem lws = new LightweightSystem(shell);

// add your application’s root figure
IFigure panel = new Figure();
panel.setLayoutManager(new FlowLayout());
lws.setContents(panel);
...

// add your application’s figures here
panel.add(...);

while (!shell.isDisposed ()) {
if (!display.readAndDispatch ())
   display.sleep ();
}
```

Tip: When you create a standalone Draw2D application, you need to make sure that your operating system is able to locate the SWT native library. For instance, on Microsoft Windows, make sure that the following file is added to your class path:
ECLIPSE_HOME\plugins\org.eclipse.swt.win32_2.1.1\os\win32\x86swt32.dll

In Example 3-1 on page 94, the shell serves as the SWT canvas that is needed to host Draw2D. The LightweightSystem class provides the bridge between SWT and Draw2D by routing mouse and paint events to the figures it contains. A root figure is then added to the LightweightSystem. The root figure is configured with a layout manager which controls the layout of any child figures that are subsequently added to it.

3.2.3 Figures

In this section we go into more detail on the capabilities of figures in general, and introduce some of the specialized figures that Draw2D provides.

Methods

Everything that is visible in a Draw2D window is drawn on a figure. The figure class contains a number of methods that provide the following functionality:

- Registering or deregistering listeners on a figure; the figure will notify listeners of mouse events within the figure
- Structural events, for structural changes in the figures hierarchy, and for movement or resizing of the figure
- Specifying the cursor to display when the mouse passes over it
- Operations to manage the figure's place in the figure hierarchy, including adding and removing children and accessing them or its parent figure
- Accessors for:
  - The figure's layout manager
  - The figure’s size and location
  - The tooltips
- Setting and getting focus
- Specifying the figure's transparency and visibility
- Performing coordinate conversion, intersection, and hit testing
- Painting
- Validating

Subclasses

Draw2D provides many subclasses of figure that provide useful additional functionality. We describe some of these in the following sections.

Shapes

Subclasses of the Shape class contain non-rectangular figures that know how to fill themselves and provide a border of configurable width and line style, and include support for XOR drawing. Some examples are the Ellipse, Polyline, Polygon, Rectangle, Rounded rectangle, and Triangle classes.

Widgets

Draw2D includes figures which allow you to create lightweight widgets that can be used when you need an input control within your Draw2D application. These include various buttons, Checkbox, and the text entry figure, Label.

Layers and panes

These are figures designed to host child figures. They providing scaling, scrolling, and the ability to place figures into different layers.

The graphics context

Figures have a paint method that is called by the LightweightSystem when the figure needs to be rendered. Each figure gets a graphical context, an instance of the Graphics class, that is passed as argument to the figure's paint method. The graphics context supports graphics operations, including drawing and filling shapes and drawing text. It also maintains the graphics state that influences these operations, such as the current font, background and foreground colors, etc. This analogous to many other graphics systems.

3.2.4 Mechanism

This section introduces the core classes of the Draw2D architecture.

LightweightSystem

The LightweightSystem class is the heart of Draw2D. It performs the mapping between an SWT canvas and the Draw2D system that is hosted within it. It contains three main components:

- **The root figure:** This is an instance of the LightweightSystem$RootFigure class; this top level figure is the parent of your application's root figure. It inherits some of the graphical environment of the hosting SWT Canvas, such as font, background, and foreground colors.
- **The event dispatcher:** The SWTEventDispatcher class translates SWT events to the corresponding Draw2D events in the root figure. It tracks which figure has focus, which figure is being targeted by mouse events, and handles tooltip activation. It provides support for figures that want to capture the mouse.

- **The update manager:** The update manager is responsible for painting and updating Draw2D figures. The LightweightSystem calls the update manager's performUpdate() method when a paint request is received from the underlying SWT canvas. The update manager typically maintains a worklist of figures that are invalid or need repainting. The update manager tries to coalesce its work lists so that it can be as efficient as possible. The default update manager, DeferredUpdateManager, allows updates to be performed asynchronously by queuing work on the Display's user interface thread.

The main processes in a figure's life cycle are painting and validating. Draw2D asks a figure to render itself by calling the figures paint methods. The paint() method invokes three more specific paint methods:

- paintfigure() — The figure should render itself.
- paintclientarea() — The figure should render its children.
- paintborder() — The figure should render its border, if any. Clipping is used to protect the border.

Validating occurs when a figure's size or location needs to be calculated:

- validate — Asks the figure's layout manager to re-layout its children.
- revalidate — Calls invalidate; adds a figure and its predecessors to the update manager's invalid list.

3.2.5 Major features

In the following sections we summarize some of the main features and functionality provided by Draw2D.

Borders

It is frequently necessary to provide a visual border to figures. The Draw2D package contains several classes, derived from the Border class, which provide a variety of border effects:

- GroupBoxBorder — Creates a labeled border similar to group boxes in native window systems
- TitleBarBorder — Creates a titled border that resembles a titled window
- CompoundBorder — A border composed of two borders
- FrameBorder — Similar to TitleBarBorder; can be used to create figures with titles
- FocusBorder — Surrounds a figure with a focus rectangle
- LineBorder — Creates an outline around a figure of the width you specify

- MarginBorder — A border for creating padding around the edges of a figure
- SchemeBorder — A base class for borders whose borders simulate shadows and highlights
- ButtonBorder — Used with a Clickable figure to create lightweight button-like controls
- SimpleLoweredBorder and SimpleRaisedBorder

Examples of some of these border types are illustrated in Figure 3-5.

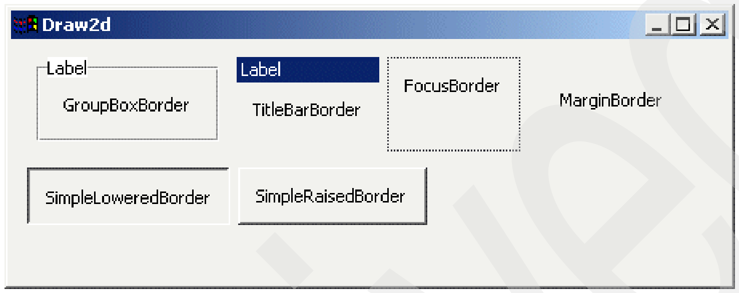


Figure 3-5 Some Draw2D border types

The Insets class is used to represent the space within a figure that is allocated to the border. Notice that the border does not have to be symmetrical. It can be occupy any combination of a figure's edges, and can be a different size on any edge. The paint method clips the client area so that painting is constrained to the area of the figure inside the inset. The border, when present, is the last part of the figure to be painted.

Layouts

LayoutManagers are used to manage the position and size of a figure's child figures. They interrogate each child figure to obtain its preferred size, and then apply some layout algorithm to calculate the final size and placement of the child figures. LayoutManagers also support constraints, which are data attached to each figure that gives additional guidance to the layout manager. The figure has accessor methods for its constraints, and the layout manager maintains a map of constraints for the figures it is managing.

The constraint accessors use the Object type for constraints since the type of the constraint depends on the layout manager being used. For instance, the XYLayout layout manager requires that the figures it manages have a constraint of type Rectangle, and the DelegatingLayout manager expects its figures to have a constraint which implements the Locator interface. All Draw2D layout managers derive from the AbstractLayout abstract class.

These are some of the provided layout managers:

- FlowLayout — Lays out its children into either rows or columns, which is configurable either by using the constructor:

  ```
  public FlowLayout(boolean isHorizontal)
  ```

  or by calling the method:

  ```
  setHorizontal(isHorizontal)
  ```

  The manager causes its children to wrap when the current row or column is full. There are also methods to control the alignment and spacing of rows in both the major and minor axes.
- DelegatingLayout — Delegates the layout of its child figures to the child figures' locators. The children must provide a Locator subclass as their constraint.
- XYLayout — Places its children at the location and dimensions specified for the child. The child's constraint must be a Rectangle object that specifies this information.

Draw2D provides a scrollable pane via the ScrollPane class. To implement this functionality it uses the ScrollPaneLayout, which manages the layout of the scroll bars and Viewport that comprise the ScrollPane. In addition the Viewport uses the ViewportLayout manager to manage the viewport's visible region and maintain the scroll position state.

Layers

Layers are transparent figures intended specifically for use in LayerPanes. They override the figure's containsPoint() and findFigureAt() methods so that hit testing will "pass through" the layer. The FreeformLayer class adds additional specialization to Layer to provide a layer that can extend indefinitely in all four directions.

Note: The term "Freeform", when used in Draw2D class names, indicates that the class supports figures that can expand in all directions — that is, they do not have a fixed size or origin, which also implies that the child figures can have negative coordinates. Some examples are the FreeformLayer, FreeformLayeredPane, and ScalableFreeformLayeredPane classes.

The ConnectionLayer class implements a FreeformLayer that is designed to contain connections. It Insures that any Connection figures added to the layer will have their connection router set correctly to the layer's connection router. Similarly, when the layer's connection router is changed, it will update the connection router of all its connection figures.

LayerPanes are figures designed to contain layers (they can only contain layers). The layers in a LayerPane are stored in a map whose key is typically a String. LayerPanes contain methods to add, insert, remove, and reorder the layers they contain.

Two subclasses of LayerPane provide additional flexibility. The FreeformLayeredPane provides a set of layers that can expand in all directions. The ScalableFreeformLayeredPane adds support for zooming.

Finally, the ScalableLayeredPane provides a LayerPane that is scalable but is not free form but instead has a finite, fixed size.

Locators

Implementors of the Locator interface are used in Draw2D to position figures. The interface consists of a single method:

```
void relocate(IFigure target)
```

Subclasses of ConnectionLocator are used for locating figures that are attached to a Connection. These can be used for placing arrowheads on the ends of connections or placing labels or other decorations or annotations on a Connection. The locator ensures that the figure stays “attached” to the Connection in the designated location as the Connection is moved.

Available locators include:

- ArrowLocator — This locator is used to position decorations, such as arrowheads, on the ends of connections. Any figure that implements RotatableDecoration can be located. Implementors of RotatableDecoration are given a position and a reference point so that they can rotate their visual representation based on the angle of the connection they are decorating.
- BendpointLocator — This locator is used to position bendpoint handles on a Connection
- MidpointLocator — This locator is used to place figures at the midpoint of a Connection
- ConnectionEndpointLocator — This locator is used to locate a figure near either the start or end of a connection.
- RelativeLocator — This locator is used to locate a figure using a 0 to 1 floating point value representing its affinity for the a weighting of the figure's affinity for the upper left corner (0) or lower right corner (1) of a reference figure. This class is generally intended for calculating the placement of handles.

Connection anchors

Draw2D provides classes that provide various styles of anchor points, which are used to represent the ends of a connection. The basic function of these classes is to contain the location of a Connection's endpoints and to register listeners that will be notified if the end of a connection is moved.

The AbstractConnectionAnchor class is the base class for anchors whose position is associated with a figure. It notifies its listeners when the figure it is associated with is moved.

Available anchors include:

- ChopboxAnchor — A ChopboxAnchor is located at the point on the figure's border where the Connection would intersect the figure, if the connection continued to the figure's center point.
- LabelAnchor — LabelAnchor is a subclass of ChopboxAnchor that is designed solely for node figures that are Draw2D Labels. Rather than projecting the connection to the center of the figure, the location of the anchor depends on the center of the Label's icon.
- EllipseAnchor — The EllipseAnchor is a variant of the ChopboxAnchor (but it is not a subclass). It locates the anchor on the edge of an elliptical figure at the point where a connection to the center of the node would intersect the edge.
- XYAnchor — The XYAnchor is used for anchors that are placed at a fixed position.

Connection routers

Connection routers are used to calculate the path that a connection takes in getting from one anchor to the other. AbstractRouter is the base class for connection routers that implement the ConnectionRouter interface. Available connection routers include:

- NullConnectionRouter — By default this simply draws a straight line between the anchors of a connection. This is shown in Figure 3-6 using a diagram created using the logic sample application.However Draw2D also provides more sophisticated routers that use different criteria to determine the path that a connection will take.

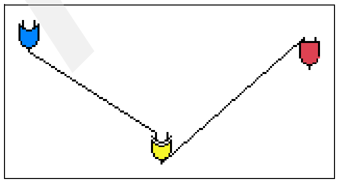


Figure 3-6 NullConnectionRouter

- AutomaticRouter — This provides a base class for routers that want to prevent two connections from overlaying each other. For instance its FanRouter subclass spreads two connections which have the same starting and ending points so that they are not superimposed.
- BendpointConnectionRouter — The BendpointConnectionRouter shown in Figure 3-7, allows the user to manually insert bendpoints into a connection. The connection is routed to follow a set of points that the user specifies by manually dragging the Connection's segments.

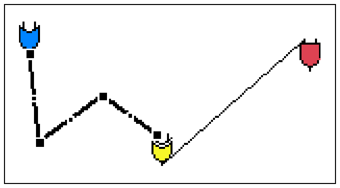


Figure 3-7 Bendpoint router

- ManhattanConnectionRouter — The ManhattanConnectionRouter (Figure 3-8) routes a connection using only vertical and horizontal line segments. It also maintains separation between Connections that would otherwise overlap.

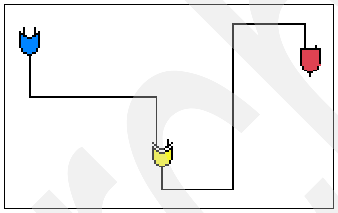


Figure 3-8 Manhattan router

Summary of Draw2D's support for connections

Many of the features in Draw2D that we have discussed are part of Draw2D's support for connections. It is worthwhile to summarize this set of features:

- PolylineConnection, a polyline figure that listens for anchor movement and supports start and end-point decorations, and has an associated connection router
- A connection layer exclusively for drawing connections
- Anchors for specifying and tracking the connection points of connections
- Routers to determine the path of connections
- Locators to find the ends and midpoint of connections, and more specialized ones to support connections with multiple bendpoints

- Rotatable decorations to place decorations on connections that can realign themselves as the angle of the connection changes.

3.3 The GEF framework

This sections covers the basic framework classes and concept of the Graphical Editing Framework.

3.3.1 Prerequisites

We assume that you already have good knowledge and experience in Eclipse plug-in development. You should have understand the concepts of Eclipse views and editors.

The following articles from eclipse.org are very useful for understanding terms and concepts mentioned in this chapter:

- *Eclipse Platform Technical Overview*
- *Notes on the Eclipse Plug-in Architecture* by Azad Bolour
- *How to Use the Eclipse API* by Jim des Rivieres
- *Creating an Eclipse View* by Dave Springgay
- *Getting Started with the Graphical Editing Framework* by Randy Hudson

You should have installed Eclipse SDK 2.1.1 including GEF SDK 2.1.1, and you should be familiar with Draw2D concepts and terms provided by the developers guides, which are available in the Eclipse online help.

3.3.2 EditParts

EditParts are the central elements in GEF applications. They are the controllers that specify how model elements are mapped to visual figures and how these figures behave in different situations.

Usually you will have to create an EditPart class for every model element class so you will have likely the same class hierarchy for the EditParts as you have for your model. The process of creating EditPart instances is not covered here. It will be explained in a later section.

EditParts are defined through the interface org.eclipse.gef.EditPart. See org.eclipse.gef.AbstractEditPart for an abstract base implementation of this interface. We strongly recommend (as do the GEF development team) that you do not implement the interface yourself. Instead, subclass the provided abstract base class AbstractEditPart.

Note: In general, you should always subclass a provided base implementation rather then implementing the interface yourself. This protects you from unexpected API changes and reduces your work in case of API changes (for example, when new methods are added to the interface). It also helps your software to stay compatible with future versions.

Actually there are three different types of EditParts. For now, in this section, we will focus on only two of them: GraphicalEditParts and ConnectionEditParts. GraphicalEditParts are those EditParts that provide a graphical representation for their model. These graphical representations are figures. ConnectionEditParts represent connections between GraphicalEditParts.

The third type, TreeEditParts, is only interesting for building trees of your model with SWT tree widgets. This is not the primary intention of a graphical editor, but is probably useful for the outline view. Our redbook sample application will show you an introduction into this.

The EditPart interfaces provide a lot of methods. Your are not expected to call them, except for get/setModel, when necessary. Nearly all methods are used by the Graphical Editing Framework to handle the edit parts. But you can add your own methods, and we encourage you to add your EditPart implementations to ease your access to model elements and properties.

Life-cycle of EditParts

We already know that EditParts will be created somewhere and somehow by a factory (details about the factory will be explained later in 3.5.3, “Creating EditParts” on page 137). We will now focus on the methods involved in EditPart life-cycles.

When an EditPart was created, it is not yet visible or active. It becomes active when the Graphical Editing Framework gets informed about it. If an EditPart becomes obsolete for some reason (either the editor is closed or the model object represented by the EditPart was deleted) and the framework no longer needs it, it will be deactivated. There are two methods, EditPart#activate and EditPart#deactivate, which will be called by the framework when the state of an EditPart changes. A third method, EditPart#isActive, always returns the current activation state.

Note: Although the JavaDoc of these methods indicates that an EditPart may be reactivated after it was obsolete, we have not experienced such situations during our development.

We suggest not to develop with the reuse of EditParts in mind. You should not try to keep track of EditPart instances across editor sessions. If an EditPart gets deactivated, throw it away. Allow it to get garbage collected by the virtual machine. By handling EditPart instances this way, you do not need to worry about the memory overhead. It will be solved for you and you can enjoy the advantages of Java.

EditPart#deactivate is a good point to release resources used by your EditPart (for example SWT images or fonts). We strongly suggest that you read the section about SWT resource management 4.2.6, “Resource management” on page 149.

Figures

GraphicalEditParts have a figure that is the visual part of the model. The GraphicalEditPart need to create the figure, update it on model changes, and dispose it (if necessary) if the EditPart is deactivated.

The figure is created by AbstractGraphicalEditPart#createFigure only once and will be cached by the abstract base implementation. Remember that you should always inherit from abstract base implementations if possible. AbstractGraphicalEditPart#createFigure is called when a figure is requested via AbstractGraphicalEditPart#getFigure for the first time.

Updating the figure is done in AbstractEditPart#refreshVisuals. You need to overwrite this method and update your figure according to the model changes you encountered. We will explain this later.

More about figures can be found in the Draw2D developers guide (see the Eclipse online help) and in 3.2.3, “Figures” on page 95 of this book.

Connections

A ConnectionEditPart, which represents a connection between two EditParts, is nothing more than a GraphicalEditPart, which has a source and a target EditPart. Connections are connected to ConnectionAnchors. These anchors should be provided by the EditParts the ConnectionEditPart points to/comes from.

The recommended way is that each GraphicalEditPart that could be a source or a target for connections implements the NodeEditPart interface. It is the most common way of how application models work. Connections usually points to some locations of the figure, and this figure is provided by a GraphicalEditPart.

3.3.3 Requests

Requests are the communication objects used in the Graphical Editing Framework. They contain information that might be necessary for executing the request later. There are several type of requests available. The three main types that are used most often in typical GEF applications, are CreateRequests, GroupRequests and LocationRequests.

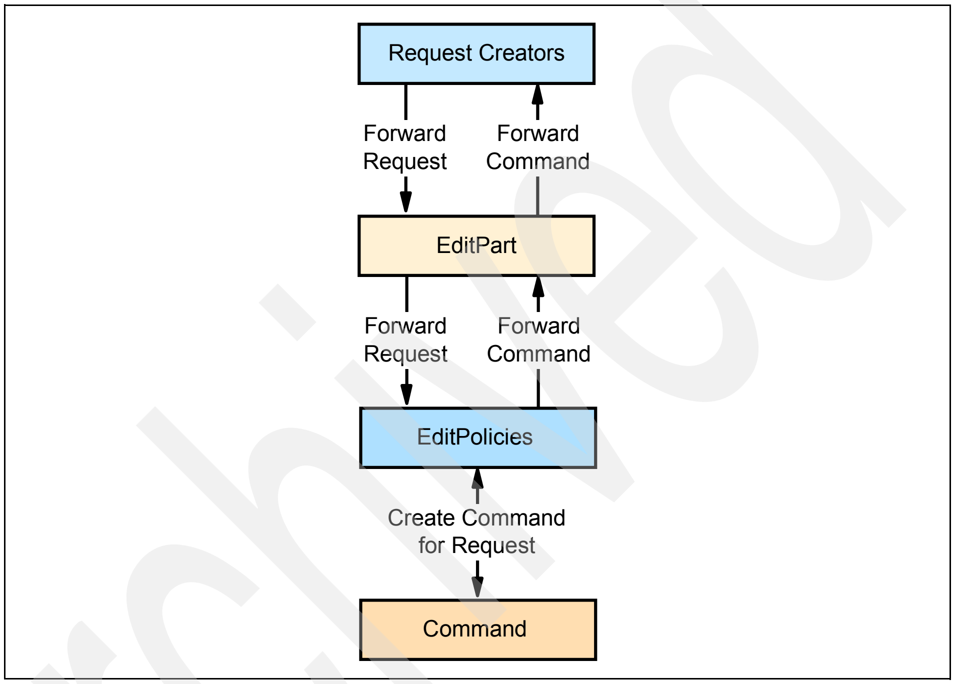


Figure 3-9 Communication chain Request — EditPart — Command

Figure 3-9 shows the typical communication chain of a request and the objects involved. As you can see, someone (typically a tool, an action or some drag or drop handler) creates a request. This request is forwarded to an EditPart. The EditPart doesn't process the request itself. Instead it delegates it to an EditPolicy, which understands the request. The EditPolicy itself creates a command for the request, which will be executed to fulfill the request.

CreateRequests

CreateRequests are used everywhere a new model object should be created. For connections the subclass CreateConnectionRequest is used. A CreateRequest has a CreationFactory, which you have to provide. This CreationFactory is responsible for creating the new model objects.

Tip: Actually, your CreationFactory implementation does not need to create new model objects. We suggest only submitting the type of the new model object and creating it later in a Command.

GroupRequests

GroupRequests are Requests that can span more than one EditPart into one single request. A typical GroupRequest is the ChangeBoundsRequests, which is responsible for moving and/or resizing EditParts.

LocationRequests

LocationRequests are requests that keep track of a location — for example, the SelectionRequest, which is responsible for selecting an EditPart. You can always determine where the user clicked into an EditPart to select it. This allows you to provide special behavior for different locations inside your EditPart.

3.3.4 EditPolicies

We already know that the communication inside the Graphical Editing Framework is done via requests and that these requests are forwarded to EditPolicies. What are EditPolicies, and why is this done in this way?

Actually, EditPolicies are those parts in the Graphical Editing Framework, which bring the editing functionallity into EditParts, This is done because it is a good object-oriented design.

An EditPolicy defines what can be done with an EditPart. EditParts without EditPolicies will do nothing. They won't even be selectable. EditPolicies are also responsible for feedback management (for example, what should be shown when an EditPart is moved or resized) and they are allowed to delegate work (forward requests) to other EditParts (for example, children).

EditPolicies are categorized into roles (see constants in interface org.eclipse.gef.EditPolicy) and EditParts are limited to have only one EditPolicy per role.

Component role

The component role is defined as EditPolicy#COMPONENT_ROLE, and the base class for these kind of EditPolicies is ComponentEditPolicy.

This is the main role for all fundamental operations that involve the model element of an EditPart directly (for example, deletion of the model element). Whenever a request has nothing to do with UI interaction and only does something on the model element, it is best handled by a command delivered from a ComponentEditPolicy.

Connection role

The connection role is defined as EditPolicy#CONNECTION_ROLE, and the base class is ConnectionEditPolicy. It is the corresponding component role for ConnectionEditParts.

Container role

The container role (EditPolicy#CONTAINER_ROLE, ContainerEditPolicy) is responsible for operations typically performed on containers (for example, the creation of children). Each EditPart with children would have a ContainerEditPolicy.

Layout role

The layout role (EditPolicy#LAYOUT_ROLE, LayoutEditPolicy) is responsible for containers that have a layout associated to them. It can calculate proper locations for requests and define where children should be placed.

Note: There is some overlap between ContainerEditPolicy and LayoutEditPolicy. ContainerEditPolicy is intended to be used in simple environments where it does not matter how children are placed. There is not any location information available inside ContainerEditPolicy.

Tree container role

The tree container role (EditPolicy#TREE_CONTAINER_ROLE, TreeContainerEditPolicy) is the corresponding container role for TreeEditParts.

Graphical node role

The graphical node role (EditPolicy#GRAPHICAL_NODE_ROLE, GraphicalNodeEditPolicy) is used for establishing and managing connections on EditParts. Whenever your EditPart deals with connections, it will need a GraphicalNodeEditPolicy.

Direct edit role

The direct edit role (EditPolicy#DIRECT_EDIT_ROLE, DirectEditPolicy) is used to bring direct editing behavior into EditParts. Thus, when the user double-clicks an EditPart, he will be able to directly edit properties on the figure.

Additional roles

More documentation about additional roles is available in the GEF developers guide available in the Eclipse online help.

3.3.5 Commands

A command is the part that actually modifies your model. Commands simplify the way of modifying your model because they provide support for:

- ► Execution limitations
- ► Undo and redo
- ► Combining and chaining

There is nothing more to say about commands than what can be found in the JavaDoc. You need to implement them and you need to instantiate them. The abstract base class is org.eclipse.gef.commands.Command.

3.3.6 GraphicalViewers

From the Draw2D developers guide, we know that figures are drawn by a LightweightSystem. But that is not all. There are some more components involved, which we do not want to take care of when we are developing our editor. That is why the Graphical Editing Framework provides the GraphicalViewer.

A GraphicalViewer provides a seamless (JFace-like integration of EditParts into the Eclipse workbench.

Typically, a JFace viewer only needs some content, a factory, and some configuration, and it is done. It already provides all necessary implementation for drag and drop support, event and update handling, and other complicated tasks. A GEF GraphicalViewer does exactly the same.

There are two GraphicalViewer implementations available: one that does support native scrolling (ScrollingGraphicalViewer) and one that does not (GraphicalViewerImpl). The most common case is to use a viewer that supports native scrolling. It is even possible to have a ScrollingGraphicalViewer never showing its scrollbars. Thus, we will focus on this implementation.

A GraphicalViewer can be created out of the box. It has a parameter-less constructor and provides the method createControl to create the SWT control of this viewer. You do not even need an editor for this. A GraphicalViewer can be used anywhere an SWT control is available.

After the viewer is instantiated and the control is created, you need only to attach a RootEditPart (EditPartViewer#setRootEditPart) and an EditPartFactory (EditPartViewer#setEditPartFactory) to it, and set the content (EditPartViewer#setContents). The content is a model element that is the root of your model.

Note: Please do not be confused by RootEditPart and the root of your model. Both are completely different and have no relation to each other.

A GraphicalViewer maintains a registry of all EditParts it contains. This map can be accessed by EditPartViewer#getEditPartRegistry. We are responsible for the key and the registration process, but the Graphical Editing Framework provides a default implementation, which automatically registers and unregisters EditParts using the model element they represent as key.

If you want to change the key mapping, you need to look at AbstractEditPart#registerModel and AbstractEditPart#unregisterModel.

3.3.7 RootEditParts

A RootEditPart is a special kind of an EditPart. It has absolutely no relation to your model and should not be understood as a typical EditPart.

The task of a RootEditPart is to provide a suitable and homogeneous environment for the real EditParts that represents your model. Thus, it can be understood as an interface between a GraphicalViewer and your model EditParts.

There are several implementations available, but actually only two of them should be used — ScalableRootEditPart and ScalableFreeformRootEditPart. All other implementations are either deprecated or provide only a subset of functionality of the implementations mentioned above.

Important: ScalableRootEditPart and ScalableFreeformRootEditPart can only be used inside a ScrollingGraphicalViewer.

Both implementations provide the possibility of scalability (zoom) support and introduce several layers to a GraphicalViewer. The only difference is that the ScalableFreeformRootEditPart can be extended in all directions, which enables negative coordinates.

Layers

Figure 3-10 gives an overview of the layers introduced by ScalableRootEditPart and ScalableFreeformRootEditPart.

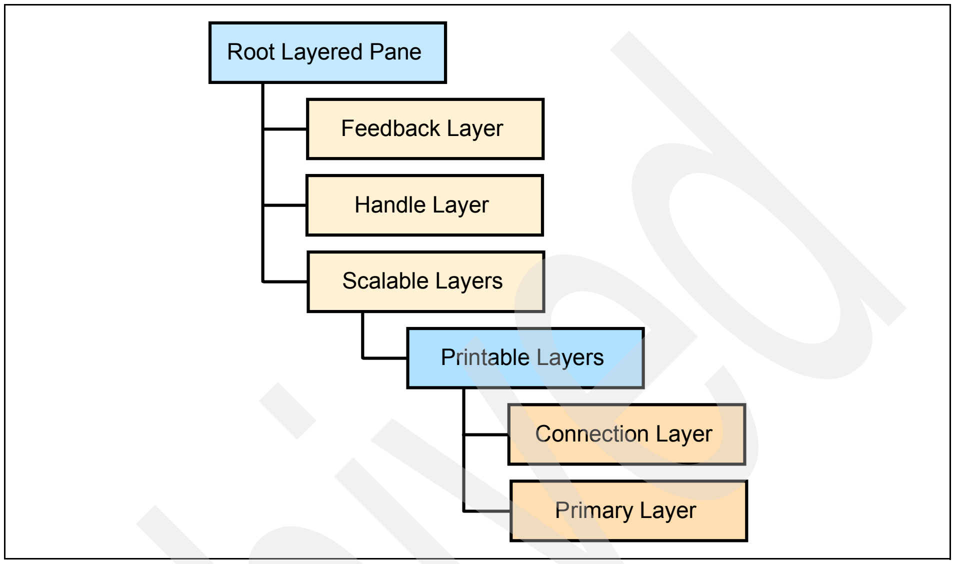


Figure 3-10 Layers of ScalableRootEditPart and ScalableFreeformRootEditPart

Layers are used to separate and/or group figures of EditParts to better control their overlapping. Actually, all figures are placed into the primary layer. Figures representing connections are placed on the connection layer and so they are always painted above the other figures. Special figures (like drag or drop feedback or handles) are painted into special layers above the scalable layers. This is important because if you ever want to paint something in the feedback or handle layers, you must be aware that you need to scale this manually before painting.

Freeform or not

When using the ScalableFreeformRootEditPart, your editor can extend in all directions. Thus, it is even possible to have negative coordinates. The (non-freeform) ScalableRootEditPart only allows extension in positive directions.

Restriction: When using the ScalableFreeformRootEditPart, the EditPart of your root model object must use a figure of type FreeformFigure.

3.4 Building an editor

We now know the base classes and concepts of the Graphical Editing Framework, and we are ready to build our first graphical editor skeleton. In this section we explain how to get started and then go forward step-by-step.

3.4.1 The editor class

First, we have to create the plug-in and then define the editor extension. We do not describe this here because it is a common process of the Eclipse plug-in programming model. The Eclipse documentation provides detailed information about this.

By default, the editor class is created by extending org.eclipse.ui.part.EditorPart. This is the main class of the editor and is responsible for receiving the input, creating and configuring the viewer, handling the input, and saving the input.

Typically you will already have a save option in your model, so we do not discuss the implementation of the methods save, isSaveAsAllowed, and saveAs here.

As a result, we will have the class skeleton shown in Example 3-2.

Note: JavaDoc comments have been removed for readability reasons.

Example 3-2 ExampleGEFEditor.java (initial stage)

```
/**
 * This is the example editor skeleton that is build
 * in <i>Building an editor</i> in chapter <i>Introduction to GEF</i>.
 *
 * @see org.eclipse.ui.part.EditorPart
 */
public class ExampleGEFEditor extends EditorPart
{
   public ExampleGEFEditor()
   {}

   public void createPartControl(Composite parent)
   {}

   public void setFocus()
   {
      // what should be done if the editor gains focus?
      // it's your part
   }
```

```
    public void doSave(IProgressMonitor monitor)
    {
       // your save implementation here
    }

    public void doSaveAs()
    {
       // your save as implementation here
    }

    public boolean isDirty()
    {
       return false;
    }

    public boolean isSaveAsAllowed()
    {
       // your implementation here
       return false;
    }

    public void gotoMarker(IMarker marker)
    {}

    public void init(IEditorSite site, IEditorInput input)
       throws PartInitException
    {}
}
```

3.4.2 EditDomain

Next we need an EditDomain. An EditDomain is an interface that logically bundles an editor, viewers, and tools. Therefore, it defines the real editor application.

An EditDomain provides a CommandStack, which keeps track of all executed commands. This is necessary for undo and redo operations and useful to determine if the model was modified (is dirty) or not.

Usually you will have one EditDomain per editor, but it is also possible to share an EditDomain across several editors in a multi-page editor.

It is up to you when to create the EditDomain. It is possible to create it lazily. You can use the class EditDomain directly, however, the Graphical Editing Framework provides an implementation, which additionally knows about the editor that created it. This implementation is called DefaultEditDomain and used in our example shown in Example 3-3.

Example 3-3 Adding EditDomain to the editor

```
/** the <code>EditDomain</code>, will be initialized lazily */
private EditDomain editDomain;

/**
 * Returns the <code>EditDomain</code> used by this editor.
 * @return the <code>EditDomain</code> used by this editor
 */
public EditDomain getEditDomain()
{
    if (editDomain == null)
        editDomain = new DefaultEditDomain(this);
    return editDomain;
}
```

3.4.3 CommandStack

After adding the EditDomain, we have access to the CommandStack. We will use the CommandStack to indicate when an editor is dirty.

Note: If you ever execute a command yourself, please ensure that you execute it through the CommandStack.

The CommandStack contains the method isDirty, which indicates if a CommandStack has executed commands after the last save. How does the CommandStack know about the last save? A CommandStack knows about this because we have to tell it whenever the editor is saved.

This is not done by simply delegating the editors isDirty method to the CommandStack; instead, we need a listener that listens to CommandStack changes and updates the dirty state of our editor. Whenever this state changes, we need to inform the Eclipse workbench. But you need not be concerned about this. The superclass EditorPart provides methods for the last part.

We start with the last part, as it is the easiest task. We simply add a flag for the dirty state and a setter that automatically fires an event, as shown in Example 3-4.

Example 3-4 Indicating the dirty state of our editor (part 1)

```
/** the dirty state */
private boolean isDirty;

/**
 * Indicates if the editor has unsaved changes.
 * @see EditorPart#isDirty
 */
public boolean isDirty()
{
    return isDirty;
}

/**
 * Sets the dirty state of this editor.
 *
 * <p>An event will be fired immediately if the new
 * state is different than the current one.
 *
 * @param dirty the new dirty state to set
 */
protected void setDirty(boolean dirty)
{
    if(isDirty != dirty)
    {
        isDirty = dirty;
        firePropertyChange(IEditorPart.PROP_DIRTY);
    }
}
```

Now we implement the listener and attach it to the CommandStack as shown in Example 3-5.

Example 3-5 The CommandStackListener

```
/**
 * The <code>CommandStackListener</code> that listens for
 * <code>CommandStack </code>changes.
 */
private CommandStackListener commandStackListener = new CommandStackListener()
{
    public void commandStackChanged(EventObject event)
    {
        setDirty(getCommandStack().isDirty());
    }
};

/**
 * Returns the <code>CommandStack</code> of this editor's
 * <code>EditDomain</code>.
 *
 * @return the <code>CommandStack</code>
 */
public CommandStack getCommandStack()
{
    return getEditDomain().getCommandStack();
}

/**
 * Returns the <code>CommandStackListener</code>.
 * @return the <code>CommandStackListener</code>
 */
protected CommandStackListener getCommandStackListener()
{
    return commandStackListener;
}
```

Attaching the listener should be done when the editor gets it input, and removing it should be done in the editor's dispose method. See Example 3-6.

Example 3-6 Attaching and removing the CommandStackListener

```
/**
 * Initializes the editor.
 * @see EditorPart#init
 */
public void init(IEditorSite site, IEditorInput input)
    throws PartInitException
{
    // store site and input
    setSite(site);
    setInput(input);

    // add CommandStackListener
    getCommandStack().addCommandStackListener(getCommandStackListener());
}

/* (non-Javadoc)
 * @see org.eclipse.ui.IWorkbenchPart#dispose()
 */
public void dispose()
{
    // remove CommandStackListener
    getCommandStack().removeCommandStackListener(getCommandStackListener());

    // important: always call super implementation of dispose
    super.dispose();
}
```

Do not forget to update the CommandStack when the editor content is saved. See Example 3-7.

Example 3-7 Update CommandStack on editor save

```
/**
 * TODO: Implement "doSave".
 * @see EditorPart#doSave
 */
public void doSave(IProgressMonitor monitor)
{
    // your implementation here

    // update CommandStack
    getCommandStack().markSaveLocation();
}

/**
 * TODO: Implement "doSaveAs".
 * @see EditorPart#doSaveAs
 */
public void doSaveAs()
{
    // your implementation here

    // update CommandStack
    getCommandStack().markSaveLocation();
}
```

3.4.4 Attaching the viewer

The GraphicalViewer is the next element that must be integrated into our editor. The method createPartControl is the best location to do this. First we create a GraphicalViewer, then we configure this instance, and add it to the EditDomain. See Example 3-8.

Note: Although we have chosen to use the ScalableFreeformRootEditPart here, you are free to use whatever RootEditPart you like.

Example 3-8 Attaching a GraphicalViewer to our editor

```
/** the graphical viewer */
private GraphicalViewer graphicalViewer;

/**
 * Creates the controls of the editor.
 * @see EditorPart#createPartControl
 */
public void createPartControl(Composite parent)
{
    graphicalViewer = createGraphicalViewer(parent);
}

/**
 * Creates a new <code>GraphicalViewer</code>, configures, registers
 * and initializes it.     *
 * @param parent the parent composite
 * @return a new <code>GraphicalViewer</code>
*/
private GraphicalViewer createGraphicalViewer(Composite parent)
{
    // create graphical viewer
    GraphicalViewer viewer = new ScrollingGraphicalViewer();
    viewer.createControl(parent);

    // configure the viewer
    viewer.getControl().setBackground(parent.getBackground());
    viewer.setRootEditPart(new ScalableFreeformRootEditPart());

    // hook the viewer into the EditDomain
    getEditDomain().addViewer(viewer);

    // acticate the viewer as selection provider for Eclipse
    getSite().setSelectionProvider(viewer);

    // initialize the viewer with input
    viewer.setEditPartFactory(getEditPartFactory());
    viewer.setContents(getContent());

    return viewer;
}

/**
 * Returns the <code>GraphicalViewer</code> of this editor.
 * @return the <code>GraphicalViewer</code>
 */
public GraphicalViewer getGraphicalViewer()
{
    return graphicalViewer;
}

/**
 * Returns the content of this editor
 * @return the model object
 */
protected Object getContent()
{
    // todo return your model here
    return null;
```

```
}

/**
 * Returns the <code>EditPartFactory</code> that the
 * <code>GraphicalViewer</code> will use.
 * @return the <code>EditPartFactory</code>
 */
protected EditPartFactory getEditPartFactory()
{
    // todo return your EditPartFactory here
    return null;
}
```

3.4.5 Being adaptable

One of the key concepts inside Eclipse is the IAdaptable technology. It is also used within the Graphical Editing Framework. That is why we have to ensure that our editor implements this interface so that the GEF elements we have created provide adaptable behavior, which may be of interest to the Graphical Editing Framework itself or to other Eclipse code. So far, we have created the following important GEF elements:

- EditDomain
- GraphicalViewer

EditDomain also provides access to a third important element, CommandStack. Example 3-9 shows how to provide adapter access to the elements in our sample editor.

Example 3-9 Overwriting the getAdapter method

```
/* (non-Javadoc)
 * @see org.eclipse.core.runtime.IAdaptable#getAdapter(java.lang.Class)
 */
public Object getAdapter(Class adapter)
{
    // we need to handle common GEF elements we created
    if (adapter == GraphicalViewer.class || adapter == EditPartViewer.class)
        return getGraphicalViewer();
    else if (adapter == CommandStack.class)
        return getCommandStack();
    else if (adapter == EditDomain.class)
        return getEditDomain();

    // the super implementation handles the rest
    return super.getAdapter(adapter);
}
```

3.4.6 Introducing the palette

The palette in GEF editors is the home for tools. But before we discuss tools, we need to create a palette inside our editor. The GEF palette is implemented reusing GEF technology; thus it has a model presented in a viewer (the PaletteViewer).

The palette model

The palette model is a simple model starting with a PaletteRoot. Each PaletteViewer needs a PaletteRoot. The PaletteRoot is a palette container (PaletteContainer). Palette containers are used to organize palette entries (PaletteEntry).

Besides the PaletteRoot, there are two additional palette containers — PaletteGroup and PaletteDrawer. We suggest that you use both of them to organize your palette. Each provides a container for palette entries, but the PaletteGroup cannot be collapsed, while the PaletteDrawer can be collapsed.

Additional information can be found in the GEF JavaDoc.

Attaching the palette

Attaching a palette is similar to attaching a viewer. First, we need to create a new PaletteViewer, as shown in Example 3-10.

Example 3-10 Creating a PaletteViewer

```
/**
 * Creates a new <code>PaletteViewer</code>, configures, registers
 * and initializes it.
 * @param parent the parent composite
 * @return a new <code>PaletteViewer</code>
 */
private PaletteViewer createPaletteViewer(Composite parent)
{
    // create graphical viewer
    PaletteViewer viewer = new PaletteViewer();
    viewer.createControl(parent);

    // hook the viewer into the EditDomain (only one palette per EditDomain)
    getEditDomain().setPaletteViewer(viewer);

    // important: the palette is initialized via EditDomain
    getEditDomain().setPaletteRoot(getPaletteRoot());

    return viewer;
}
```

```
/**
 * Returns the <code>PaletteRoot</code> this editor's palette uses.
 * @return the <code>PaletteRoot</code>
 */
protected PaletteRoot getPaletteRoot()
{
    // todo add your palette entries here
    return null;
}
```

Next, we need to add this viewer to the editor's composite. The SWT SashForm is used to have the palette's width modifiable, as shown in Example 3-11.

Example 3-11   Adding the PaletteViewer to the editor's composite

```
/**
 * Creates the controls of the editor.
 * @see EditorPart#createPartControl
 */
public void createPartControl(Composite parent)
{
    SashForm sashForm = new SashForm(parent, SWT.HORIZONTAL);
    sashForm.setWeights(new int[] {30,70});
    paletteViewer = createPaletteViewer(sashForm);
    graphicalViewer = createGraphicalViewer(sashForm);
}

/** the palette viewer */
private PaletteViewer paletteViewer;

/**
 * Returns the <code>PaletteViewer</code> of this editor.
 * @return the <code>PaletteViewer</code>
 */
public PaletteViewer getPaletteViewer()
{
    return paletteViewer;
}
```

There are several default tools available, and we need to add them so that we have an initial PaletteRoot. See Example 3-12.

Example 3-12   Initial PaletteRoot with default tools

```
/** the palette root */
private PaletteRoot paletteRoot;

/**
```

```
     * Returns the <code>PaletteRoot</code> this editor's palette uses.
     * @return the <code>PaletteRoot</code>
     */
    protected PaletteRoot getPaletteRoot()
    {
        if (null == paletteRoot)
        {
            // create root
            paletteRoot = new PaletteRoot();

            List categories = new ArrayList();

            // a group of default control tools
            PaletteGroup controls = new PaletteGroup("Controls");

            // the selection tool
            ToolEntry tool = new SelectionToolEntry();
            controls.add(tool);

            // use selection tool as default entry
            paletteRoot.setDefaultEntry(tool);

            // the marquee selection tool
            controls.add(new MarqueeToolEntry());

            // a separator
            PaletteSeparator separator =
                new PaletteSeparator(
                    EditorExamplePlugin.PLUGIN_ID + ".palette.seperator");
            separator.setUserModificationPermission(
                PaletteEntry.PERMISSION_NO_MODIFICATION);
            controls.add(separator);

            // a tool for creating connection
            controls.add(
                new ConnectionCreationToolEntry(
                    "Connections",
                    "Create Connections",
                    null,
                    ImageDescriptor.createFromFile(
                        getClass(),
                        "icons/connection16.gif"),
                    ImageDescriptor.createFromFile(
                        getClass(),
                        "icons/connection24.gif")));

            // todo add your palette drawers and entries here

            // add all categroies to root
            paletteRoot.addAll(categories);
        }
        return paletteRoot;
    }
```

Palette customizer

It is possible to attach a palette customizer to the palette. This will enable the users of your editor to modify the palette to work in the way they prefer. For implementation details, please see our completed redbook sample application as described in Chapter 7, “Implementing the sample” on page 203, or the Logic example application provided from the GEF development team.

3.4.7 Actions

Actions are common objects in the Eclipse workbench to do something when user requests are initiated through menu items, toolbar buttons or context menu items. The Graphical Editing Framework provides a set of standard actions and an infrastructure for using these actions within the Graphical Editing Framework.

ActionRegistry

The class org.eclipse.gef.actions.ActionRegistry serves as a container for editor actions. The editor is responsible for providing and maintaining an ActionRegistry. See Example 3-13.

Example 3-13 Adding an ActionRegistry to the editor

```
/** the editor's action registry */
private ActionRegistry actionRegistry;

/**
 * Returns the action registry of this editor.
 * @return the action registry
 */
public ActionRegistry getActionRegistry()
{
    if (actionRegistry == null)
        actionRegistry = new ActionRegistry();

    return actionRegistry;
}

/* (non-Javadoc)
 * @see org.eclipse.core.runtime.IAdaptable#getAdapter(java.lang.Class)
 */
public Object getAdapter(Class adapter)
{
    // we need to handle common GEF elements we created
    if (adapter == GraphicalViewer.class
        || adapter == EditPartViewer.class)
        return getGraphicalViewer();
    else if (adapter == CommandStack.class)
        return getCommandStack();
    else if (adapter == EditDomain.class)
        return getEditDomain();
    else if (adapter == ActionRegistry.class)
        return getActionRegistry();

    // the super implementation handles the rest
    return super.getAdapter(adapter);
}

/* (non-Javadoc)
 * @see org.eclipse.ui.IWorkbenchPart#dispose()
 */
public void dispose()
{
    // remove CommandStackListener
    getCommandStack().removeCommandStackListener(getCommandStackListener());

    // disposy the ActionRegistry (will dispose all actions)
    getActionRegistry().dispose();

    // important: always call super implementation of dispose
    super.dispose();
}
```

Managing actions

As soon as we have the ActionRegistry, we are able to create actions and to register them.

The Graphical Editing Framework provides a set of default actions (redo, undo, delete, print, and save). These actions need some special handling to stay up-to-date with the editor, the CommandStack or the EditParts. The GEF default actions are not implemented as listeners to some events. Instead, they have to be updated manually. This can be done from within the editor as shown in Example 3-14.

Example 3-14 Added infrastructure for supporting different actions

```
/** the list of action ids that are to EditPart actions */
private List editPartActionIDs = new ArrayList();

/** the list of action ids that are to CommandStack actions */
```

```
private List stackActionIDs = new ArrayList();

/** the list of action ids that are editor actions */
private List editorActionIDs = new ArrayList();

/**
 * Adds an <code>EditPart</code> action to this editor.
 *
 * <p><code>EditPart</code> actions are actions that depend
 * and work on the selected <code>EditPart</code>s.
 *
 * @param action the <code>EditPart</code> action
 */
protected void addEditPartAction(SelectionAction action)
{
    getActionRegistry().registerAction(action);
    editPartActionIDs.add(action.getId());
}

/**
 * Adds an <code>CommandStack</code> action to this editor.
 *
 * <p><code>CommandStack</code> actions are actions that depend
 * and work on the <code>CommandStack</code>.
 *
 * @param action the <code>CommandStack</code> action
 */
protected void addStackAction(StackAction action)
{
    getActionRegistry().registerAction(action);
    stackActionIDs.add(action.getId());
}

/**
 * Adds an editor action to this editor.
 *
 * <p><Editor actions are actions that depend
 * and work on the editor.
 *
 * @param action the editor action
 */
protected void addEditorAction(EditorPartAction action)
{
    getActionRegistry().registerAction(action);
    editorActionIDs.add(action.getId());
}

/**
 * Adds an action to this editor's <code>ActionRegistry</code>.
 * (This is a helper method.)
 *
 * @param action the action to add.
 */
protected void addAction(IAction action)
{
    getActionRegistry().registerAction(action);
}
```

Now that we have the action infrastructure, we must implement the automatic updating of the different actions. Editor actions must be updated when the editor changes, CommandStack actions when the CommandStack changes, and EditPart actions when the selection changes. Example 3-15 shows how to add update support for actions to our sample editor.

Example 3-15 Adding update support for the actions

```
/**
 * Updates the specified actions.
 *
 * @param actionIds the list of ids of actions to update
 */
private void updateActions(List actionIds)
{
    for (Iterator ids = actionIds.iterator(); ids.hasNext();)
    {
        IAction action = getActionRegistry().getAction(ids.next());
        if (null != action && action instanceof UpdateAction)
            ((UpdateAction) action).update();

    }
}

/**
 * The <code>CommandStackListener</code> that listens for
 * <code>CommandStack </code>changes.
 */
private CommandStackListener commandStackListener =
    new CommandStackListener()
{
    public void commandStackChanged(EventObject event)
    {
        updateActions(stackActionIDs);
        setDirty(getCommandStack().isDirty());
    }
};

/** the selection listener */
```

```
private ISelectionListener selectionListener = new ISelectionListener()
{
    public void selectionChanged(IWorkbenchPart part, ISelection selection)
    {
        updateActions(editPartActionIDs);
    }
};

/**
 * Returns the selection listener.
 *
 * @return the <code>ISelectionListener</code>
 */
protected ISelectionListener getSelectionListener()
{
    return selectionListener;
}

/**
 * Initializes the editor.
 * @see EditorPart#init
 */
public void init(IEditorSite site, IEditorInput input)
    throws PartInitException
{
    // store site and input
    setSite(site);
    setInput(input);

    // add CommandStackListener
    getCommandStack().addCommandStackListener(getCommandStackListener());

    // add selection change listener
    getSite()
        .getWorkbenchWindow()
        .getSelectionService()
        .addSelectionListener(
        getSelectionListener());
}

/* (non-Javadoc)
 * @see org.eclipse.ui.IWorkbenchPart#dispose()
 */
public void dispose()
{
    // remove CommandStackListener
    getCommandStack().removeCommandStackListener(getCommandStackListener());

    // remove selection listener
    getSite()
        .getWorkbenchWindow()
        .getSelectionService()
        .removeSelectionListener(
        getSelectionListener());

    / disposy the ActionRegistry (will dispose all actions)
    getActionRegistry().dispose();

    // important: always call super implementation of dispose
    super.dispose();
}

/* (non-Javadoc)
 * @see org.eclipse.ui.part.WorkbenchPart#firePropertyChange(int)
 */
protected void firePropertyChange(int propertyId)
{
    super.firePropertyChange(propertyId);
    updateActions(editorActionIDs);
}
```

Now, when all the infrastructure is ready, we are able to create and add our actions as shown in Example 3-16.

Example 3-16 Adding actions to the editor

```
/**
 * Initializes the editor.
 * @see EditorPart#init
 */
public void init(IEditorSite site, IEditorInput input)
    throws PartInitException
{
    // store site and input
    setSite(site);
    setInput(input);

    // add CommandStackListener
    getCommandStack().addCommandStackListener(getCommandStackListener());

    // add selection change listener
    getSite()
        .getWorkbenchWindow()
        .getSelectionService()
        .addSelectionListener(
        getSelectionListener());

    // initialize actions
    createActions();
}

/**
 * Creates actions and registers them to the ActionRegistry.
 */
protected void createActions()
{
    addStackAction(new UndoAction(this));
    addStackAction(new RedoAction(this));

    addEditPartAction(new DeleteAction((IWorkbenchPart) this));

    addEditorAction(new SaveAction(this));
    addEditorAction(new PrintAction(this));
}
```

3.4.8 Adapting to the properties view

EditParts are responsible for delivering IPropertySource adapters for the properties view, but this is not discussed here.

The Graphical Editing Framework provides a solution to cover modifications occurred in the properties view into the CommandStack. This enables the possibility to undo and redo changes.

To enable this, the editor must deliver its own IPropertySheetPage. This IPropertySheetPage is a default PropertySheetPage customized with an undoable root entry provided by GEF. See Example 3-17.

Example 3-17 Making the properties view undoable

```
/* (non-Javadoc)
 * @see org.eclipse.core.runtime.IAdaptable#getAdapter(java.lang.Class)
 */
public Object getAdapter(Class adapter)
{
    // we need to handle common GEF elements we created
    if (adapter == GraphicalViewer.class
        || adapter == EditPartViewer.class)
        return getGraphicalViewer();
    else if (adapter == CommandStack.class)
        return getCommandStack();
    else if (adapter == EditDomain.class)
        return getEditDomain();
    else if (adapter == ActionRegistry.class)
        return getActionRegistry();
    else if (adapter == IPropertySheetPage.class)
```

```
        return getPropertySheetPage();

    // the super implementation handles the rest
    return super.getAdapter(adapter);
}

/** the undoable <code>IPropertySheetPage</code> */
private PropertySheetPage undoablePropertySheetPage;

/**
 * Returns the undoable <code>PropertySheetPage</code> for
 * this editor.
 *
 * @return the undoable <code>PropertySheetPage</code>
 */
protected PropertySheetPage getPropertySheetPage()
{
    if (null == undoablePropertySheetPage)
    {
        undoablePropertySheetPage = new PropertySheetPage();
        undoablePropertySheetPage.setRootEntry(
            GEFPlugin.createUndoablePropertySheetEntry(getCommandStack()));
    }

    return undoablePropertySheetPage;
}
```

3.4.9 Providing an outline view

Providing an outline view is handled in the typical Eclipse way. We need to provide an adapter of type IContentOutlinePage. We can do this in several ways.

One way is to create a complete SWT based outline view without using the Graphical Editing Framework. In many cases this is suitable and can be easily done, because many components, such as content and label providers or even tree viewers, can be reused to show a tree of your model objects.

If you do not have these reusable components available, then a second way is to build a tree with GEF components. The Graphical Editing Framework provides a TreeViewer and TreeEditParts for this case. You can also reuse actions created for your graphical editor. For details about implementing a model tree with the GEF TreeViewer and TreeEditParts, please see our redbook sample application.

A third way is to provide an overview of your graphical editor. Example 3-18 and Example 3-19 show a possible implementation of this.

Example 3-18 An overview outline page

```
/**
 * This is a sample implementation of an outline page showing an
 * overview of a graphical editor.
 *
 * @author Gunnar Wagenknecht
 */
public class OverviewOutlinePage extends Page implements IContentOutlinePage
{

    /** the control of the overview */
    private Canvas overview;

    /** the root edit part */
    private ScalableFreeformRootEditPart rootEditPart;

    /** the thumbnail */
    private Thumbnail thumbnail;

/    **
     * Creates a new OverviewOutlinePage instance.
     * @param rootEditPart the root edit part to show the overview from
     */
    public OverviewOutlinePage(ScalableFreeformRootEditPart rootEditPart)
    {
        super();
        this.rootEditPart = rootEditPart;
    }

    /* (non-Javadoc)
     * @see ISelectionProvider#addSelectionChangedListener
     * (ISelectionChangedListener)
     */
    public void addSelectionChangedListener(ISelectionChangedListener listener)
    {}

    /* (non-Javadoc)
     * @see IPage#createControl(Composite)
     */
    public void createControl(Composite parent)
    {
        // create canvas and lws
        overview = new Canvas(parent, SWT.NONE);
        LightweightSystem lws = new LightweightSystem(overview);

        // create thumbnail
        thumbnail =
            new ScrollableThumbnail((Viewport) rootEditPart.getFigure());
        thumbnail.setBorder(new MarginBorder(3));
        thumbnail.setSource(
            rootEditPart.getLayer(LayerConstants.PRINTABLE_LAYERS));
        lws.setContents(thumbnail);
    }

    /* (non-Javadoc)
     * @see org.eclipse.ui.part.IPage#dispose()
     */
    public void dispose()
    {
        if (null != thumbnail)
            thumbnail.deactivate();

        super.dispose();
    }

    /* (non-Javadoc)
     * @see org.eclipse.ui.part.IPage#getControl()
     */
    public Control getControl()
    {
        return overview;
    }

    /* (non-Javadoc)
     * @see org.eclipse.jface.viewers.ISelectionProvider#getSelection()
     */
    public ISelection getSelection()
    {
        return StructuredSelection.EMPTY;
    }

    /* (non-Javadoc)
     * @see ISelectionProvider#removeSelectionChangedListener
     * (ISelectionChangedListener)
     */
    public void removeSelectionChangedListener(
        ISelectionChangedListener listener)
    {}

    /* (non-Javadoc)
     * @see org.eclipse.ui.part.IPage#setFocus()
     */
    public void setFocus()
    {
        if (getControl() != null)
            getControl().setFocus();
    }
```

```
    /* (non-Javadoc)
     * @see ISelectionProvider#setSelection(ISelection)
     */
    public void setSelection(ISelection selection)
    {}
}
```

Now this page can be used in the editor, as shown in Example 3-19.

Example 3-19 Attaching the overview to the editor

```
/* (non-Javadoc)
 * @see org.eclipse.core.runtime.IAdaptable#getAdapter(java.lang.Class)
 */
public Object getAdapter(Class adapter)
{
    // we need to handle common GEF elements we created
    if (adapter == GraphicalViewer.class
        || adapter == EditPartViewer.class)
        return getGraphicalViewer();
    else if (adapter == CommandStack.class)
        return getCommandStack();
    else if (adapter == EditDomain.class)
        return getEditDomain();
    else if (adapter == ActionRegistry.class)
        return getActionRegistry();
    else if (adapter == IPropertySheetPage.class)
        return getPropertySheetPage();
    else if (adapter == IContentOutlinePage.class)
        return getOverviewOutlinePage();

    // the super implementation handles the rest
    return super.getAdapter(adapter);
}

/** the overview outline page */
private OverviewOutlinePage overviewOutlinePage;

/**
 * Returns the overview for the outline view.
 *
 * @return the overview
 */
protected OverviewOutlinePage getOverviewOutlinePage()
{
    if (null == overviewOutlinePage && null != getGraphicalViewer())
    {
        RootEditPart rootEditPart = getGraphicalViewer().getRootEditPart();
        if (rootEditPart instanceof ScalableFreeformRootEditPart)
        {
            overviewOutlinePage =
                new OverviewOutlinePage(
                    (ScalableFreeformRootEditPart) rootEditPart);
        }
    }

    return overviewOutlinePage;
}
```

3.4.10 Controlling your editor with the keyboard

The Graphical Editing Framework uses the concept of KeyHandlers to answer key strokes. By default, the anGEF GraphicalViewer does not answer key strokes. We have to enable this.

This is not a difficult task, because as with all other GEF concepts, there is a default implementation available, which provides a very feature-rich set of keys for interacting with a GraphicalViewer. The default implementation is the class org.eclipse.gef.ui.parts.GraphicalViewerKeyHandler. Example 3-20 shows how to use this key handler with our editor sample.

Example 3-20 Enabling our editor for keyboard interaction

```
/**
 * Creates a new <code>GraphicalViewer</code>, configures, registers
 * and initializes it.
 * @param parent the parent composite
 * @return a new <code>GraphicalViewer</code>
 */
private GraphicalViewer createGraphicalViewer(Composite parent)
{
    // create graphical viewer
    GraphicalViewer viewer = new ScrollingGraphicalViewer();
    viewer.createControl(parent);

    // configure the viewer
    viewer.getControl().setBackground(parent.getBackground());
    viewer.setRootEditPart(new ScalableFreeformRootEditPart());
    viewer.setKeyHandler(new GraphicalViewerKeyHandler(viewer));

    // hook the viewer into the EditDomain
    getEditDomain().addViewer(viewer);

    // acticate the viewer as selection provider for Eclipse
    getSite().setSelectionProvider(viewer);
```

```
        // initialize the viewer with input
        viewer.setEditPartFactory(getEditPartFactory());
        viewer.setContents(getContent());

        return viewer;
    }
```

Tip: If you like to attach actions to your own key strokes, you do not need to overwrite the GraphicalViewerKeyHandler. It is simply possible to attach a parent to KeyHandlers. Thus, you simply create your own KeyHandler instance (not GraphicalViewerKeyHandler), configure this KeyHandler instance, and set it as the parent of the GraphicalViewerKeyHandler you created for the GraphicalViewer.

3.5 Managing your model

Now that the editor base is built, you probably need to start reflecting your model. In this section we provide an overview of things to consider and describe some important issues related to handling models with our editor.

3.5.1 Reflecting a model

First we have to think about the architecture of our EditParts. The easiest way is usually to build the EditParts according to our model, but sometimes you may like another kind of representation of the model.

Whether you want a simple one-to-one representation or not, you need to have one main EditPart. This main EditPart is also called content EditPart and serves as the main entry point for your representation. It is important to understand this because each EditPartViewer can only have one content EditPart.

A content EditPart has nothing to do with a RootEditPart, but it might be possible that a RootEditPart defines some restrictions on the figure of the content EditPart (for example ScalableFreeformRootEditPart).

The figure of a content EditPart serves as the background figure of your graphical editor. Children can only be placed inside a figure.

Graphical model properties

A graphical editor presents your model in a graphical way. It is quite common for editors to allow users to lay out things the way they like. We have to think about this and decide which kind of support we will provide for layout.

The Graphical Editing Framework can use any Draw2D layout manager that is available. Some layouts requires the use of constraints (for example, location and size). These constraints belongs to a certain graphical representation of a model element. You have to decide if these constraints are persistent or not. If they are persistent, you have to find a location where you can store the constraint information.

The best way to do this depends on your preference and what fits best with your model. Mostly it is possible to store constraints together with the model element, either as a model property or as some kind of element annotation. If you do not like doing this, it is also possible to store them separately from your model.

3.5.2 Communication

If a model element is changed somehow or somewhere; a new issue arises — we have to ensure that all graphical representations of the model are updated accordingly. This requires communication between the model and the controller, which represents it.

It is not acceptable practice for your model to know about its controller, but a model can call attention to itself and to the change which has affected it.

It is up to you how you want to implement this, but a common way is to create some kind of event object, which is fired every time a change is done in the model. Then every controller can register with the model element itself or with an event manager and listen for such events.

As already mentioned in 3.3.2, “EditParts” on page 103, this is done best in EditPart#activate and EditPart#deactivate. When an EditPart receives an event for its model element, it has to decide whether it was a simple property change that only affects the figure (UI representation) or a structural change. In the first case, you would simply call EditPart#refreshVisual; and in the second case, you need to call EditPart#refreshChildren.

There are some dependencies that must be considered in event based communication for EditParts; for example, when connections are reconnected to another EditPart, both the old and the new EditParts need to be refreshed. You need to work out which parts need to be refreshed, but this is logically quite easy. For example, if a connection is reconnected to a new element, then elements affected by this change should also fire events in this case.

3.5.3 Creating EditParts

As mentioned in 3.3.2, “EditParts” on page 103, creating EditParts is best done through a factory. The EditPartFactory interface is simple to implement. It has only one method. In that method, you need to create an EditPart for a given model element in a specified context. The context might be useful if you consider a different UI representation than your model actually shows. Of course it is also necessary that you associate the specified model element with the created EditPart before you return the new EditPart.

You may consider building a map between the created EditParts and the model elements, but we do not recommend this. The EditPartViewer maintains an EditPart registry. EditParts register themselves to that registry. The default EditPart implementation already does this for you by using the model element as the key. Using the EditPart registry is a safe way to map between EditParts and the model elements.

One possible need for an EditPart registry is a domain based (global) listener model where there is only one listener, which does not belong to the model. This *listener* receives all the events from all model objects, and therefore it needs to find EditParts.

Note: If your model elements need to locate EditParts, you are probably not using the model-view-controller paradigm and should consider another solution.

4

GEF examples

In this chapter, we cover some more advanced Graphical Editing Framework subjects and present our solutions and example code for useful or frequently requested techniques.

4.1 Additional concepts

In this section we look at some GEF and Draw2D concepts and features in greater detail.

4.1.1 RootEditParts

The RootEditPart is at the root of the EditPart hierarchy. It is the link between your application's root edit part and the EditPartViewer. GEF provides a few RootEditPart implementations that you can use. In order to clear up any potential confusion about which RootEditPart is appropriate for your application, we summarize the features of the various implementations below.

The code snippet in Example 4-1 illustrates the essential steps in initializing a GEF application. Notice that in an actual application, these steps may be distributed across more than one method. It shows an instance of the root EditPart being created and used to initialize the GraphicalViewer.

Example 4-1 Configuring a RootEditPart

```
EditDomain editDomain = new DefaultEditDomain(null);
ScalableFreeformRootEditPart root = new ScalableFreeformRootEditPart();
GraphicalViewer viewer = new ScrollingGraphicalViewer();
viewer.createControl(parent);
editDomain.addViewer(viewer);

viewer.setRootEditPart(root);
viewer.setEditPartFactory(new EditPartFactory());
```

In selecting a root EditPart, we can first eliminate the GraphicalRootEditPart class. This implementation has been deprecated and will eventually be removed from GEF. The equivalent functionality of this EditPart can be achieved by using a ScrollingGraphicalViewer with a ScalableRootEditPart.

The three root EditParts to consider using are:

- ScalableRootEditPart
- FreeformGraphicalRootEditPart
- ScalableFreeformRootEditPart

All three of these EditParts must be used with a ScrollingGraphicalViewer, and therefore they all support scrolling using scrollbars. The main deciding criteria are whether your application requires scalability or a freeform diagram.

Remember that a freeform diagram expands automatically in all directions as the user drags figures beyond the current bounds of the diagram. This feature is generally desirable whenever you want the user to control the placement of the figures in your application. On the other hand, if your application constrains the placement of graphical objects, for example, into cells of a grid, then the freeform feature might not be desirable. Table 4-1 summarizes the main characteristics of the three root EditParts.

Table 4-1 Root EditPart characteristics

EditPart	Primary Figure	Is freeform?	Is scalable?
ScalableRootEditPart	Viewport	no	yes
FreeformGraphicalRoot EditPart	FreeformViewport	yes	no
ScalableFreeformRoot EditPart	FreeformViewport	yes	yes

4.1.2 Coordinate systems

Figures have a protected method, useLocalCoordinates(), that allows subclasses of figure to choose a coordinate system for their child figures that is either absolute or relative to the parent figure. A figure whose parent uses local coordinates will have a bounds whose upper left coordinate will be (0,0).

A figure's getClientArea returns the rectangle in which child figures are visible. It is cropped by any border/insets that are in effect for the figure, and the origin of the rectangle is set to (0,0) if the figure is using local coordinates.

The figure class includes four methods for translating coordinates between relative and absolute coordinates:

- translateToParent() — translates a point in the figure's coordinates to its value in the parent's coordinates
- translateFromParent() — translates a point in the parent's coordinates to its coordinates in this figure
- translateToRelative(), translates an absolute coordinate to a coordinate that is relative to this figure, that is, recursively translates from parent
- translateToAbsolute() — translates a coordinate that is relative to this figure to an absolute coordinate, that is, recursively translates to parent

Anchors and locator reference points work with absolute coordinates. Hit testing uses local coordinates.

4.1.3 Layers

In 3.2, "Introduction to Draw2D" on page 93, we discussed support for graphical layers using the LayeredPanes and layers classes. This feature allows us to segregate graphical elements into layers based on their functionality, and then control their visibility, z-order, and targetability. In this section we look at some of the specific ways that layers are configured in GEF's root EditParts, and the possibilities for customizing this behavior.

GEF's root EditPart classes, ScalableRootEditPart, FreeformGraphicalRootEditPart, and ScalableFreeformRootEditPart classes all expose methods that allow subclasses to modify the structure of their layers. The protected methods createLayers() and createPrintableLayers() are where these classes set up their layers. The implementation of these methods in FreeformGraphicalRootEditPart is shown in Example 4-2. This is similar to the other EditPart classes, except that it does not include scaling support. Examining this code reveals the default layer organization in GEF:

- Only the primary and connection layers are printable.
- The feedback layer is on the top of the z-order, followed by the handle layer, and finally the printable layers.
- Within the printable layers, the connection layer is on top of the primary drawing layer.

Example 4-2 Layer creation methods in FreeformGraphicalRootEditPart

```
protected void createLayers(LayeredPane layeredPane) {
   layeredPane.add(getPrintableLayers(), PRINTABLE_LAYERS);
   layeredPane.add(new FreeformLayer(), HANDLE_LAYER);
   layeredPane.add(new FeedbackLayer(), FEEDBACK_LAYER);
}

/**
 * Creates a layered pane and the layers that should be printed.
 * @see org.eclipse.gef.print.PrintGraphicalViewerOperation
 * @return a new LayeredPane containing the printable layers
 */
protected LayeredPane createPrintableLayers() {
   FreeformLayeredPane layeredPane = new FreeformLayeredPane();
   layeredPane.add(new FreeformLayer(), PRIMARY_LAYER);
   layeredPane.add(new ConnectionLayer(), CONNECTION_LAYER);
   return layeredPane;
}
```

The EditParts that support scaling, that is, ScalableRootEditPart and ScalableFreeformRootEditPart, also contain the method:

```
   protected ScalableFreeformLayeredPane createScaledLayers();
```

By subclassing these root EditPart classes, you can gain control over the ordering of layers, or customize which layers are printable or scalable. There are probably few cases where it would be useful to modify the configuration of the “stock” layers. One application that has been discussed is to place connections under rather than over the primary figure layer. Although this can have some aesthetic advantages, if you are considering this, keep in mind that it will be possible for your figures to completely cover connections, making them difficult to access. It becomes more problematic when your application includes container nodes, because connections between nodes in a container will be occluded.

A more likely customization scenario is to create additional, custom layers, for example, to provide annotation layers that can be turned on and off, and selectively printed.

4.2 Techniques

In this section we discuss some GEF and Draw2d techniques that are useful when developing a GEF application.

4.2.1 Drag and drop

One of the most essential parts in today’s desktop applications is drag and drop. In this section we discuss drag and drop inside GEF applications.

As you might know, drag and drop is organized in SWT around Transfers. They are the base representation of something that is transferred between the SWT controls in an drag and drop operation. Basically, there is no difference compared with GEF.

The Graphical Editing Framework provides some classes and concepts to ease the development of drag and drop in GEF applications. For example, you won’t have to deal with SWT DragSource objects and other lower level classes.

The base concept in GEF is that you add TransferDragSourceListener and/or TransferDropTargetListener to an EditPartViewer. TransferDragSourceListeners are used to enable EditPartViewers as a source for drag operations and TransferDropTargetListener are used to enable EditPartViewers as target for drop operations.

When implementing drag and drop listeners for the viewers in your GEF application, be sure to inherit from the abstract base classes. A good point to look for further implementation information is the template drag and drop palette demonstrated in the Logic example application provided by GEF. Our sample application will also provide an introduction into implementing drag and drop.

4.2.2 Palette: Implementing a sticky tool preference

The default behavior for GEF tools is to unload a tool after it is used once. This causes the EditDomain's default tool, which is typically the SelectionTool, to be reactivated. This behavior is desirable for some types of operations and not for others. In addition, sometimes a tool that is normally used sporadically needs to be repeated several times, and then the default behavior becomes cumbersome.

In these cases it is useful to customize each tool's unloading behavior based on the type of tool, or based on user preference. The base class for GEF tools, org.eclipse.gef.tools.AbstractTool, contains the method setUnloadWhenFinished(boolean), which is used to control the unloading behavior. Because tools are not instantiated until they are ready to use, setting this property requires gaining access to tool instances, either in the factory that creates them, or by obtaining them from the EditDomain as they become activated.

The latter technique can be used in applications that use the GEF palette and whose editor class is derived from GraphicalEditorWithPalette.

1. Create a class that implements the org.eclipse.gef.palette.PaletteListener interface, which contains the single method:

   ```
   void activeToolChanged(PaletteViewer palette, ToolEntry tool)
   ```

   Add your listener to the palette by calling the palette's method:

   ```
   public void addPaletteListener(PaletteListener paletteListener);
   ```

The code inside your listener then needs to get the actual active tool from the EditDomain's getActiveTool() method. Then you can set the active tool's setUnloadWhenFinished() method to set this behavior based on whatever criteria your application wants to use, such as a user setting or preference, or based on what tool is active.

4.2.3 Printing

Starting with GEF version 2.0, there is built-in support for printing from GEF applications. You simply need to add a PrintAction to your editor's action registry as shown in Example 4-3.

Example 4-3 Adding a PrintAction

```
ActionRegistry registry = getActionRegistry();
IAction action;

action = new PrintAction(this); // “this” is your IEditorPart-derived editor
registry.registerAction(action);
```

In your application's action bar contributor class, you can provide keyboard or menu access to the PrintAction by calling:

```
addGlobalActionKey(IWorkbenchActionConstants.PRINT);
```

The Draw2D class PrintOperation and its subclasses PrintFigureOperation, PrintGraphicalViewerOperation provide support for printing in Draw2D, ultimately using a Graphics context, PrinterGraphics, that is created using an instance of SWT's org.eclipse.swt.printing.Printer.

The PrintGraphicalViewerOperation class locates the printable layers in your editor's viewer. The current selection in your editor is saved, then disabled while printing, and restored after printing. Only the contents of the printable layers are printed. Also, the parent figure's background color is set to white for printing, then restored. Figures are scaled by the ratio of the printer's resolution (in DPI®) to that of the display, so that the actual size of a figure is maintained.

4.2.4 Zooming

Support for zooming was added to GEF in version 2.0. The scaling functionality is built into ScalableLayeredPane and ScalableFreeformLayeredPane. These classes support scaling by maintaining the current scaling level, and by taking the scaling factor into account when doing point translations and calculating their client area and preferred size. They use the ScaledGraphics subclass of Graphics as the their graphics context for painting their child figures.

The ScaledGraphics class applies the current scale factor to the normal graphics operations, performing transformations on point lists and rectangles before painting them. It also scales fonts, stretches or shrinks bitmaps, and scales the line width for line drawing operations.

The ZoomManager class is used to manage zoom operations on ScalableFigure figures, that is, ScalableLayeredPane or ScalableFreeformLayeredPane. It provides several methods to control the zooming operations:

- It supports a zoom style, which currently cab be either a "jump" zoom or animated zoom.
- You can set a list of zoom levels, which is used be the zoomIn and zoomOut methods to determine the next of zoom.
- It can set a view location.
- You can set the zoom level to a specified magnification, or zoom in or out one level.
- It supports zoom listeners — controls that allow for zooming up or down can register themselves as zoom listeners, so that when the zoom level changes, they can determine their enablement.

The root EditParts ScalableFreeformRootEditPart and ScalableRootEditPart contain a ZoomManager, which is accessible via their getZoomManager() method. You can access the ZoomManager through your editor's GraphicalViewer:

```
ZoomManager zoomManager =
((ScalableFreeformRootEditPart)getGraphicalViewer().getRootEditPart()).
getZoomManager();
```

You can do this if you need to modify the ZoomManager's configuration, such as to set the supported zoom levels.

When you want to add zoom controls to your editor's user interface, GEF includes classes for zoom actions and a ContributionItem, named ZoomComboContributionItem. These are shown in Figure 4-1.

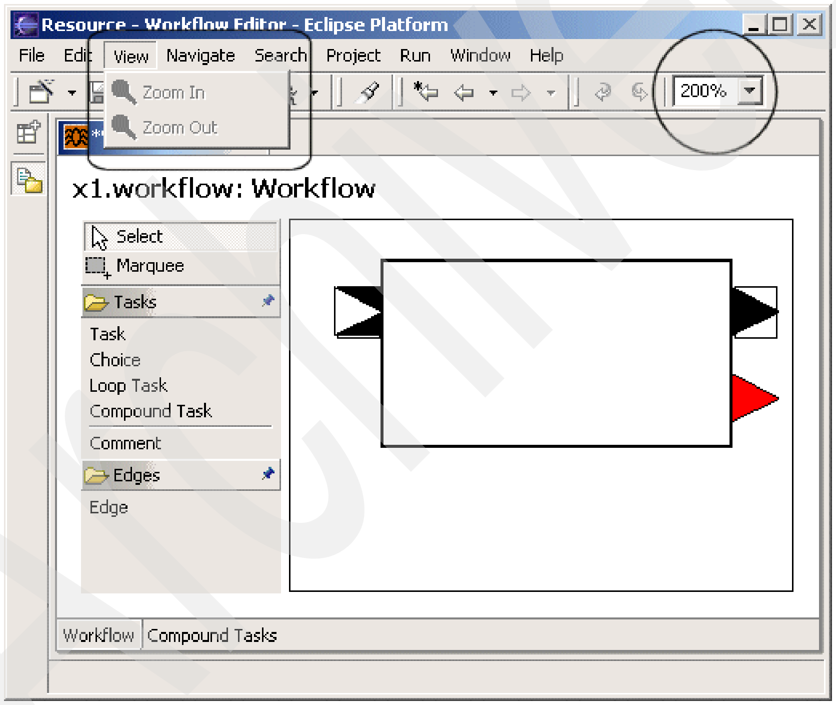


Figure 4-1 Zoom controls provided by GEF

The ZoomComboContributionItem creates a combo box interface that controls the zoom level for the active workbench part. It uses the IAdapter interface to locate the ZoomManager for the current IWorkbenchPart. Your editor should include code similar to the following, in Example 4-4, in order to allow its zoom level to be controlled by this mechanism.

Example 4-4 Returning the ZoomManager via the IAdapter interface

```
public Object getAdapter(Class type) {
      if (type == ZoomManager.class)
         return ((ScalableFreeformRootEditPart)getGraphicalViewer()
                  .getRootEditPart()).getZoomManager();
      return null;
   }
```

The actions supplied by GEF, ZoomInAction, and ZoomOutAction, enable you to easily add menu items or tool bar buttons that let the user zoom in or out, one level at a time. These actions are derived from the base class ZoomAction, which saves the current ZoomManager and registers itself as a ZoomListener. The actions call ZoomManager.zoomIn() or zoomOut() each time they are invoked, until the minimum or maximum zoom level is reached. They detect this by implementing the zoomChanged() method of the ZoomListener interface. They then update their enablement each time the zoom level is changed.

Adding zoom support

To add zoom support to your GEF application, you must first make your root EditPart either ScalableFreeformRootEditPart and ScalableRootEditPart. Then you only need to add some user interface access to set the current zoom level.

To add the zoom combo box, add it to the tool bar manager in the class that implements your application's ActionBarContributor, as shown in Example 4-5:

Example 4-5 Adding ZoomComboContributionItem to the tool bar

```
public void contributeToToolBar(IToolBarManager toolBarManager)
    {
        super.contributeToToolBar(toolBarManager);

   // other items added here...

        toolBarManager.add(new Separator());
        toolBarManager.add(new ZoomComboContributionItem(getPage()));
    }
```

To add the zoom actions, add code in your EditorPart-derived editor class that registers the actions with the action registry. See Example 4-6.

Example 4-6 Registering the zoom actions

```
IAction zoomIn = new ZoomInAction(zoomManager);
IAction zoomOut = new ZoomOutAction(zoomManager);
getActionRegistry().registerAction(zoomIn);
getActionRegistry().registerAction(zoomOut);
```

```
// also bind the actions to keyboard shortcuts
getSite().getKeyBindingService().registerAction(zoomIn);
getSite().getKeyBindingService().registerAction(zoomOut);
```

Finally, to add menu items for the zoom actions to the action bar menu, you add them to the MenuManager as shown in Example 4-7.

Example 4-7 Adding a menu for the zoom actions

```
public void contributeToMenu(IMenuManager menuManager) {
   super.contributeToMenu(menuManager);

   // add a "View" menu after "Edit"
   MenuManager viewMenu = new MenuManager("View");
   viewMenu.add(getAction(GEFActionConstants.ZOOM_IN));
   viewMenu.add(getAction(GEFActionConstants.ZOOM_OUT));
}
```

4.2.5 Decorating connections

The connections in Draw2d, and therefore also in GEF, are generally drawn using the PolylineConnection figure. In many cases you will want more than a plain line connecting the nodes in your GEF application, and the PolylineConnection class has built-in support for decorating connections that you can take advantage of.

You can decorate the ends of a PolylineConnection by specifying a RotatableDecoration for either the source, target, or both ends of the connection figure. The RotableDecoration interface is designed to allow a figure to rotate itself based on the position of a specified reference point. This allows the decoration to stay aligned with the line it is decorating as the line changes its angle, such as if one of the nodes that the line connects is moved.

The classic endpoint decoration is the arrowhead. Draw2d includes two implementations of RotatableDecoration; these are RotatablePolyline and RotatablePolygon. The default constructors for both of these classes create an arrowhead. These can be attached to a PolylineConnection by calling its setSourceDecoration or setTargetDecoration methods.

If you want a more customized decoration, you can call either classes' setTemplate method, passing it a list of points for your custom polyline or polygon figure. You can also control the size of your decoration by calling its setScale method.

PolylineConnections use a DelegatingLayout for their layout manager, meaning that its child figures must provide a constraint that is a subclass of Locator. When you place a decoration on a connection using setSourceDecoration or setTargetDecoration the these methods will automatically create an ArrowHeadLocator and set it as the constraint on the decoration. The arrowhead locator will ensure that the decoration is placed correctly on the ends of the connection.

In some applications, you may want to attach additional figures to a PolylineConnection. One example is to add a label to a PolylineConnection to display some annotation text. As the PolylineConnection has a DelegatingLayout manager, all you need to do is to create a Label figure and add it as a child of the PolylineConnection, and then set the constraint to position your label. The code shown in Example 4-8 places a label at the connection's midpoint.

Example 4-8 Placing a label in the center of a connection

```
PolylineConnection connection = new PolylineConnection();
Label connectionLabel = new Label();
connection.add( connectionLabel );
connection.getLayoutManager().setConstraint( connectionLabel,
                                             new MidpointLocator() );
```

4.2.6 Resource management

When implementing your GEF application, it is important that you pay attention to your application's usage of the underlying graphics system's resources. You must take care to manage your use of graphics objects, including images, bitmaps, colors, fonts, and so on. There are several techniques that can help to control your application's use of resources:

- Create static variables for resources that you will use frequently throughout the life span of your application. In the case of colors, the class org.eclipse.draw2d.ColorConstants provides many useful constants.
- Manage graphics resources that you want to create dynamically using the EditPart life cycle. Override the EditPart's activate() and deactivate() methods to handle creating and disposing of the EditPart's graphics resources.
- When your application has graphics objects that may not be used in every session, use a lazy loading scheme, deferring the object(s) creation until they are needed.
- If there are resources that are used across different parts of the application, consider implementing a cache that manages the objects. Different parts of your application can then share the same instance of these objects.

4.2.7 Feedback techniques

Visual feedback is an important part of a graphical editor's user interface. GEF allows for a number of techniques for proving feedback to the user:

- Changing the cursor when targeting parts to indicate whether the part supports the tool's operation. Similarly, the cursor graphic is changed to indicate where drag and drop operations are supported.
- Indication of a part's selection and focus state. Typically, a selected part should be clearly differentiated from unselected parts of the same type. This can be accomplished by enclosing the part with a selection figure or by changing the part's color or shape.
- Display of handles, graphical elements that visually indicate the targets for operations that allow the user to move or reshape a graphical object.

In GEF commands, most feedback effects are controlled by EditPolicies. In this section we examine in more detail the various EditPolicies that contribute visual feedback to the editor framework, and we discuss where these effects can be customized. EditPolicies that contribute feedback effects are subclasses of GraphicalEditPolicy. This base class provides access to the EditPart's figure. It also declares addFeedback() and removeFeedback() methods that draw feedback figures on the root figure's root feedback layer, LayerConstants.FEEDBACK_LAYER.

Note: All EditPolicies can disallow operations by returning null when requested to create a command. This will cause the tool to display a "disallowed operation" cursor. Similarly, commands which return false from their canExecute() will also cause this feedback.

DirectEditPolicy

Direct editing allows for visual editing of graphical elements by launching a cell editor in response to mouse clicks on the target EditPart. Creating a direct edit implementation is discussed in detail in "4.2.9, "Using direct edit" on page 158", and is demonstrated in our redbook sample application.

Customizing:

- You can expose different properties of your model to editing based on where the user clicked.
- You can respond differently to double-clicks.
- You can create custom cell editors

GraphicalNodeEditPolicy

This class provides visual feedback while creating or reconnecting connections. It works in conjunction with NodeEditPart-derived EditParts to provide a simulated connection while the connection is being dragged to a target. The NodeEditPart returns the best potential anchor point given the current mouse position. The simulated connection, drawn on the feedback layer, will snap to anchor proposed by the NodeEditPart. In typical implementations this will be the source anchor that is closest to the current mouse position.

Customizing:

- Override createDummyConnection() to return a customized figure to show the creation feedback, possibly by changing the color, or line style or weight.
- In your NodeEditPart-derived EditPart's implementation of getTargetConnectionAnchor or getSourceConnectionAnchor, apply additional filtering criteria to hide anchors that are not appropriate sources or targets for the current request. These anchors will then be ignored.
- Add target connection feedback. The default implementation has no visual effect for highlighting the target connection.

LayoutEditPolicy

This is a base class for EditPolicies that place their child EditParts using some type of LayoutManager. Subclasses should provide visual feedback that shows how the layout constraints will determine where a new element can be inserted. Key methods include:

- showLayoutTargetFeedback — This method gives visual feedback showing where the current operation will place the resulting figure. Subclasses will typically be constraining the placement of new figures to certain locations, and this feedback should make those constraints clear to the user. The figure returned by this method is effectively a type of cursor showing where the insertion point for the operation is located.
- getSizeOnDropFeedback — Shows the size that the new figure will assume if the drag operation is completed.

Customizing:

- The default implementation of showLayoutTargetFeedback does nothing. Implement this in a subclass to show the insertion point for new objects.
- The default implementation of getSizeOnDropFeedback() can be changed to use a different shape or color, and so on.
- Override getSizeOnDropFeedback() can be used if you want to provide visual feedback indicating the size of the new figure is constrained to some minimum and/or maximum size.

FlowLayoutEditPolicy

This EditPolicy is used in conjunction with EditParts whose figure uses a FlowLayout layout manager. This class provides an insertion point indicator which is a two pixel thick solid line.

Customizing:

- The getLineFeedback() method can be called to get the default line figure, and then some of its attributes can be modified, such as thickness, line style, or thickness. If you want to use a different figure altogether, then you will probably need to override the showLayoutTargetFeedback() to do the math required to locate and size your figure correctly.

SelectionEditPolicy

This is an abstract base for EditPolicies that provide visual feedback for the focus selection state of EditParts. Note that the feedback figures are drawn on the feedback layer (LayerConstants.FEEDBACK_LAYER). The methods in this class include:

- protected void showFocus()
- protected abstract void showSelection()
- showPrimarySelection()
- hideFocus()
- hideSelection()

The purpose of these methods is clear from the method names. Custom subclasses could be used to:

- Provide a non-standard focus or selection indicator, perhaps to conform to a non-rectangular figure.
- Provide an implementation for figures that can be selected but not moved.
- Render the selection indicator of the primary selection in a way that distinguishes it from the other selections.

SelectionHandlesEditPolicy

This EditPolicy supplies a specialization of the SelectionEditPolicy that supports selections with handles. Subclasses provide the list of Handles that should decorate the selected EditPart. For instance, GEF includes the following SelectionHandlesEditPolicy-derived subclasses:

- **BendpointEditPolicy:** This SelectionEditPolicy displays bendpoint handles when a Connection is selected.
- **ConnectionEndpointEditPolicy:** This EditPolicy displays handles on the ends of a Connection when it is selected to support disconnecting and reconnecting connections.

- **NonResizableEditPolicy:** This EditPolicy, which prevents resizing, surrounds the selected EditPart with a simple outline and places a small square in each corner that allows for dragging.
- **ResizableEditPolicy:** This class extends the NonResizableEditPolicy class by adding handles on each side of the selection rectangle to allow for resizing. Customize these classes to:
  - Indicate that resizing is limited to one dimension, or may be constrained to maintain the part's aspect ration. Implementing this would also require customizing the DragTracker to enforce these constraints.
  - Provide a selection effect that is more visually harmonious with unusually shaped parts.
  - Create a "lighter weight" visual for selection that may work better for showing the selection on small parts.
  - Use custom handles.

4.2.8 Palette-less applications

For many applications, it may be desirable to display a GEF viewer without the palette. For instance this may be useful when:

- The GEF view is read-only, but the user is allowed to select objects in the view in order to view their properties.
- The GEF application lays out its graphical objects automatically. The user is not allowed to add or rearrange these objects, but may be able to modify the model state by selecting objects, modifying their properties, and so on.
- There are a small number of tools in the application which do not justify the screen real estate that the palette would consume.

The EditDomain class is designed to integrate with the palette when it is present. There are two methods that are used to establish its connection to the palette:

- When the EditDomain's PaletteRoot is set by calling setPaletteRoot(PaletteRoot root), the EditDomain will then obtain its default Tool from the PaletteRoot
- Setting the EditDomain's PaletteViewer by calling its setPaletteViewer(PaletteViewer palette) method will cause it to register itself as a listener for tool selection changes in the palette.

Therefore, the first step in creating a palette-less application is to omit setting the EditDomain's PaletteRoot and PaletteViewer. This EditDomain will then return the SelectionTool when its getActiveTool() method is called. This can be achieved simply by constructing your editor as a subclass of org.eclipse.ui.parts.GraphicalEditor.

For some types of applications, the SelectionTool may be the only tool needed. In that case, there is no additional user interface needed to select tools, since the SectionTool is already selected by default. Other applications may have a need to change the active tool through some other user interface mechanism besides the palette. In these cases the EditDomain's setActiveTool(Tool tool) method can be called by the actions you create for the tools your application requires.

In this section we demonstrate how to add a toolbar button that sets the EditDomain's active tool. The action we create could also be used to make the tool available on the editor's context menu if desired. For the purposes of this example we use the Graphical Editing Framework logic editor example program. We modify it so that the tool to create a "Flow Container" is available as a button on the tool bar, as show in Figure 4-2.

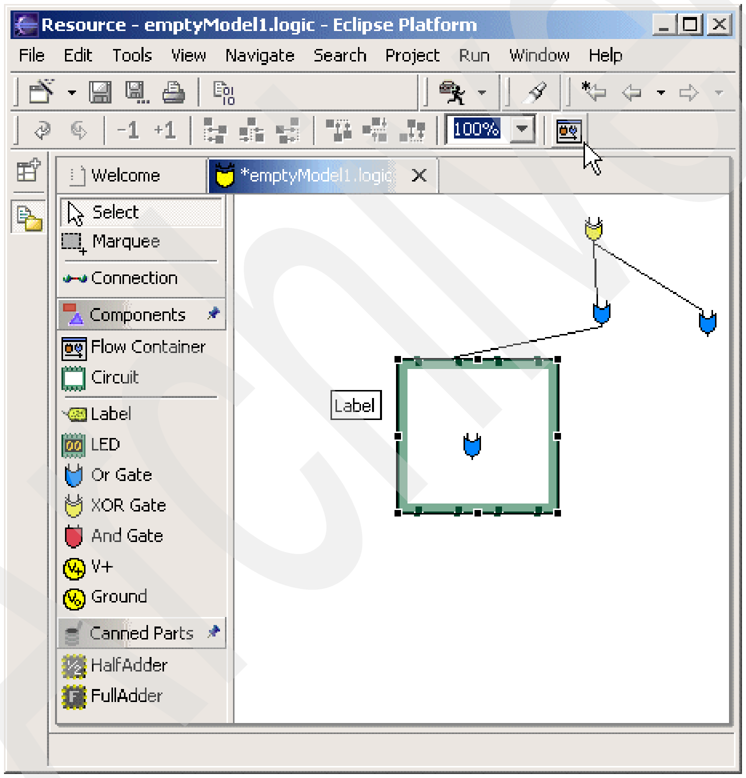


Figure 4-2 The Flow Container toolbar button

The steps required to make this change are as follows:

1. Create an Action for the tool:

Create the new class AddFlowContainerAction in the package org.eclipse.gef.examples.logicdesigner.actions. An abbreviated version of the Java code appears below in Example 4-9. Notice that this is a fairly generic-looking action implementation. The constructor includes calls to set the action's image, description, and title to the same values that were formally set in the palette entry. The run() method creates a new tool instance and uses it to call the editor's setActiveTool method, which in turn sets the EditDomain's active tool via its own setActiveTool method.

Example 4-9 The `AddFlowContainerAction` *class*

```
package org.eclipse.gef.examples.logicdesigner.actions;

import org.eclipse.gef.Tool;
import org.eclipse.gef.examples.logicdesigner.LogicEditor;
import org.eclipse.gef.examples.logicdesigner.LogicMessages;
import org.eclipse.gef.examples.logicdesigner.ToolActivationListener;
import org.eclipse.gef.examples.logicdesigner.model.Circuit;
import org.eclipse.gef.examples.logicdesigner.model.LogicFlowContainer;
import org.eclipse.gef.requests.CreationFactory;
import org.eclipse.gef.requests.SimpleFactory;
import org.eclipse.gef.tools.CreationTool;
import org.eclipse.gef.ui.actions.EditorPartAction;
import org.eclipse.jface.action.Action;
import org.eclipse.jface.resource.ImageDescriptor;
import org.eclipse.ui.IEditorPart;

public class AddFlowContainerAction extends EditorPartAction {
   private CreationFactoryfactory;
   private Tool tool;
   static public StringADD_CONTAINER = “add container”;


   /**
    * @param editor
    */
   public AddFlowContainerAction(IEditorPart editor) {
      super(editor);

      setDescription(
LogicMessages.LogicPlugin_Tool_CreationTool_FlowContainer_Description );
      setImageDescriptor( ImageDescriptor.createFromFile(Circuit.class,
“icons/logicflow16.gif”) );
      setText( LogicMessages.LogicPlugin_Tool_CreationTool_FlowContainer_Label
);

      factory = new SimpleFactory(LogicFlowContainer.class);
   }
   /* (non-Javadoc)
    * @see org.eclipse.jface.action.IAction#run()
    */
   public void run() {
      tool = new CreationTool( factory );

      ((LogicEditor)getEditorPart()).setActiveTool( tool );
   }

   protected boolean calculateEnabled() {
      return getEditorPart() != null;
   }

      protected void init() {
      super.init();

      setId( ADD_CONTAINER );
   }

}
```

2. Add the button to the action bar:

   Modify the class LogicActionBarContributor in the same package, org.eclipse.gef.examples.logicdesigner.actions, first adding the member variable addFlowContainerAction, as shown in Example 4-10. Note that the constructor's IEditorPart argument is passed as null. This is because there is no active editor when the buildActions method is called. This is the reason for storing a reference to the action. Its setEditorPart method is called whenever the active editor changes. This is done by overriding the setActiveEditor method, and after calling the super class implementation calling:

   ```
   addFlowContainerAction.setEditorPart( editor )
   ```

   The only other change we make in the LogicActionBarContributor class is adding the code in the contributeToToolBar method which appends a separator and the button for our AddFlowContainerAction action.

Example 4-10 Modifications to the LogicActionBarContributor class

```
   private AddFlowContainerActionaddFlowContainerAction;

/**
 * @see org.eclipse.gef.ui.actions.ActionBarContributor#createActions()
 */
protected void buildActions() {
   addRetargetAction(...);

// existing actions here...
```

```
    addFlowContainerAction = new AddFlowContainerAction( null );
    addAction( addFlowContainerAction );
}

public void contributeToToolBar(IToolBarManager tbm) {
    tbm.add(..);
    // existing tbm.add() calls here

    tbm.add(new Separator());
    tbm.add( getAction( AddFlowContainerAction.ADD_CONTAINER ) );
}

// override this so that the addFlowContainerAction instance can track the
current editor
    public void setActiveEditor(IEditorPart editor) {
        super.setActiveEditor(editor);

        addFlowContainerAction.setEditorPart( editor );
    }

}
```

At this point you should have a functioning tool bar button whose function is identical to the palette entry that adds a flow container.

Another user interface option for a palette-less application is to add commands to a menu. The AddFlowConterAction that we developed can also be used for this purpose. As shown in Figure 4-3, a new Tools menu item has been added which contains a submenu item that invokes our AddFlowConterAction action.

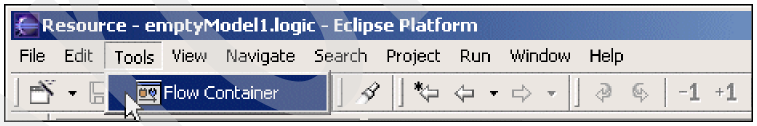


Figure 4-3 The AddFlowContainerAction added to the menu

The code fragment in Example 4-11 shows the changes you must make to the contributeToMenu method of the LogicActionBarContributor class.

Example 4-11 Adding a Tools menu in contributeToMenu

```
public void contributeToMenu(IMenuManager menubar) {
    super.contributeToMenu(menubar);

    // existing menu insertion code here...

    MenuManager toolsMenu = new MenuManager("Tools");
    toolsMenu.add(getAction(AddFlowContainerAction.ADD_CONTAINER));
    menubar.insertAfter(IWorkbenchActionConstants.M_EDIT, toolsMenu);
}
```

4.2.9 Using direct edit

Direct edit is a feature of the Graphical Editing Framework that allows you to open a cell editor on a selected EditPart using a mouse gesture. There are three ways for a user to invoke direct edit:

- Double-click the mouse on an EditPart.
- Click once on an EditPart that is already selected. This method is analogous to the Windows Explorer file name editing capability, which opens a text edit box if you click selected file name, allowing you to then edit the file's name.
- Press the F2 key when an EditPart is selected.

Figure 4-4 shows a simple Label figure when its cell editor is activated. This simple example can be seen in the GEF logic example application. However, direct edit can be used for much more visually complex EditParts in which clicking in areas of your EditPart invokes different cell editors, including dialogs.

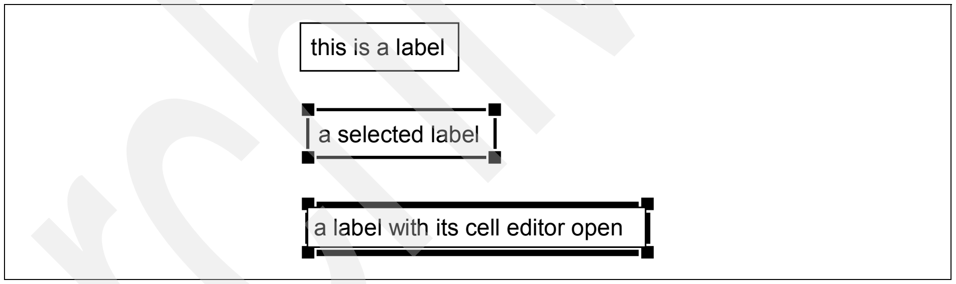


Figure 4-4 A Label figure showing the selected and cell editing states

The behavior of direct edit in your application is customizable. You specify what cell editor to open, and your application can determine this dynamically depending on criteria such as where the user clicked on your EditPart as well as the current state of your model, and so on. Direct Edit supports activating cell editors derived from org.eclipse.jface.viewers.CellEditor. Eclipse includes several useful subclasses that provide cell editors for booleans, combo boxes, text, and dialogs. By subclassing the DialogCellEditor, you have full flexibility to create dialogs that allow for the display or editing of your EditPart's properties.

This section outlines the steps you take to implement Direct Edit for one of your application's EditParts:

1. First modify your EditPart's createEditPolicies to install a new edit policy with the key EditPolicy.DIRECT_EDIT_ROLE, as shown in Example 4-12.

Example 4-12 Install the `DIRECT_EDIT_ROLE edit policy`

```
protected void createEditPolicies(){
   super.createEditPolicies();
   installEditPolicy(..);

   installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE, new
LabelDirectEditPolicy());
}
```

2. Add code in performRequest, as in Example 4-13, that invokes your DirectEditManager-derived edit manager class. The second argument to your DirectEditManager constructor determines what type of CellEditor will be created. For the third argument, you provide a class derived from:

   ```
   org.eclipse.gef.tools.CellEditorLocator
   ```

   This will calculates where you want the cell editor to appear within your EditPart. The example code shown here checks for a request type of RequestConstants.REQ_DIRECT_EDIT. The value of the request type allows you determine which of two mouse gestures was used to request the direct edit function. A request type of RequestConstants.REQ_DIRECT_EDIT indicates that the user single-clicked on a selected edit part, whereas the request type RequestConstants. REQ_OPEN indicates that the user double clicked on the edit part. You have the option to handle both request types as the same operation, ignore one of the types, or respond with different user interfaces for each type.

Example 4-13 Calling DirectEditManager.show() in performRequest

```
public void performRequest(Request request){
   if (request.getType() == RequestConstants.REQ_DIRECT_EDIT) {
      if(manager == null)
         manager = new LogicLabelEditManager(this,
            TextCellEditor.class, new
LabelCellEditorLocator((Label)getFigure()));
      manager.show();
   }
}
```

3. The EditPolicyclass you create must be a subclass of org.eclipse.gef.editpolicies.DirectEditPolicy.

   Your subclass must minimally provide implementations of the two abstract methods:

   ```
   protected abstract Command getDirectEditCommand(DirectEditRequest request);
   ```

This method constructs the command for the direct edit request. It should return a class, subclassed from org.eclipse.gef.commands.Command that updates your model with the results of the cell editing session. It should also support undo operations by caching the pre-edited state of you EditPart's model.

```
protected abstract void showCurrentEditValue(DirectEditRequest request);
```

The showCurrentEditValue method is called to update your EditPart's figure with the current value obtained from the request's cell editor. See Figure 4-5 for an example.

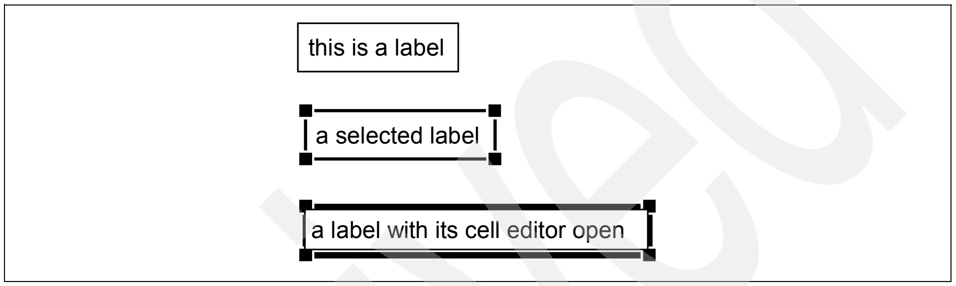


Figure 4-5 Current edit value

Tip: Normally during direct edit, your EditPart's selection handles will be shown, because the EditPart had to be selected in order to enter direct edit mode. This may not be aesthetically desirable. There are a couple of approaches to address this.

In the show() method of your DirectEditManager subclass you can save the current selection state:

```
List savedSelection = source getSelectedEditParts();
```

Then temporarily remove the selections:

```
source.deselectAll();
```

Then override the bringDown() so that the saved selection state is restored when the cell editor is closed.

A second option is to customize the graphics that indicate your EditPart's selected state, changing them to something that doesn't interfere visually with the cell editor.

4.2.10 Accessibility

Designing an accessible application is fundamentally about allowing for choice and flexibility in both input and output methods. An accessible application may receive input from the keyboard or serial port rather than the mouse. It needs to support accessibility clients that use sound, speech synthesis, or screen magnification to convey the output to the user.

GEF provides built-in support for accessibility, allowing you to create visual editors that can be controlled with little or no mouse interaction. The editor and palette are preconfigured to understand several keyboard navigation commands. Examples are the ability to select objects and palette entries, cycle through an object's selection handles, and drag or resize an object using the arrow keys:

- It supports the use of keyboard commands for users with limited dexterity.
- It provides annotations for the selected part, such as name, description, help text, and so on. that can be used by accessibility clients which may magnify or speak these strings for the user.
- It maps between the EditPart in focus and the accessibility client's view of the screen. This allows applications such as the Windows magnifier to track the user's actions within a GEF editor. This assists sight-impaired users.
- GEF supports autoscrolling, which allows the editor to scroll automatically to expose parts of a diagram that may be outside of the viewable area as the user drags their mouse to the edge of the view.

Tip: WIndows users can experiment with the accessibility features in the logic example by launching the Windows Magnifier application. Launch the Windows Magnifier by selecting **Programs -> Accessories -> Accessibility ->Magnifier**

GEF's accessibility implementation

In this section we describe the classes that implement GEF's accessibility support. We describe the roles they play and what you need to do to include accessibility in your GEF application.

Accessible EditParts

Accessible EditParts are able to participate in the accessibility support that is included in SWT and ultimately in the underlying operating system on which that your GEF application is running. Accessibility client applications can listen for selection changes in your GEF application and then obtain accessibility information about the selected EditPart via the EditPartViewer.

The AccessibleEditPart abstract class declares the methods that accessibility clients may use to interrogate your EditPart. These methods mirror the interface in org.eclipse.swt.accessibility.AccessibleAdapter, which defines the equivalent interface for SWT parts. The Javadoc in that class is a good source for documentation of the semantics of each of these methods. These methods allow your EditPart to enhance its accessibility by returning information such as its name, help string, keyboard shortcut, description, its selection and focus state, and by providing access to its child parts.

When you create an EditPart, you override the getAccessibleEditPart method in AbstractEditPart in order make your EditPart accessible. AccessibleGraphicalEditPart provides much of the default behavior needed by a custom EditPart. You will typically need to override the methods to return your part’s name, description, and so on.

AccessibleGraphicalEditPart

AccessibleGraphicalEditPart is an inner class of AbstractGraphicalEditPart that provides GEF’s implementation for the underlying SWT accessibility API, defined in org.eclipse.swt.accessibility package. This an abstract class, so EditParts supporting accessibility must provide a concrete subclass that is returned when the AbstractEditPart.getAccessibleEditPart() is called.

Accessible handles

Making a handle accessible requires that the handle provide a single point, in absolute coordinates, at which it can be selected. Keyboard navigation can then use this coordinate when selecting the handle, effectively simulating a mouse click at that location. Accessible handles are obtained from the EditPolicies that are responsible for handle management, such as subclasses of SelectionEditPolicy.

The AccessibleHandleProvider interface is used to collect a list of accessible handles for a Handle or EditPart. The AccessibleGraphicalEditPart implements this interface through its IAdaptable implementation. It collects a merged list of all the accessible handles contributed by its EditPolicy instances which also implement the AccessibleHandleProvider interface. Ultimately each Handle interface’s getAccessibleLocation method returns the coordinate that indicates the location of its accessible handle. The AbstractHandle class provides most handles with a default implementation of this method that returns the center point of the handle. Other handle types can override this as appropriate.

The SelectionHandlesEditPolicy is an abstract class that is adaptable to an AccessibleHandleProvider, providing accessibility for subclasses that use GEF’s default handles. If you design your own handles, you will need to provide an implementation of the getAccessibleLocation that returns a point inside your handle.

Accessible anchors

Accessible anchors work similarly to accessible handles. An EditPart provides an implementation of the AccessibleAnchorProvider interface by implementing the IAdaptable interface. The AccessibleAnchorProvider interface contains methods to return a list of source and target anchor locations. These points will be used to programmatically simulate a mouse event at that location. The targeting tool will then provide the same targeting behavior and feedback as if a mouse was used.

To implement this capability in your own EditParts, you will need to traverse all the ConnectionAnchor-derived children of your EditPart's parent figure, and return an appropriate point for each one.

AbstractTool

The AbstractTool class serves as the base class for contains the state machine which interprets accessible actions such as translating arrow keys into drags, and so on. Pressing the Enter key commits a drag

SelectionTool

When an edit part is selected, the SelectionTool's accessibility support enables the user to traverse the EditPart's available selection handles, select one, and perform drag operations all by using the keyboard. The keyboard commands supported by this class are summarized in Table 4-2.

Table 4-2 Keyboard commands provided by the SelectionTool class

Key	Action
Period	Select next handle
'>'	Select previous handle
Left Arrow	Drag left
Right Arrow	Drag right
Up Arrow	Drag up
Down Arrow	Drag down
Enter	Commit the drag operation
Esc	Abort the drag operation

ConnectionCreationTool

The keyboard handling in this class allows the user to indicate the start and end of connections using the Enter key. The user can cycle through the available anchor points of accessible EditParts by using the arrow keys. The tool will snap the connection to the next available anchor.

GraphicalViewerKeyHandler

This key handler class provides keyboard-based navigation for the GraphicalViewer. Table 4-3 lists the key bindings provided by the GraphicalViewerKeyHandler class. Note that SHIFT and CTRL keys can be used to modify the navigation keys. Pressing the CTRL key will cause the focus, rather than the selection, to move. Pressing the SHIFT key while using one of the navigation keys will extend the selection.

Table 4-3 Navigation key bindings defined in GraphicalViewerKeyHandler

Key	**Action**
SPACE	Selects
LEFT_ARROW	Navigates to EditPart on left
RIGHT_ARROW	Navigates to EditPart on right
UP_ARROW	Navigates to EditPart above
DOWN_ARROW	Navigates to EditPart below
'/' or '?'	Navigates to EditParts's next connection
'\' or '|'	Navigates to EditPart's previous connection
ALT + DOWN_ARROW	Navigates into a container node
ALT + UP_ARROW	Navigates out of a container node

PaletteViewerKeyHandler

This class, the keyhandler for the palette, supports keyboard commands in the palette.It supports moving between palette entries and moving into and out of palette drawers. The commands are summarized in Table 4-4.

Table 4-4 Arrow key bindings to palette navigation

Key	**Action**
LEFT_ARROW	If the focus is on an expanded drawer, then collapse it, otherwise sets focus on the drawer.
RIGHT_ARROW	If the focus is on a collapsed drawer, then it expands it. If the focus is on an expanded drawer, then it moves into it.
UP_ARROW	If the focus is inside a drawer, it sets the focus on the drawer.
DOWN_ARROW	It moves to the next container.

5

Using GEF with EMF

In this chapter, we discuss developing graphical editors based on EMF and GEF, and we provide examples of how to use the two frameworks together. We also discuss how to use JET to assist in developing a GEF-based editor from an EMF model.

Note: The sample code we describe in this chapter is available as part of the redbook additional material. See Appendix A, “Additional material” on page 225 for details on how to obtain and work with the additional material. The sample code for this chapter is provided as Eclipse projects that can be imported into your Eclipse workbench.

Each major section of this chapter has a matching Eclipse project in the additional material. Also, be sure to import the appropriate model project for the editor project you want to work with. For example, to work with the NetworkEditor project, you need to also import the NetworkEditorModel project. Some of the sample projects in this chapter also expect that you have the SAL330RWorkflowModel project in your workspace. You may have created this project by working through the examples described in Chapter 1, “Introduction to EMF” on page 3, or you can import this from our redbook sample material.

5.1 Overview

As GEF is based on an MVC architecture, every GEF-based application uses a model to represent the state of the diagrams being created and edited. GEF allows you to use any objects as model objects within your application, however, using an EMF model provides some advantages over using arbitrary objects:

- You can use EMF's code generation facilities to produce consistent, efficient and easily customizable implementations of your model objects. If your model evolves during development, you can regenerate the code to reflect changes to the model, while preserving your customizations.
- The MVC architecture used by GEF relies on controllers that listen for model changes and update the view in response. If you use an EMF model, notification of model change is already in place, as all EMF model objects notify change via EMF's notification framework.
- The implementations generated for your model objects ensure that the model remains consistent, for example, when a reference is updated, the opposite reference is also updated.
- EMF provides support for persisting model instances, and the serialization format is easily customizable.
- Your applications can use the reflective API provided by EMF to work with any EMF model generically.

Although we can generate EMF.Edit-based editors from EMF models using the org.eclipse.emf.codegen.ecore plug-in, these editors use JFace viewers, such as the TreeViewer to display model instances, and typically provide a view that has a one-to-one correspondence with the model. Sometimes we may wish to create editors where the view is more loosely coupled with the model. This is often the case when we want to use a graphical notation that may hide some of the detail of the underlying model objects, or may impose additional or a different structure to the model, for visualization purposes.

We can think about using GEF and EMF together from two different perspectives; using an EMF model within a GEF application, and augmenting EMF.Edit-based editors using GEF. In this book, we focus on the first perspective only, due to time constraints. The second approach deserves a book of its own, as integrating an EMF.Edit-based editor with GEF provides its own challenges. For an example of an application that uses GEF and EMF.Edit together, take a look at the Jeez report designer, available from:

`http://jeez.sourceforge.net`

5.2 Using an EMF model within a GEF-based application

This section describes how to use model interfaces and implementations generated from an EMF model as the model within a GEF-based application. This is the approach that we have used for our sample application, described in Chapter 7, “Implementing the sample” on page 203. We assume that you have read Chapter 3, “Introduction to GEF” on page 87and that you have a basic understanding of how an arbitrary (not necessarily EMF-based) model is usually integrated into a GEF-based application. Because GEF can use almost any type of model, integrating an EMF model into an editor is much the same as integrating any other sort of model into a GEF-based application. When tying our model into our editor, we can take advantage of mechanisms provided by EMF for notification, reflection and serialization.

We use a simple application to illustrate our approach for using GEF and EMF together. The application is an editor that allows us to define networks consisting of nodes that may be linked together. We discuss how to implement an example application based on the model shown in Figure 5-1.

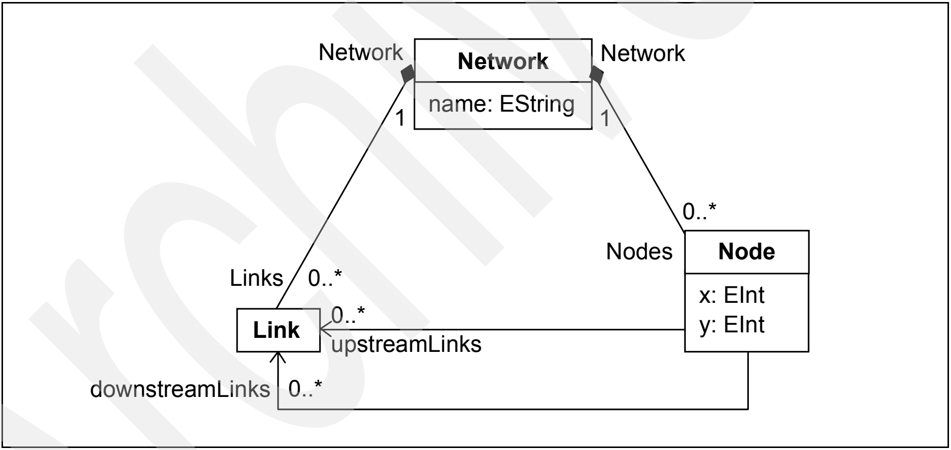


Figure 5-1 Simple Network Model

5.2.1 Mapping from the model to the graphical representation

In a GEF-based editor, EditParts are the controllers that bridge objects from the model and their representation in the view, however, there does not have to be a one-to-one correspondence between model objects and EditParts. Hence, the first step in developing our application is to decide which EditParts to provide to represent objects from the model.

Mapping to EditParts

The first EditPart that we consider is the contents EditPart. This is the part that contains all of the other EditParts, that is, it represents a diagram that is edited within our editor. In our example, the contents EditPart corresponds to the Network class.

In general, if a model has a top-level element that contains all other model objects, as is the case with the NetworkModel, and the WorkflowModel used for our sample application, then the contents EditPart corresponds directly to that container. For models that do not have a top-level container, you can think of the contents EditPart as corresponding to the contents of a ResourceSet that contains model objects, rather than corresponding directly to an EObject from the model.

GEF provides two base implementations of EditPart that are used in graphical viewers; AbstractGraphicalEditPart and AbstractConnectionEditPart. We can subclass either of these classes for the EditParts that correspond to the objects from our model. For the NetworkEditor example, we subclass AbstractGraphicalEditPart as NetworkEditPart, our contents EditPart. As an AbstractEditPart, NetworkEditPart has methods getModel() and setModel() for getting and setting the corresponding model object with the EditPart. We implement NetworkEditPart so that the Network associated with the part is supplied to the constructor, as shown in Example 5-1.

Example 5-1 NetworkEditPart constructor

```
public NetworkEditPart(Network network) {
        setModel(network);
}
```

Tip: When basing an editor on an EMF model, most of the objects returned by the getModel() methods of the EditParts will be EObjects, however, you can use *any* object as the model for an EditPart. This is one way to provide EditParts that do not correspond directly to EObjects from the EMF model.

We implement the getModelChildren() method for NetworkEditPart, as shown in Example 5-2. This method returns all of the objects directly contained by the EditPart's model object, in this case, all of the Nodes contained by the Network. This method needs to be implemented by any EditPart that contains children EditParts.

Example 5-2 NetworkEditPart's getModelChildren() method

```
protected List getModelChildren(){
   return getNetwork().getNodes();
}
```

Usually, the containment hierarchy of EditParts mirrors the containment hierarchy present in the model, so the getModelChildren() method often returns the objects contained by the EditPart's model object. When this is the case, we can call the methods generated by EMF for each containment EReference to construct a List of all contained objects, as we also see in the sample application in Example 7-3 on page 210. However, if your EditPart containment hierarchy differs from your model hierarchy, remember that this method needs to return *all* of the objects corresponding to children EditParts, and *only* the objects corresponding to children EditParts.

How you choose to map the other objects from your model to EditParts will depend on how each object is to be represented graphically. The graphical representation for some objects may be simple, but for others, it may be composed of multiple graphical components. These components will either be implemented as child EditParts, or as children of the figure that represents the model object in the view. An example of an appropriate use of child figures is to represent object attributes with simple string or number values. An EditPart is typically used to represent something with which the user interacts, which can be selected and manipulated in its own right.

Note: While it is usual for an EditPart to have a direct correspondence to a single object in the model, this is not a requirement. You can choose to use more than one EditPart to represent an object from the model, use a single EditPart to represent multiple model objects, or even create EditParts that have no direct correspondence to model objects. See "Indirect mappings" on page 171 for examples.

One approach for mapping from the model is to provide an EditPart for each class, and then decide if you need any extra EditParts to represent its features. For the NetworkEditor, we use an AbstractGraphicalEditPart for both the Network and the Node class. Objects referenced by a containment reference are represented as child EditParts, that is, NetworkNodeEditParts are children of NetworkEditPart, and Links between Nodes are represented as LinkEditParts, which subclass AbstractConnectionEditPart.

In addition to implementing the EditParts, we also subclass EditPartFactory as GraphicalEditPartFactory. It is from this class that EditParts are created and associated with their corresponding model objects, as shown in Example 5-3.

Example 5-3 The createEditPart() method

```
public class GraphicalEditPartsFactory implements EditPartFactory{
   public EditPart createEditPart(EditPart context, Object obj){
      if(obj instanceof Network)
         return new NetworkEditPart((Network)obj);
      else if(obj instanceof Node)
         return new NetworkNodeEditPart((Node)obj);
      else if (obj instanceof Link)
         return new LinkEditPart((Link)obj);
      return null;
   }
}
```

Figures

Each EditPart has a corresponding figure which is created and returned by the EditPart's createFigure() method. For each EditPart that you implement, you will need to decide if you also need to provide a specialized figure to represent that EditPart. EditParts that have simple graphical representations can often be represented using one of the figures provided by Draw2D, such as a label or a shape. We use a layer as the figure for NetworkEditPart in the NetworkEditor, as Example 5-4 shows.

Example 5-4 NetworkEditPart's createFigure() method

```
protected IFigure createFigure(){
   FreeformLayer layer = new FreeformLayer();
   layer.setLayoutManager(new FreeformLayout());
   layer.setBorder(new LineBorder(1));
   return layer;
}
```

EditParts with visual representations consisting of multiple parts will usually require a custom Figure to contain all of the child figures. We implement NodeFigure to represent NetworkNodeEditParts. The id attribute of each Node is represented as a child Label of the NodeFigure, as shown in Example 5-5.

Example 5-5 NodeFigure with child Label for id attribute

```
public class NodeFigure extends Ellipse {
   protected EllipseAnchor incomingConnectionAnchor;
   protected EllipseAnchor outgoingConnectionAnchor;
   protected Label label;
   protected XYLayout layout;
   public NodeFigure() {
      layout = new XYLayout();
      setLayoutManager( layout );
      setBackgroundColor( ColorConstants.white );
      setOpaque( false);
      incomingConnectionAnchor = new EllipseAnchor(this);
      outgoingConnectionAnchor = new EllipseAnchor(this);
      label = new Label(" ");
      add(label);
   }
   public void setId(String id){
      label.setText(id);
   }
   ...
}
```

Indirect mappings

There are few restrictions on how you may map your model objects to EditParts; however, if you decide to map a single model object into multiple EditParts, you will need to contain those parts by a (possibly invisible) parent EditPart that corresponds to the model object. The reason why your model object can only correspond to a single EditPart is because the viewer uses a java.util.Map to map model objects to their corresponding EditParts, using the model object as the key. If grouping the parts is not appropriate for the graphical representation that you have chosen to use, this usually indicates a mismatch between the model and the graphical representation, and you may need to reconsider the representation or refactor your model.

Using this approach, when a new object is created, your EditPartFactory implementation can simply return an instance of the parent EditPart as the result of the createEditPart() method. Then the getModelChildren() method of the parent EditPart can construct appropriate Java objects for the children that only contain the data that is relevant to each child EditPart. Usually such objects would represent some subpart of the EObject, such as a feature or collection of features. Grouping the EditParts within a parent can make it easier to update the parts in response to model change, as only the parent EditPart needs to listen for changes to the model object and can then selectively update its children EditParts. We discuss how EditParts listen for and respond to model change in 5.2.4, “Reflecting model changes” on page 175.

It is common for multiple model objects to be mapped to a single EditPart in the graphical representation, particularly where containment relationships exist in the model. An example is provided in the sample application and discussed in Chapter 7, “Implementing the sample” on page 203, where ports that are contained by a WorkflowNode are represented as child figures rather than as separate EditParts.

Sometimes, you may wish to implement EditParts that do not directly represent an instance of a class from the model, for example, EditParts that represent state that is derived from model objects. In this case, you still need to provide an object to the EditPart via the setModel() method, but it does not have to be an EObject from your model.

A common example of an EditPart that does not have a direct correspondence to a class from the model is a ConnectionEditPart used to represent a reference. In the following example, we demonstrate how you can implement this mapping. Figure 5-2 shows a modified version of the NetworkModel. In this model, there is no Link class to represent the links between nodes explicitly. Instead, the references upstreamLinks and downstreamLinks are used to maintain the relationships between nodes.

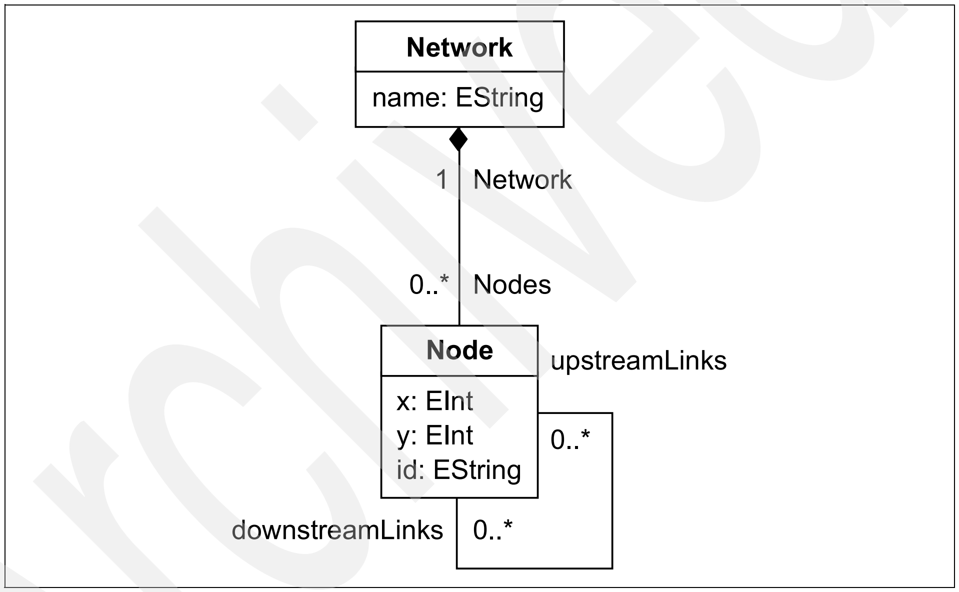


Figure 5-2   NetworkModel without the Link class

Each LinkEditPart still needs to correspond to an object so that it can be looked up in the EditPartRegistry of the viewer whenever refreshSourceConnections() or refreshTargetConnections() is called in NetworkNodeEditPart, to create or update the connected links. The corresponding object can be any Java object, and in our example, we use a String that identifies the source and target of the LinkEditPart as its model object.

We modify the NetworkEditor as follows:

- Modify LinkEditPart so that it takes a String argument in the constructor and uses that String as the model object instead of a Link.

- Modify GraphicalEditPartFactory so that it provides a String to the LinkEditPart constructor when creating a new LinkEditPart.
- Remove references to Link from the ModelCreationFactory, and use null instead of a ModelCreationFactory in the NetworkPaletteRoot when creating the tool entry for link creation.
- Provide new implementations of getModelSourceConnections() and getModelTargetConnections() in NetworkNodeEditPart, to return the Strings that we use to identify the links. The String that we use to identify a Link is constructed using toString() on the source and target Nodes. An example of how we construct the identifying String is shown in Example 5-6.

Example 5-6 Returning derived objects from getModelSourceConnections()

```
protected List getModelSourceConnections() {
   Vector s = new Vector();
   Iterator i = getNetworkNode().getDownstreamLinks().iterator();
   Node n;
   while(i.hasNext()){
      n = (Node)i.next();
      s.add(getNetworkNode().toString() + "->" + n.toString());
   }
   return s;
}
```

Remember that you can use any Java object as the model for your EditParts. You can create parts with more complex derivations from the model by providing your own objects to represent those values. You should only use this technique for transient or derived values, as any data that is not stored in the model will not be persisted by default.

As we have seen in Example 5-6, you can then associate your custom objects with their corresponding EditParts from within getModelChildren(), getModelSourceConnections() or getModelTargetConnections(), depending on whether you are using child or connection EditParts to represent those objects.

Fitting the graphical representation to the model

Sometimes you may wish to modify your model so that it corresponds more closely to the graphical representation that you choose to use in your GEF-based application.

GEF assumes that all of the information that you need to store about the diagrams that you are editing is represented in the model. For this reason, you may also need to augment your model to include information such as co-ordinates or dimensions.

There are several approaches for constructing the model that you will use in your application (the view model) from your original model (the business model):

- Create a modified version of the original model, with the additional view information added directly to your original model objects. This approach is straightforward to implement, however, the correspondences between the view model and the business model are not explicit, as there is no tangible link between the two models. This is the approach that we use in the sample application.
- Use two separate models, the business model, and a new model for view-specific information. This is the approach used by the Omondo UML Editor.
- Use modelling techniques to make the link between the view and business model explicit. For example, create a new package that imports the business model, and subclasses all of the business model objects, adding the necessary view information in the subclasses.

We discuss examples of the latter two approaches in Chapter 1, "Introduction to EMF" on page 3.

5.2.2 Displaying properties

In the sample application, and also in the NetworkEditor example, we use reflection to construct property sheets for our model objects."Register the EditPart as a property source:" on page 205 describes the implementation in more detail.

5.2.3 Support for editing the model

Changes to the model are made via commands. Remember that commands only know about model objects. It is the responsibility of EditParts to listen for changes made to model objects by commands and update the view accordingly. When you are using a hand-coded model, usually when you use commands to change the model, you know exactly how the changes effect the state of the model. An important thing to note when using an EMF model is that changes that are made to the model sometimes have consequences that you may not take into consideration when implementing undo functionality.

For example, if you remove a reference to an object, the reference back from the opposite will also be removed. If you delete an object that contains others, they will also be deleted. This is because the EMF types are implemented to ensure that the model remains consistent. When you are using EMF for the first time, these behind the scenes changes are convenient, as they save you from having to enforce these constraints manually; however, they can come as a surprise if you are not aware of how the underlying objects behave.

If you are not expecting such changes, you can run into problems, for example when undoing multiple changes in the sample application, if you delete an edge and then the node it was connected to, the port that the edge was connected to will be deleted with the node, so you need to store enough information about the ports and the edge, so that the edge can find the right ports to reconnect to if those deletions are undone. When you are working with a known model it is not usually a problem to know how much information you need to store to facilitate undo, however, if you are working with models generically using the reflective API, the safest way to ensure that undo restores the model exactly how it was before the change, is to snapshot the model each time a command executes. You can either represent each snapshot as a separate serialization, or use diffs to reconstruct model state.

5.2.4 Reflecting model changes

EditParts are the representation of model objects in the editor, hence they need to listen for any changes that are made to their corresponding objects in the model and update their representation accordingly. EMF provides a Notification framework: Every EObject is a Notifier that can be adapted (or observed) by any class that implements the Adapter interface provided by org.eclipse.emf.common.notify. You will notice that in the implementation classes generated from an Ecore model, whenever the state of the object is modified by setting or unsetting a feature or adding or removing contained objects, any adapters are notified by a call to eNotify() that provides details of the change. Each Adapter receives these notifications via the notifyChanged() method. The EditParts in our Network editor adapt their corresponding model objects and implement notifyChanged() to respond accordingly to the changes.

Each EditPart adds itself to the adapters of any objects that it represents in its activate() method, and removes itself from the adapters of those objects in its deactivate() method. Example 5-7 shows how NetworkEditPart adds itself as an adapter of the Network it represents in its activate() method.

Example 5-7 The activate() method of NetworkEditPart

```
public void activate(){
   if (isActive())
      return;
   ((Notifier)getNetwork()).eAdapters().add(this);
   super.activate();
}
```

Each EditPart also implements the notifyChanged() method. Depending on what has changed, the EditPart may need to update its children, connections or visual representation to reflect the changed state of the model, by calling the refreshChildren(), refreshSourceConnections(), refreshTargetConnections() or refreshVisuals() methods. We outline the methods that we might typically call in our implementation of the notifyChanged() method of an EditPart, in response to the different types of Notification, in Table 5-1.

Table 5-1 Typical response to change Notifications

Notification type	Circumstances	Response
ADD ADD_MANY	Added objects are represented as a child EditPart	refreshChildren()
	Added objects are represented as connected ConnectionEditParts	refreshSourceConnections() or refreshTargetConnections()
REMOVE REMOVE_MANY	Notifier object is represented by a child EditPart	refreshChildren()
	Notifier is represented by a connected ConnectionEditPart	refreshSourceConnections() or refreshTargetConnections()
`SET` `UNSET`	Notifier is the model object of this EditPart	refreshVisuals()

When the graphical representation corresponds closely to the model, as is the case in our Network editor example, the notifyChanged() method is straightforward, as we see in Example 5-8. In this case, the EditPart needs only to refresh its children when the contents of the Network that it represents change, or to refresh its visual representation when a feature of the Network is changed.

Example 5-8 NetworkEditPart refreshing children EditParts

```
public void notifyChanged(Notification notification) {
   int type = notification.getEventType();
   switch( type ) {
      case Notification.ADD:
      case Notification.ADD_MANY:
      case Notification.REMOVE:
      case Notification.REMOVE_MANY:
         refreshChildren();
         break;
      case Notification.SET:
         refreshVisuals();
         break;
   }
}
```

If an EditPart does not use all of the features of the model object in its visual representation, additional code could be added so that refreshVisuals() is only called when features that are visualized change. If the visualization is made up of many parts, you may want to provide methods that will only refresh specific parts of the view, and use them from notifyChanged() instead of refreshVisuals().

Refreshing source or target connections is similar to refreshing children. For example, whenever a NetworkNodeEditPart receives notification of changes to its upstreamLinks or downstreamLinks features, it refreshes the connections that represent that link, as we see in Example 5-9.

Example 5-9 NetworkNodeEditPart refreshing connected EditParts

```
public void notifyChanged(Notification notification) {
   int featureId = notification.getFeatureID( NetworkPackage.class );
   switch( featureId ) {
      case NetworkPackage.NODE__UPSTREAM_LINKS:
         refreshTargetConnections();
         break;
      case NetworkPackage.NODE__DOWNSTREAM_LINKS:
         refreshSourceConnections();
         break;
      default:
         refreshVisuals();
         break;
   }
}
```

In summary, EditParts need to know whenever their corresponding model objects change, so that they can update their children, connections, and visuals appropriately. We can implement this by making each EditPart an Adapter on its corresponding model object, and this works well if the model corresponds closely to the graphical representation, that is, if most EditParts correspond directly to model objects, and the EditPart containment hierarchy mirrors the hierarchy in the model. If the correspondence between objects from your model and the EditParts that you choose to represent them is not so close, you will need to customize this approach. You may wish to consider the following guidelines:

- If an EditPart represents multiple objects from the model, that EditPart needs to listen for changes to all of those model objects. If the group of objects that it represents can change, it may be necessary for the EditPart to also add or remove itself from the adapters of those objects in response to the objects being added or removed, in notifyChanged(). The sample application provides an example of this for WorkflowNodeEditPart, which represents WorkflowNodes and their Ports and which is described in 7.2.2, “Tracking model events in the editor” on page 207.

- For EditParts that contain or connect to EditParts that do not correspond directly to objects contained by the parent EditPart's model object, the EditPart must listen for changes to all model objects that contribute to the state of objects represented by its children, and then update its children or connections whenever those objects change.
- EditParts that do not directly correspond to model objects do not need to implement the Adapter interface as they rely on their parent to refresh them.

5.2.5 Loading and saving model instances

2.3, "Model instances and serialization" on page 64 demonstrates how to serialize model instances via resources. In the NetworkEditor example, we use the default XMI serialization provided by XMIResource, however the way that we load and save models from the editor is the same regardless of the type of resource that we choose to represent our network instances.

We provide a class NetworkModelManager, which manages an XMIResource containing a network, and which provides methods that create, load and save that resource. Using a different serialization would simply require another implementation of the NetworkModelManager class that used a custom resource type and factory, instead of XMIResource.

The NetworkEditor class uses NetworkModelManager, creating one per file that is open in the multi-page editor, and provides methods to get and save the Network instance currently being edited via the NetworkModelManager.

Example 5-10 shows how the editor uses the NetworkModelManager instance to get a network from a file opened in the editor. This method is called when the editor is initialized from its init() method.

Example 5-10 Getting an instance from the ModelManager

```
private Network create(IFile file) throws CoreException{
   Network network = null;
   modelManager = new NetworkModelManager();
   if (file.exists()){
      try{
         modelManager.load(file.getFullPath());
      }
      catch (Exception e)
      {
         modelManager.createNetwork(file.getFullPath());
      }
      network = modelManager.getModel();
      if (null == network){
         throw new CoreException(
            new Status(
                IStatus.ERROR,
                NetworkEditorPlugin.PLUGIN_ID,
                0,
                "Error loading the worklow.",
                null));
        }
    }
    return network;
}
```

The editor uses a similar mechanism to save Networks via the NetworkModelManager, using the following method call: `modelManager.save(file.getFullPath());`

When the save() method is called, the NetworkModelManager calls the save() method on the Resource containing the Network, and it is serialized into an XMI document and saved to the path supplied.

5.2.6 Putting it all together

We complete the editor by integrating the model-specific code into a multi-page editor that we package as a plug-in.

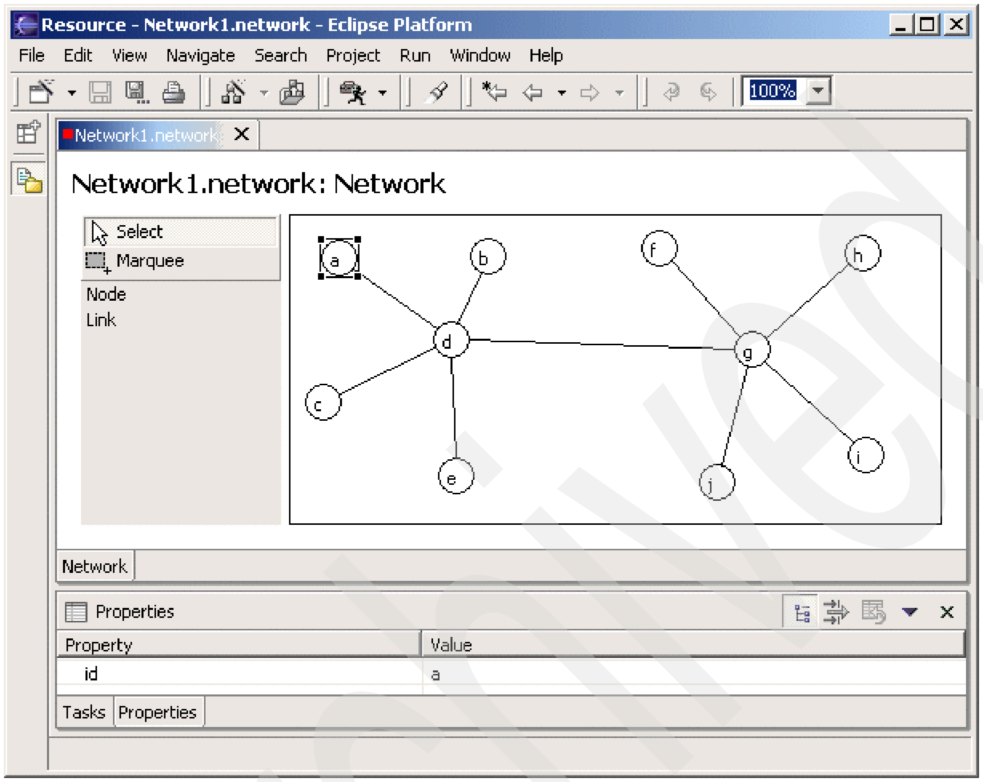
We subclass MultiPageEditorPart as NetworkEditor. This class sets up commands, actions and the palette used in the editor. As this is standard GEF, and is very similar to the code described for the sample application, we do not describe these details of the NetworkEditor implementation here.

Finally we hook our model and corresponding EditParts into the viewer when we create the GraphicalViewer within the NetworkPage class, as shown in Example 5-11.

Example 5-11 Hooking the model into the GraphicalViewer

```
private void createGraphicalViewer(Composite parent){
    viewer = new ScrollingGraphicalViewer();
    ...
    // initialize the viewer with input
    viewer.setEditPartFactory(new GraphicalEditPartsFactory());
    viewer.setContents(getNetworkEditor().getNetwork());
}
```

Figure 5-3 shows a screen capture of the graphical view of the completed NetworkEditor application.


Resource - Network1.network - Eclipse Platform
File Edit View Navigate Search Project Run Window Help
100%
Network1.network
Network1.network: Network
Select
Marquee
Node
Link
a
b
c
d
e
f
g
h
i
j
Network
Properties
Tasks
Properties

Property	Value
id	a

Figure 5-3 The NetworkEditor

5.3 Using JET in GEF-based editor development

In this section, we discuss how JET may be used to speed up development of an editor based on EMF and GEF.

We provide an example that generates skeletons for some classes that are used in a GEF editor, from a model. We can use the technique described in this section regardless of whether we take the approach described in 5.2, "Using an EMF model within a GEF-based application" on page 167, or whether we are using GEF to augment an EMF.Edit-based editor. You can flesh out the generated code into an application as described in Chapter 3, "Introduction to GEF" on page 87.

When developing your GEF-based application based on an EMF model, you will notice that you are usually creating many similar classes, for example, often you will create NodeEditParts for most of the classes in your model, perhaps using ConnectionEditParts for some of them. Often you will use a custom figure for your NodeEditParts. In the following example, we use JET templates to generate EditParts and Figures from classes in our model. This is a very basic example, to illustrate concepts. We do not provide a complete example due to time constraints, as the templates required to generate more complete implementations would be non-trivial. You would probably want to provide more detail in the templates if you wanted to generate EditParts specific to your application.

Refer to the *JET Tutorial, Part one* for an introduction to using JET. We use a similar process to the example described to generate our skeleton EditParts and Figures from the WorkflowModel. We take the following steps:

1. To begin with, we create a project, and add a JET Nature to the project from the right-click context menu. This sets up the template directory.
2. In the template directory, we create a new file NodeEditPart.javajet.
3. We edit the NodeEditPart to create all of the required methods. We base the content of the template on the NetworkNodeEditPart from the NetworkEditor described in 5.2, “Using an EMF model within a GEF-based application” on page 167. Example 5-12 shows an excerpt from the template. Our example only really uses the name of the class so far to generate the skeleton, however you could use methods on the EClass to get more detail. For example, you might want to generate a skeleton notifyChanged() method with a switch that selected from all of the features of the class.

Example 5-12 NodeEditPart template

```
<%@ jet package="com.ibm.itso.sal330r.codegen"
imports="org.eclipse.emf.ecore.*" class="NodeEditPartTemplate" %>
<%EClass eClass = (EClass) argument;%>
... imports ...
<%String name = eClass.getName();%>
public class <%=name%>EditPart
   extends AbstractGraphicalEditPart
   implements NodeEditPart, Adapter
{
   private IPropertySource propertySource = null;
   private Notifier target;

    public <%=name%>EditPart(<%=name%> o)
    {
      setModel(o);
    }
   public <%=name%> get<%=name%>() {
    return (<%=name%>)getModel();
    }
   /* (non-Javadoc)
    * @see
org.eclipse.gef.editparts.AbstractGraphicalEditPart#getModelSourceConnections()
    */
   protected List getModelSourceConnections() {
      // TODO: implement to return the objects represented by the connections
sourcing from this node
      throw new UnsupportedOperationException();
   }
   /* (non-Javadoc)
    * @see
org.eclipse.gef.editparts.AbstractGraphicalEditPart#getModelTargetConnections()
...
}
```

4. We also create NodeFigure.javajet, to generate a figure for each EditPart.
5. We change the JET properties for our project to ensure that the translated templates are compiled into the src directory. To do this, we open the properties of the project, select **JET Settings**, and then set **Source Container** to `src`.
6. We compile each template by selecting the template and then selecting **Compile Template** from the right-click context menu. Now, we should see the translated templates appear in the src directory.
7. We create a class EditPartGenerator in the com.ibm.itso.sal330r.codegen package that was created for the translated templates.
8. In the main method of EditPartGenerator, we add code to get classes from the model, and use them as arguments to the generate() method of our compiled templates. Example 5-13 shows the code that we add to facilitate this. Note that we must use the init() method on the NetworkPackage to initialize it before use.

Example 5-13 Using the templates

```
NodeEditPartTemplate n = new NodeEditPartTemplate();
NodeFigureTemplate f = new NodeFigureTemplate();
WorkflowPackageImpl.init();
Map registry = EPackage.Registry.INSTANCE;
String workflowURI = WorkflowPackage.eNS_URI;
WorkflowPackage workflowPackage = (WorkflowPackage) registry.get(workflowURI);
// Generate TaskEditPart
EClass taskClass = workflowPackage.getTask();
String result = n.generate(taskClass));
```

9. The result of calling generate() on the template is a string containing the text generated from the template. In our simple example, we print this to System.out, however if you were really generating code, you would want to create a resource containing the contents of the String.
10. If you run the EditPartGenerator as a Java application, you will see the resulting code printed to the console.

Using a similar approach to the EMF codegen for the model, edit, and editor plug-ins, you could generate a generic graphical editor for any model using JET. You would probably want to use your own GenModel to represent options such as whether a class maps to a Node or Connection EditPart, whether it can contain other nodes, and possibly also to specify the type of Figure used to represent the class. You could then generate from instances of that model rather than from the application model directly.

Part 2

Sample application

In this part of the book, we describe our redbook sample application. We discuss sample requirements and design, and show how to implement the sample.

6

Sample requirements and design

In this chapter, we introduce our redbook sample application, and describe its objectives. We define the requirements and explain our design decisions.

6.1 Sample application requirements

In this section we introduce the sample application and describe its features.

6.1.1 The application

The problem space we have chosen to demonstrate model construction is workflow. It has a few key concepts that should be interesting to our readers, and is general enough so that all our readers should understand these concepts.

A workflow is a collection of tasks. Two types of task have been defined: simple and complex.

Simple tasks

A simple task, as represented in Figure 6-1, has one input, one output, and one fault output. A simple task does some sort of processing on the data given to it. Two tasks are linked together with an edge. Data on the input is processed by the task and made available on the output.

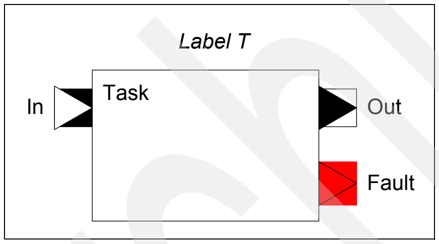


Figure 6-1 Task representation

Complex tasks

These are the complex tasks that we use in our sample application:

- **Compound task:** A compound task is a kind of container. It follows the composite pattern. It contains a containment reference to a workflow, which can contain other simple or compound tasks.
- **Loop task:** The loop task gives us the ability to iterate, as long the condition, a predicate, is true.
- **Choice task:** The choice task implements branching.
- **Transformation:** Transformation has been introduced in order to enable a task to do a combination of its multiple inputs.

Figure 6-2 shows the representation we use for complex tasks.

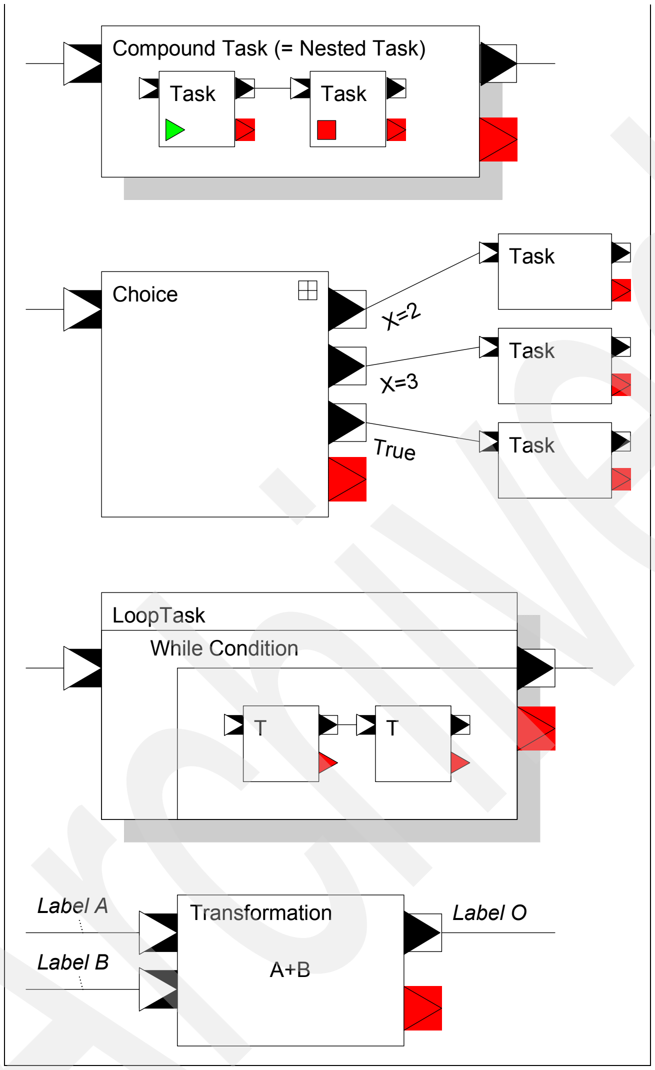


Figure 6-2 Complex task representation

Edge

An edge is used to link two tasks together. The output of the first task is redirected to the input of the second task. An edge represents both control and data flow. This means that once the first task has completed, the data made available to and the control flow is transferred to the second task.

Multiple edges can end to a task input slot. This means that the task has to wait for all former tasks to reach the completion stage, before being able to process the multiple data set available.

Figure 6-3 shows our representation of edges.

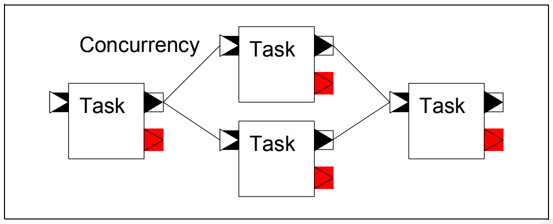


Figure 6-3 Concurrency and edge representation

Variables and labels

The final two concepts that we introduce are the use of labels and the use of variables. Labels can be used to decorate any of the input, output, and fault slots of a task, to decorate conditions on a conditional edge from a conditional task, and to decorate variables.

Variables are used to store data, usually coming from the output of a task, and to hold the data until another task in the workflow makes use of it. Variables can be seen as a way to separate the control flow from the data flow. Control goes to the next task, while the data is held in the variable.

Start and end tasks

In order to run the workflow, we define a start and end point as a decoration of a task. The start icon is a green triangle, while the stop icon is a small red square. See Figure 6-4 for an example.

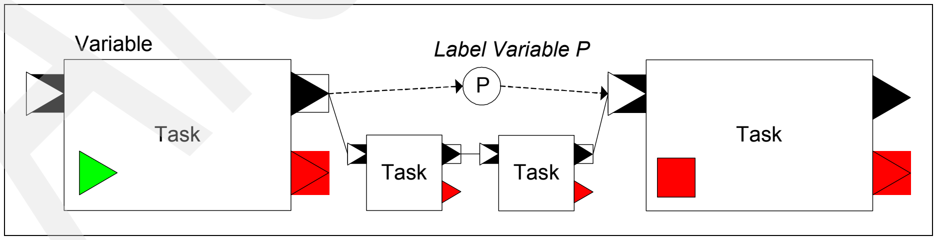


Figure 6-4 Data flow, variable, start, and stop tasks representation

6.2 Sample application design

The EMF and GEF sample application is an Eclipse plug-in. It is an editor, which uses a property view to capture user input, and provides an outline view to help the user to navigate more easily through the model. It also has multiple levels of undo and redo, and provides several Eclipse plug-in extension points.

These plug-in extension points are:

- The view menu, which contains the Zoom In and Zoom Out menu items
- The undo, redo tool bar

At the editor level, additional context dependent features have been defined:

- Inplace editing for compound tasks:
  - A task can be added to a compound task by means of drag and drop.
  - A sub-workflow can be accessed through the compound task itself.
- A separate editor tab for compound task editing:
  - This provides an extended work space to work on a sub-workflow defined in a compound task.
- During edge creation, which is a link between two nodes, the link creation tool is smart enough to recognize where the link can be connected:
  - No loop on a single task is allowed.
  - No link from a task of a sub-workflow to a task in the main workflow is allowed.
  - Link cursor is dynamically updated to represent the ability to connect to a given endpoint.
- Dynamic update of the main and properties view, to reflect the user action on the outline view.
- Drag and drop from the palette into the viewer.
- Right-click contextual menu:
  - For example, the Choice right-click menu contains undo, delete, add condition to choice and save actions.

6.2.1 Design decisions

During the design process, we made some important decisions, including these:

- There is one top level workflow per file.
- The sub-workflow of a compound task is contained in the workflow itself. No reference to an external workflow or sub-workflow is supported.

Note: In Eclipse, only one editor can be opened on a workflow at a time, but multiple workflows can be edited in different Eclipse workflow editors.

6.2.2 The workflow model

This section documents the WorkflowModel, shown in Figure 6-5.

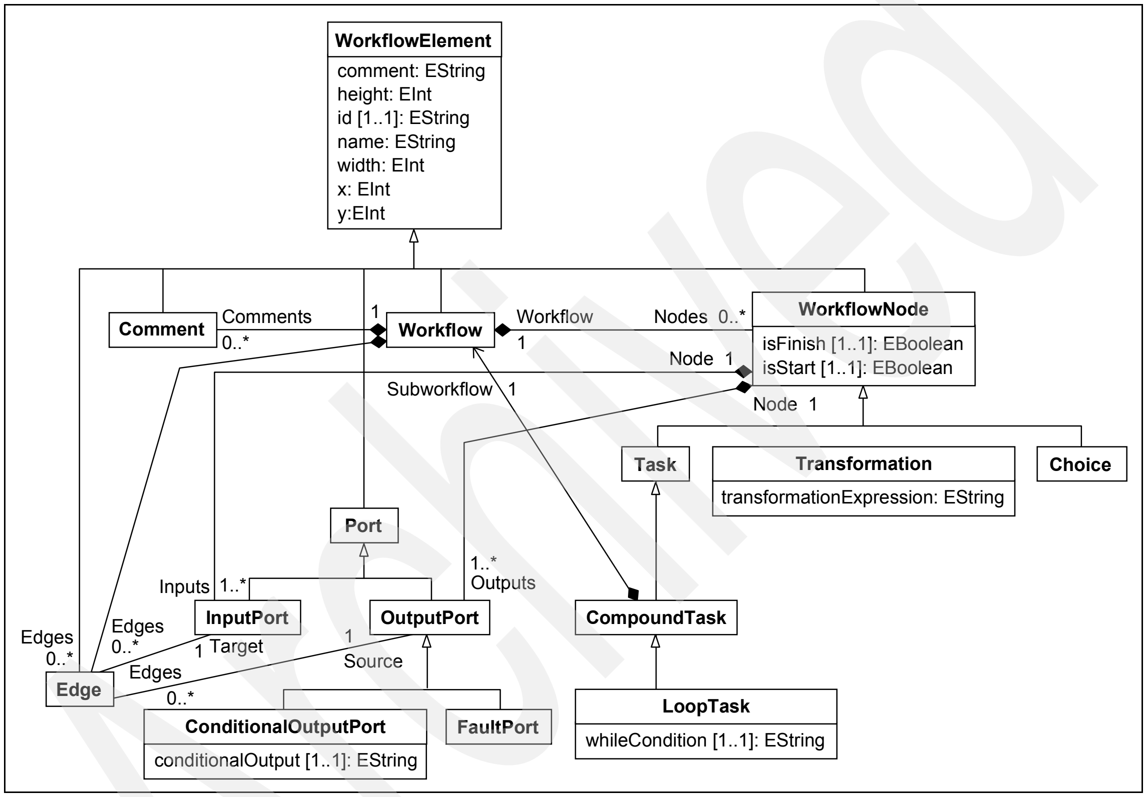


Figure 6-5 The WorkflowModel

WorkflowElement

The WorkflowElement class provides features common to all elements present in a workflow. It is the common abstract supertype of the Workflow, WorkflowNode Port, Edge and Comment classes. Table 6-1 provides a summary of the WorkflowElement class.

Table 6-1 WorkflowElement summary

Owned By	
Inheritance	
Features	name: Identifies the Workflow comment: An optional comment string x: Coordinate used for layout y: Coordinate used for layout width: Used for layout of container elements height: Used for layout of container elements id: Used to uniquely identify a workflow element
Constraints	

Workflow

The Workflow class represents the description of a process. A Workflow contains WorkflowNodes representing the steps in the process and Edges that represent data and control flow between the nodes. A Workflow may also contain comments that annotate the process described by the Workflow. Table 6-2 provides a summary of the Workflow class.

Table 6-2 Workflow summary

Owned By	
Inheritance	WorkflowElement
Features	nodes: The WorkflowNodes contained within this Workflow edges: The Edges contained within the Workflow comments: The Comments contained within the Workflow
Constraints	

WorkflowNode

The class WorkflowNode represents a step in a Workflow. WorkflowNodes have ports and may be connected to other WorkflowNodes via those ports. Table 6-3 provides a summary of the WorkflowNode class.

Table 6-3 WorkflowNode summary

Owned By	
Inheritance	WorkflowElement

Features	isStart: Indicates whether this is the starting node of a Workflow isFinish: Indicates whether this is the finishing node of a Workflow workflow: A reference to the Workflow that contains the node outputs: Output ports (including fault ports) owned by the node inputs: Input ports owned by the node
Constraints	WorkflowNodes have no more than one Fault port.

Task

The class Task represents an action or unit of work within the Workflow.

The start and end icons in the documentation are different from the ones currently implemented. In the application, the start task's InputPort is replaced by a green square and the end task's OutputPort is replaced by a red square as shown in Figure 6-6.

Table 6-4 provides a summary of the Task class.

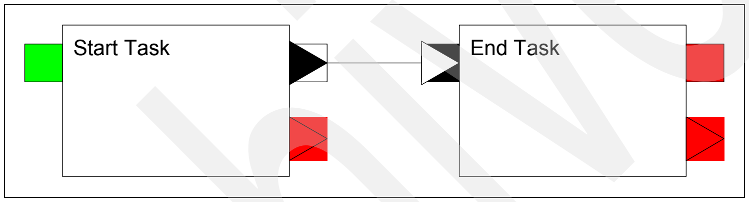


Figure 6-6 Task visual

Table 6-4 Task summary

Owned By	
Inheritance	WorkflowNode
Features	
Constraints	A Task has exactly one input port and one (non-fault) output port.

CompoundTask

The class CompoundTask is a Task that is defined by a sub-workflow. The CompoundTask is complete when the sub-workflow that composes it is complete. As CompoundTask inherits from Task, it has a single input port, output port and fault port. When a CompoundTask begins, the inputs to the start nodes of the sub-workflow are the inputs that are received at the input port of the CompoundTask. Similarly, when the sub-workflow completes, the output data from the finishing nodes of the sub-workflow provide the data that is output from the output port of the CompoundTask. Figure 6-7 shows the visual representation of a CompoundTask.

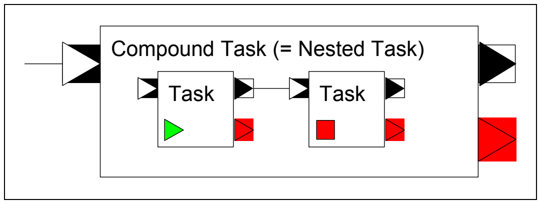


Figure 6-7 Compound task visual

Table 6-5 provides a summary of the CompoundTask class.

Table 6-5 CompoundTask summary

Owned By	
Inheritance	Task
Features	subworkflow: The Workflow that defines the CompoundTask
Constraints	

LoopTask

The LoopTask represents actions that are repeated while a condition is true. The actions that are repeated are contained by the sub-workflow of the LoopTask. The data received at the Input of the LoopTask is provided as the input to the first execution of the start node in the LoopTask's sub-workflow. For each repetition of the sub-workflow, the output from the previous execution becomes the input to the current one. The output from the finishing nodes from the final execution of the loop becomes the output of the LoopTask. Figure 6-8 shows the visual representation of the LoopTask

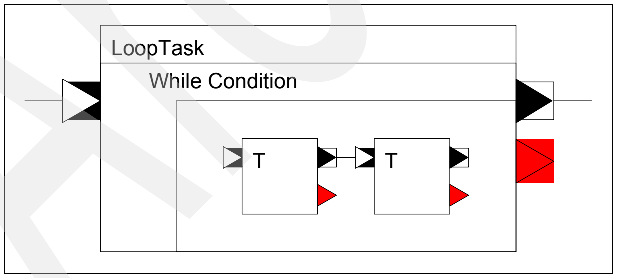


Figure 6-8 LoopTask visual

Table 6-6 provides a summary of the LoopTask class.

Table 6-6 LoopTask summary

Owned By	
Inheritance	CompoundTask
Features	whileCondition: While this holds, the sub-workflow is repeated
Constraints	

Choice

The class Choice represents a switch between alternative execution and data flow paths. Data and control flow is only activated for Edges that source from output ports of the Choice where the condition of the OutputPort evaluates to true. Conditions must be unique in a Choice. The default name for a condition is false.

The way conditions are represented in the workflow editor differ a little bit from the present documentation, where a condition is drawn close to the corresponding edge. In the editor, they are drawn inside the Choice visual itself. A condition is placed in front of the corresponding ConditionalOutputPort port, see Figure 6-9.

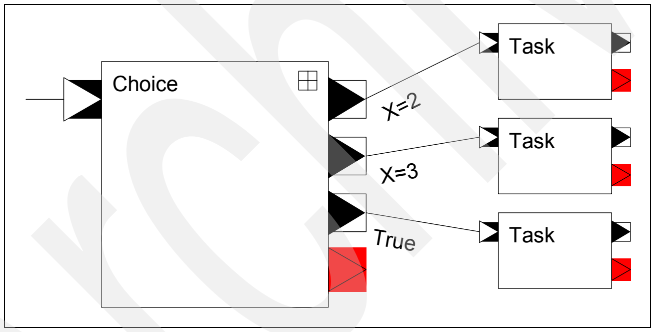


Figure 6-9 Choice visual

Note: The little icon on the upper right corner of the Choice is used for adding a condition to it. If you click it and nothing happens, check if the default false condition is not already defined in the Choice.

Table 6-7 provides a summary of the Choice class.

Table 6-7 Choice summary

Owned By	
Inheritance	WorkflowNode
Features	
Constraints	Choice has only one input port, but may have multiple output ports. Non-fault OutputPorts owned by a Choice must be ConditionalOutputPorts.

Transformation

The class Transformation takes multiple inputs and performs a transformation on that data to produce a single result. Figure 6-10 shows the visual representation of a Transformation.

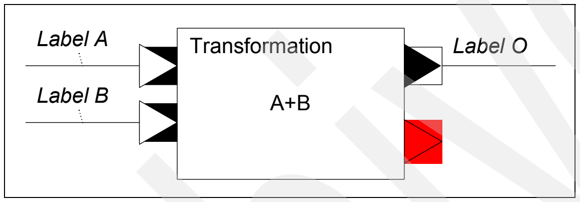


Figure 6-10 Transformation task visual

Table 6-8 provides a summary of the Transformation class.

Table 6-8 TransformationTask summary

Owned By	
Inheritance	WorkflowNode
Features	transformExpression: Expresses how the input data is transformed into the output data
Constraints	Transformation has only one (non-fault) output port, but may have multiple input ports

Edge

The class Edge represents a connection between an output port and an input port, that is, a flow of data from the output of one WorkflowNode to the input of another. Table 6-9 provides a summary of the Edge class.

Table 6-9 Edge summary

Owned By	Workflow
Inheritance	WorkflowElement
Features	workflow: The containing Workflow source: The output port from which the Edge begins target: The input port at which the Edge terminates
Constraints	

Port

The abstract class Port is the common supertype for InputPort and OutputPort.Table 6-10 provides a summary of the Port class

Table 6-10 Port summary

Owned By	
Inheritance	WorkflowElement
Features	
Constraints	

InputPort

The class InputPort represents the Port at which data and control is received by a node in the Workflow. Table 6-11 provides a summary of the InputPort class

Table 6-11 Input Port summary

Owned By	WorkflowNode
Inheritance	Port
Features	edges: The edges that target the InputPort node: The WorkflowNode that owns the InputPort
Constraints	

OutputPort

The class OutputPort represents the Port at which data and control is provided by a WorkflowNode upon completion. Table 6-12 provides a summary of the OutputPort class.

Table 6-12 OutputPort summary

Owned By	WorkflowNode
Inheritance	Port
Features	node: The WorkflowNode that owns the OutputPort edges: The edges for which the OutputPort is the source
Constraints	

FaultPort

FaultPort represents the output of a node that terminates under exceptional conditions. The exception may be handled by another node if there is an Edge linking the FaultPort with the InputPort of the handling WorkflowNode, otherwise the Workflow containing the WorkflowNode that owns the FaultPort fails. Table 6-13 provides a summary of the FaultPort class

Table 6-13 FaultPort summary

Owned By	
Inheritance	OutputPort
Features	
Constraints	

ConditionalOutputPort

The ConditionalOutputPort class represents an output of a Choice. For any Choice, the conditions of its ConditionalOutputs determine the execution paths that are taken upon evaluation of the Choice, as only Edges sourcing from a ConditionalOutputPort where the condition evaluates to true will be activated when the Choice completes. Table 6-14 provides a summary of the ConditionalOutputPort class

Table 6-14 ConditionalOutputPort summary

Owned By	
Inheritance	OutputPort
Features	condition: The condition used to determine the execution path
Constraints	

Comment

The Comment class represents a free-standing comment within a Workflow. The text of the comment is represented in the comment attribute inherited from WorkflowElement. The Comment class provides a mechanism for including comments in the workflow that are not attached to the Ports, Edges or WorkflowNodes. Table 6-15 provides a summary of the Comment class.

Table 6-15 Comment summary

Owned By	Workflow
Inheritance	WorkflowElement
Features	
Constraints	

6.3 Sample application demo

The sample application workflow editor is the default editor for file with a `.workflow` extension.

To run the workflow sample application, we need to first create a simple project, than create a workflow file using the simple file creation wizard or the workflow wizard. The workflow wizard provides workflow file extension handling and control.

To create a simple project:

1. Click **File -> New -> Other...**, select **Simple -> Project**, click **Next**.
2. Give the project name, click **Finish.**

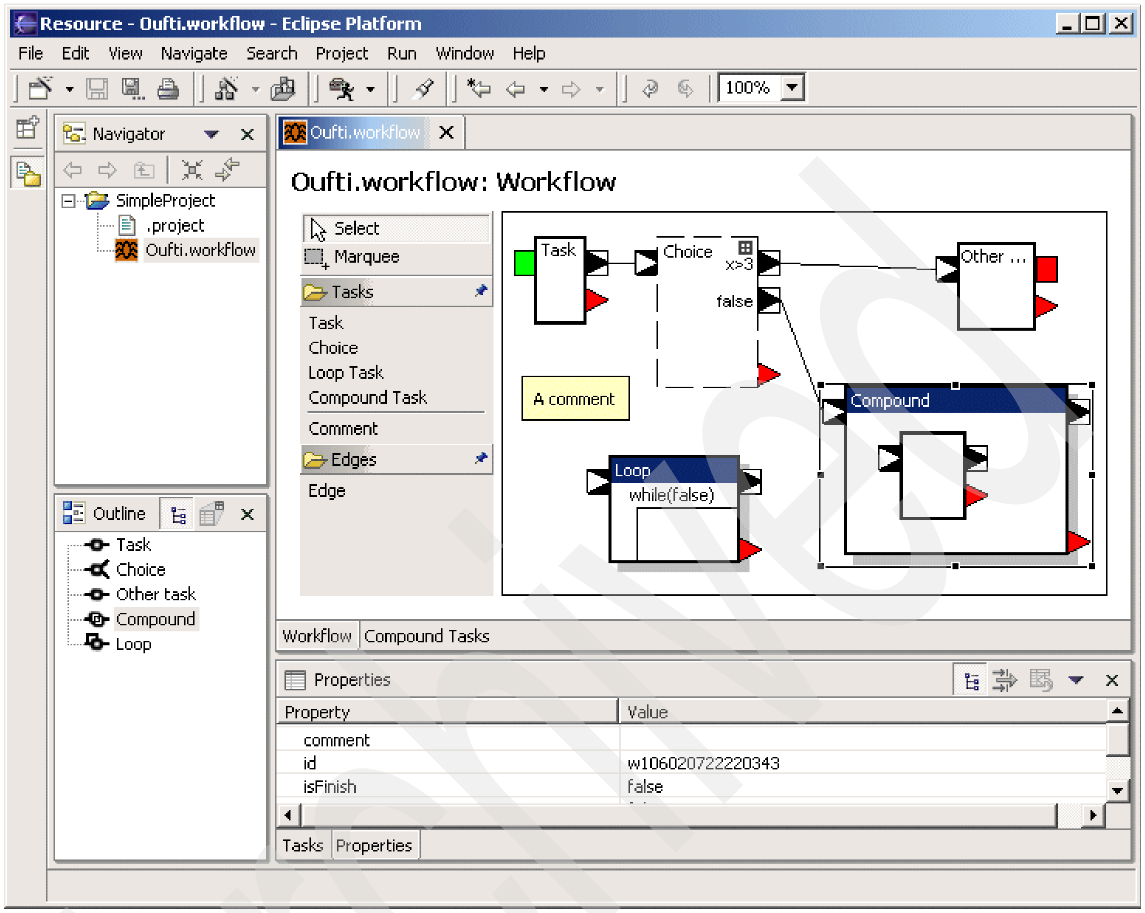
To create a workflow model with the simple file creation wizard:

1. Click **File -> New -> Other...**, select **Simple -> File**, click **Next**.
2. Give the file name, for example `myworkflow.workflow`, click **Finish**.

To create a workflow model with the simple file with the workflow wizard:

1. Click **File -> New -> Other...**, select **Other -> Workflow**, click **Next**.
2. Give the file name, for example `My.workflow`, click **Finish**.

In both cases, the workflow editor opens automatically on a new empty workflow. Figure 6-11 shows a workflow model built using our redbook sample application.



Property	Value
comment	
id	w106020722220343
isFinish	false

Figure 6-11 Workflow sample application window

Notes:

1. If your starting point is the additional material, which contains the plug-in code, you have to run the plug-in on a Run-time Workbench. Eclipse automatically opens the editor on the workflow file, created during the simple file creation process.
2. The Edge creation tool was considered as a composite of the model. It was presented in a way similar to Tasks, with an Edges menu and an Edge entry. Later on, it has been considered not as being a composite, but more like a link between two composites. As such, it was moved to the top of the menu, just after the Select and Marquee tools.

7

Implementing the sample

In this chapter, we discuss the implementation of our workflow editor sample application. We describe its architecture, the model, and the multi-page editor.

7.1 Overview

In this section we provide an overview of the sample application. We provide a summary of the packages and classes in the implementation, along with instructions for how to run the sample application. We also highlight notable sections from the JavaDoc.

7.2 Architecture

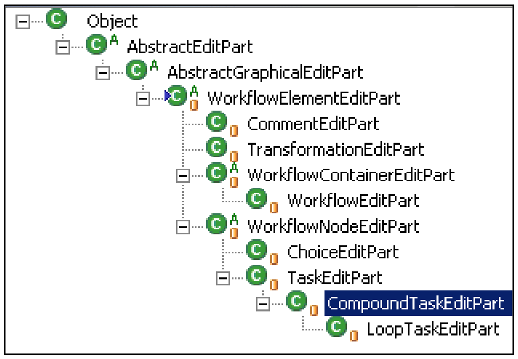
In this section we describe the architecture of the sample application. We discuss the techniques that we use to connect the EMF model with the editor framework.

7.2.1 Mapping the EMF model to GEF EditParts

One of the key tasks in creating a GEF application is the process of mapping your applications EditParts with your model. In this section we discuss the process that we took to bind our sample application's EMF-based model with the editor framework. The GEF provides a lot of flexibility as far as how its EditParts relate to the underlying model. There are no strict requirements on how EditParts map to actual objects in the model. The first step, then, is to decide what this mapping will be in your application.

In general, there will probably be fewer EditParts than there are object classes in your model. For instance, in our sample application, we created the WorkflowNodeEditPart to be the base class for model elements that have connections. In the model, the ports are separate objects; but in the editor, we chose to have the WorkflowNodeEditPart represent both the node and all its ports. One criterion for making a determination about this mapping is to consider how dynamic the visual behavior of a component needs to be.

For instance, if a model object needs a visual representation that can be moved, resized, or can be individually added or deleted, then it may be a good candidate for mapping it to its own EditPart. In our sample application, we designed the EditPart class hierarchy shown in Figure 7-1.



```
Object
  AbstractEditPart
    AbstractGraphicalEditPart
      WorkflowElementEditPart
        CommentEditPart
        TransformationEditPart
        WorkflowContainerEditPart
          WorkflowEditPart
        WorkflowNodeEditPart
          ChoiceEditPart
          TaskEditPart
            CompoundTaskEditPart
              LoopTaskEditPart
```

Figure 7-1 The sample application's EditPart class hierarchy

EditPart functionality

The base class for our EditParts is the WorkflowElementEditPart class, which provides the following three main functions needed by all its subclasses:

- **Register the EditPart a listener of its model:**

  The WorkflowElementEditPart class implements the interface that is used by listeners of EMF's notification mechanism:

  ```
  org.eclipse.emf.common.notify.Adapter
  ```

  Tracking changes in the model is crucial to the EditPart's function (discussed in detail in 7.2.2, "Tracking model events in the editor" on page 207). We override the EditPart's life cycle methods activate() and deactivate() to manage registration of the EditPart as an Adapter on its model.

- **Register the EditPart as a property source:**

  All EditParts inherit the IAdaptable interface from their AbstractEditPart base class. This Eclipse interface supports a kind of multiple inheritance in which a class can offer a proxy object to implement an interface requested by the Eclipse framework. In our case we want all of our EditParts to provide an implementation of the IPropertySource interface. By doing so the Eclipse property page viewer will display the properties of our EditParts as they are selected, and also allow them to be edited.

  The implementation of the IPropertySource interface requires adding Eclipse-specific code. While we could have extended our model's objects to implement this interface directly, we felt that it would be preferable to keep the Eclipse properties handling out of the model classes. Fortunately, the EMF-generated classes provide a lot of metadata. This made it simple to create a proxy class that provides a generic IPropertySource implementor, WorkflowElementPropertySource, that can provide the requisite IPropertyDescriptor's for any of our model's classes.

This also provides for editing property values. Most of the work happens in this class's getPropertyDescriptors() method, shown in Example 7-1.

Example 7-1 A generic getPropertyDescriptors implementation for EMF classes

```
public IPropertyDescriptor[] getPropertyDescriptors() {
               Iteratorit;
      EClass cls = element.eClass();
      Collectiondescriptors = new Vector();

      it = cls.getEAllAttributes().iterator();
      while( it.hasNext() ) {
         EAttributeattr = (EAttribute)it.next();

         EDataTypetype = attr.getEAttributeType();
         if( attr.isID() ) {
            // shouldn't be editable
            descriptors.add( new PropertyDescriptor( Integer.toString(
attr.getFeatureID() ),
                                                  attr.getName() ) );

            }
         else if( type.getInstanceClass() == String.class ) {
            descriptors.add( new TextPropertyDescriptor( Integer.toString(
attr.getFeatureID() ),
                                                  attr.getName() ) );
         }
         else if( type.getInstanceClass() == boolean.class ) {
            descriptors.add( new CheckboxPropertyDescriptor( Integer.toString(
attr.getFeatureID() ),
                                                     attr.getName() ) );
         }
      }

      return (IPropertyDescriptor[])descriptors.toArray( new
IPropertyDescriptor[] {} );
   }
```

Using metadata in the EAttribute and EDataType classes, we construct a property descriptor for each property (attribute) of a model object. The EAttribute provides a unique ID and displayable name for each attribute. The type information in the EDataType is used to create subclasses of PropertyDescriptor that will provide cell editors appropriate for the data type of each attribute. Notice the special case for ids, which are identified by testing the EAttribute.isID() method. This attribute is the unique ID that is generated for each object in the model. We don't want this attribute to be editable, so we create an instance of PropertyDescriptor, which results in a read-only entry in the property page.

- **Provide a default implementation of the refreshVisuals method:**

  Our implementation of this method handles changes to an EditPart's size and position. In our sample application all of our EditParts can be moved, and most of them can be resized. Therefore we provide this functionality in our EditPart base class. The refreshVisuals() method simply applies the changed position and extent values in the model to the EditPart's figure by updating its layout constraint accordingly.

  The WorkflowNodeEditPart derives from WorkflowElementEditPart and its purpose is to support EditParts that have connections. This is the base EditPart for EditParts that map to the model classes derived from the WorkflowNode. This class implements GEF's NodeEditPart interface, which supports the connection feedback mechanism in GraphicalNodeEditPolicy. This gives user feedback when Connections are initially connected and also if they are later disconnected and reconnected.

7.2.2 Tracking model events in the editor

Once your EditPart to model mapping strategy has been designed, the next step is to enable your EditParts to be able to track changes in your model. As we have said, GEF itself does not provide nor require any specific event notification mechanism. If you are working with a model that does not include an event mechanism, one approach is to use the Java beans event support provided by the classes, java.beans.PropertyChangeSupport and java.beans.PropertyChangeListener.

This is the approach taken in the logic example that the GEF project provides. In our case we are fortunate to have the full-featured notification mechanism that is generated automatically in EMF classes. Recall that all the EditParts in our sample are registered as adapters on the EMF model class(es) that they represent. Each EditPart then provides an override of the notification method:

```
public void notifyChanged(Notification notification)
```

This method is called when any attribute of a model class is changed, or when a child object is added or removed. The Notification class provides extensive context describing the model change that has occurred. It includes information such as:

- The notifier, that is, which object's property has changed, or had a child added or removed
- The new and previous values of the target attribute
- The data type information for the affected attribute
- An identifier for the attribute

This information is used to filter out events so that each EditPart only processes changes that are unique to properties of that particular part. Processing of more generic changes, for instance a change to a part's size or location, should be delegated to the superclasses implementations of notifyChanged(). Notice that the notification mechanism provided by EMF is very thorough, so that a change to any attribute will result in a notification event. This means that a more complicated model operation, in which several attributes are manipulated, results in a large number of notification events. Ideally the EditPart's implementation will filter these events accordingly so that the visual representation is maintained accurately while events that do not require a change to the visual representation are ignored.

Remember that a single EditPart may be responsible for the representation of more than one object in the underlying model. In our sample application this is the case with WorkflowNodeEditParts, which represent a WorkflowNode with some number of Ports. In our model the action of adding or removing a connection is something that happens to ports, not the WorkflowNode to which it is attached. Therefore our WorkflowNodeEditPart needs to perform some additional registration to make itself a listener on its WorkflowNode's ports. Otherwise it will not be notified of connection changes to its ports. This is done in the notifyChanged() method of the WorkflowNodeEditPart, which is a base class for all the EditParts in our sample application that support connections. When a port is added to any WorkflowNode model element, the WorkflowNodeEditPart adds itself as a listener on the new port.

7.2.3 Refreshing

Once we have ensured that our EditParts are receiving all the notifications they require to track their model, we then need to add code in our EditParts that acts on this information. The implications of a model event to an EditPart can be distilled into three general operations. The EditPart must interpret the notification to decide which of these operations are required:

- **Updating the visual representation:**

  Underlying attributes of the model are often represented visually using colored indicators, text annotations, or other graphical effects. For example, in the sample application the name of an element is drawn inside a task rectangle or on the title bar of a compound task. The ports change color to indicate when a task is a start or finish task. The EditPart class provides the method refreshVisuals(). Its implementation should provide a full update of every graphical feature that is mapped to a model attribute. This method will be called once when the EditPart is first activated so that the model and figure are synchronized. Subsequently it is the responsibility of the application to decide when a model change event requires an update to the visualization. It is not required, or always advisable, to update the entire visualization if only a single attribute has changed. This is a judgement call depending on the complexity of the figure. With a detailed notification mechanism such as the one provided by EMF, it is easy to determine exactly what has changed in the model and decide whether to update details of the figure vs. calling refreshVisuals() to update the entire figure.

- **Updating children:**

  In our sample application, our model has containment relationships that are mirrored in our EditPart hierarchy, which is a common situation in GEF applications. In our case the Workflow object may contain Task, CompoundTasks, Choice and LoopTask objects, and so on. Our model supports nesting, so that there are sub-workflows within CompoundTasks and LoopTasks. The EditParts that represent these objects maintain a similar structure. When a container EditPart is notified that a child model element has been added or removed from its model, it must interact with the GEF framework to synchronize by either adding or removing the EditParts that represent the affected model children. GEF provides the EditPart method refreshChildren() for this purpose. GEF provides the implementation of this method. Our notification just needs to call it when appropriate, as we do in the WorkflowNodeEditPart, shown in Example 7-2:

Example 7-2 The notifyChanged() implementation in WorkflowNodeEditPart

```
public void notifyChanged(Notification notification) {
      int type = notification.getEventType();
      int featureId;

      switch( type ) {
         case Notification.ADD:
         case Notification.ADD_MANY:
            if( notification.getNewValue() instanceof Edge ) {
                  if( notification.getNotifier() instanceof InputPort ) {
                  refreshTargetConnections();
               }
               else {
                  refreshSourceConnections();
               }
            }
            else {
               // listen for connection changes on the port
               if( notification.getNewValue() instanceof Port ) {
                  Port port = (Port)notification.getNewValue();
                  port.eAdapters().add( this );
               }
               refreshChildren();
            }
            break;
            case Notification.REMOVE:
            case Notification.REMOVE_MANY:
               if( notification.getOldValue() instanceof Edge ) {
                  if( notification.getNotifier() instanceof InputPort ) {
                     refreshTargetConnections();
                  }
                  else {
                     refreshSourceConnections();
                  }
               }
               else {
                  if( notification.getNewValue() instanceof Port ) {
                     ((Port)notification.getNewValue()).eAdapters().remove( this
);
                  }
                  refreshChildren();
               }
               break;
```

We detect the addition or removal of children using the Notification.getEVentType() method. GEF's refreshChildren() method it will need to know what parts of the model the EditPart considers to be its model's children. Therefore EditParts that contain other EditParts must provide an implementation of the EditPart.getModelChildren, which returns a list of the child model elements. GEF then reconciles the model children against the list of EditParts that it maintains. If a there is a new child element then an EditPart will be created for it, or if one has been deleted than the corresponding EditPart will be removed. The implementation of getModelChildren() for CompoundTaskEditParts is show here Example 7-3:

*Example 7-3 The getModelChildren implementation in CompoundTask*EditParts

```
protected List getModelChildren() {
   List result = new ArrayList();

   if( getCompoundTask().getSubworkflow() != null ) {
      Iterator it;

      it = getCompoundTask().getSubworkflow().getNodes().iterator();
      while( it.hasNext() ) {
         result.add( it.next() );
      }
      it = getCompoundTask().getSubworkflow().getComments().iterator();
      while( it.hasNext() ) {
         result.add( it.next() );
      }
}

      return result;
   }
```

Notice that the Comment objects are also added here because they are owned by the containing workflow (but are not part of the node hierarchy).

- Updating connections

  EditParts must notify GEF when they detect model changes indicating the making and breaking of connections. The mechanism for this is very similar to the mechanism described above for child additions and deletions. In the case of our sample application we do this processing in the example Example 7-2 above. GEF provides two methods for refreshing connections, depending on whether the affected EditPart is the source or target of the connection. The methods are named EditPart.refreshSourceConnections and EditPart.refreshTargetConnections. As it does when refreshing children, GEF then asks our EditPart to provide a list of the connections for which our EditPart is the source or target. For our model we simply need to return the result of the WorkflowNode class's getOutputEdges or getInputEdges, which conveniently return a List as required by GEF (see Example 7-4)

Example 7-4 Returning a node's connections

```
protected List getModelSourceConnections() {
      return getWorkflowNode().getOutputEdges();
   }

   protected List getModelTargetConnections() {
      return getWorkflowNode().getInputEdges();
   }
```

7.2.4 Factories

We use two factories in order to integrate between GEF and our EMF-based model. First we need to use the EMF-generated factory, WorkflowFactory, whenever we are creating new model objects. Typically this happens when a creation command is either initialized by a policy or when the creation command is executed. The factory is made available to these functions by setting it as the factory for the creation tools created in the palette, as we do in the WorkflowPaletteRoot class. The factory is set in the constructor for the CreationToolEntry class. The class ModelCreationFactory, which implements CreationFactory, is where the EMF factory is invoked. The getNewObject() method in this class is where objects are actually created, as show in the snippet in Example 7-5.

Example 7-5 A snippet of the getNewObject() factory method

```
public Object getNewObject() {
   Map registry = EPackage.Registry.INSTANCE;
   String workflowURI = WorkflowPackage.eNS_URI;
   WorkflowPackage workflowPackage =
   (WorkflowPackage) registry.get(workflowURI);
   WorkflowFactory factory = workflowPackage.getWorkflowFactory();

   Object result = null;

   if( targetClass.equals( Task.class ) ) {
      result = factory.createTask();
   }
   else if( targetClass.equals( CompoundTask.class ) ) {
      result = factory.createCompoundTask();
   }
   else if( ... ) ) {
   }
         return result;
}
```

In 7.2.3, “Refreshing” on page 208, we discussed the reconciliation process that GEF performs when our EditParts call refreshModelChildren, refreshTargetConnections or refreshSourceConnections. If GEF detects that there are model elements without an associated EditPart, it uses the graphical viewer’s factory to create the missing EditPart. In the sample application the class GraphicalEditPartsFactory is our implementation of EditPartFactory that performs this function. This simple class is what ultimately specifies how our model’s objects will be mapped to our application’s EditParts.

7.2.5 Policies and commands

GEF editors only become interactive when the appropriate EditPolicy implementations are added to EditParts. The EditPolicies are responsible for creating commands to operate on the model and to provide feedback behaviors that allow figures to be selected, dragged, added, deleted, and edited. Our sample uses the following EditPolicies:

- **WorkflowContainerXYLayoutEditPolicy:** One of the main function of this policy is to construct creation commands in response to a CreateRequest request. Most of the objects in the sample application’s model that map to EditParts are subclasses of WorkflowNode. The class CreateWorkflowNodeCommand is the command that handles creation of these objects. In the policy’s getCreateCommand the command is initialized with parent workflow, and then the factory is called to get a new child instance (Example 7-6). Notice the special handling when the host is a

CompoundTask. In that case the parent workflow is obtained by calling the CompoundTask's getSubworkflow() method.

Example 7-6 Initializing the CreateWorkflowNodeCommand

```
CreateWorkflowNodeCommand create = new CreateWorkflowNodeCommand();
if( getHost().getModel() instanceof Workflow ) {
   create.setParent((Workflow)getHost().getModel());
}
else {
   create.setParent(((CompoundTask)getHost().getModel()).getSubworkflow());
}
create.setChild( (WorkflowNode)request.getNewObject() );
```

The Comment object has its own creation command, CreateCommentCommand, because it is not a WorkflowNode; it has no connections and is always contained by a workflow.

Other functionality provided by the WorkflowContainerXYLayoutEditPolicy includes providing a ChangeConstraintCommand. This command is executed when the user changes the size or location of a model element. The policy also determines the SelectionHandlesEditPolicy for new EditPart children, making CommentEditParts nonresizeable, while the other EditParts are allowed to be resizeable.

- **ChoiceDirectEditPolicy:** This class supports the direct edit mechanism that the sample application uses for editing the expressions in ChoiceEditParts. It constructs a ChoiceExpressionCommand for a DirectEditRequest. It also performs some ancillary functions such as saving the current value of an expression and implementing the showCurrentEditValue() method to take into account which label the user is attempting to edit.
- **CompoundHighlightEditPolicy:** We created this subclass of GraphicalEditPolicy to provide visual feedback when a CompoundTask is the target of an operation, such as when a Task is being dragged into it. The feedback is simply to change the background color of the figure which contains the sub-workflow figures.
- **EdgeEditPolicy:** This policy supports the deletion of Edges from the model by constructing a ConnectionCommand for the host Edge with null specified for the source and target.
- **EdgeEndpointEditPolicy:** We override the ConnectionEndpointEditPolicy to provide some extra visual feedback when an EdgeEditPart is selected. The feedback is simply to double the width of the connection's polyline figure, and to return its width to a single pixel again when it is deselected.

- **EdgeSelectionHandlesEditPolicy:** We must provide a concrete implementation of the abstract base class SelectionHandlesEditPolicy that returns the selection handles for our connection EditParts. Since in the sample we do not support a bendpoint router, we just return handles for the start and end of the connection.
- **WorkflowContainerEditPolicy:** This is another container related policy.
- **WorkflowNodeEditPolicy:** This policy creates commands for connection initiation and completion (ConnectionCommand). Its superclass, GraphicalNodeEditPolicy, provides visual feedback while a connection is being drawn.

7.3 The model

In this section, we describe the model used by the sample application.

7.3.1 Modifying the WorkflowModel

In this section we describe the modifications made to the WorkflowModel in order to use it in the workflow editor sample application.

Choosing the naming convention for references

A reference between two classes has usually two names associated with it. One for each of the navigation paths between them.

The one-to-one association names are singular and are always easily chosen. For the one-to-many association names, we have an extra level of freedom, because we can choose the Modeling or the Java naming convention to give it a name. The main difference between the two is that Modeling uses singular while Java uses plural.

Java coding conventions are strong. By respecting them, code is generally more readable and understandable. It is not that those conventions are the only way or the best way to go, but when you follow them, code become more easily familiar to developers. Modeling uses different conventions, because the interests are not the same.

Knowing that the convention choice has some effect on the generated code, the result is that you rapidly end up with some sort of decision like, do we privilegize the modeling or Java standpoint? When implementing the sample application, with Java code as the only mapping, we decide to use the Java standpoint.

To help you visualize the potential implications of the choice of one view, we use the Workflow to WorkflowNode association, called `node(s)` from Workflow to WorkflowNode.

In Java, the association is implemented by a collection called `nodes.`See Example 7-7:

Example 7-7 Java reference implementation

```
package com.ibm.itso.sal330r.workflow.impl;

public class WorkflowImpl extends WorkflowElementImpl implements Workflow {
    /**
     * The cached value of the '{@link #getNodes() <em>Nodes</em>}' containment 
reference list.
     * <!-- begin-user-doc -->
     * <!-- end-user-doc -->
     * @see #getNodes()
     * @generated
     * @ordered
     */
    protected EList nodes = null;

    /**
     * <!-- begin-user-doc -->
     * <!-- end-user-doc -->
     * @generated
     */
    public EList getNodes() {
   if (nodes == null) {
       nodes = new EObjectContainmentWithInverseEList(
            WorkflowNode.class,
            this,
            WorkflowPackage.WORKFLOW__NODES,
            WorkflowPackage.WORKFLOW_NODE__WORKFLOW);
   }
   return nodes;
    }
}
```

In XML, if you look at a file containing the result of a workflow serialization, you will see an extra ‘s’ at the end of each node entity, which is unusual for an XML entity. You can look at any Ecore file for more examples of eClassifiers or eReferences XML entities. See Example 7-8.

Example 7-8 Workflow XMI file serialization

```
<?xml version="1.0" encoding="ASCII"?>
<workflow:Workflow xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:workflow="http://www.redbooks.ibm.com/sal330r/workflow"
id="w10606325114530">
  <nodes xsi:type="workflow:Task" x="99" y="80" id="w10606325138591">
    <outputs xsi:type="workflow:FaultPort" name="fault" id="w10606325257817"/>
    <outputs name="output" id="w10606325257818"/>
    <inputs name="input" id="w10606325257819"/>
  </nodes>
  <nodes xsi:type="workflow:Choice" x="83" y="213" id="w10606325189373">
    <outputs xsi:type="workflow:FaultPort" name="fault" id="w106063252578110"/>
    <outputs xsi:type="workflow:ConditionalOutputPort" name="ConditionalPort0"
id="w106063252578111" condition="false"/>
    <inputs name="input" id="w106063252578112"/>
  </nodes>
  <nodes xsi:type="workflow:LoopTask" x="293" y="184" id="w10606325204534">
    <outputs xsi:type="workflow:FaultPort" name="fault" id="w106063252578113"/>
    <outputs name="output" id="w106063252578114"/>
    <inputs name="input" id="w106063252578115"/>
    <subworkflow id="w106063252578116"/>
  </nodes>
</workflow:Workflow>
```

7.3.2 Modifying the code generated from the model

This section describes additions and customizations made to the interfaces and implementations generated from the WorkflowModel, in order to use this generated code as the model for the workflow editor sample application.

7.3.3 Respecting model constraints in the editor

In this section we use the connectTo method in the Workflow class to show model object relationships, and explain the execute and undo method of the ConnectionCommand class. We describe the containment relationship between Workflow and Edge and the relationship between InputPort and OutputPort and Edge.

Enforcing model constraint in the model implementation

WorkflowNodes have Ports. The application requires Task, CompoundTask, and LoopTask tasks to have only one Input and one Output. Transformations can have multiple Inputs, and Conditionals can have multiple Outputs. All nodes have a default FaultPort.

The model, as designed, tell us that the association between tasks and ports are inherited from the WorkflowNode, where two one-to-many associations are defined between WorkflowNode and InputPort on one side and WorkflowNode and OutputPort on the other side. The reference named `outputs` contains all the OutputPort, all the ConditionalOutputPort, and the default FaultPort ports.

With those elements in mind, we can see that we have a problem to reduce the visibility of the inherited methods for `outputs` and `inputs` relationships. The model provides methods dealing with a collection, where methods dealing only with one object should be defined.

Nothing prevents the following code to be written in the case of a CoumpoundTask:

```
this.getInputs().addAll(collection);
```

Several solutions have been investigated and evaluated. Here is a short description of the most important ones:

1. A model redesign to move `inputs` and `outputs` references to the subclasses, Task, Transformation, and Choice, would have solved the problem elegantly, but at a price of three times the number of references. Task would have three one-to-one references for the InputPort, the OutputPort, and the FaultPort port. Transformation would have a one-to-many reference for the InputPort, a one-to-one reference for the OutputPort, and a one-to-one for the FaultPort port. Choice would have a one-to-one reference for the InputPort, a one-to-many reference for the OutputPort ports, and a one-to-one for the FaultPort port. The main problem with this approach is that the WorkflowNode loses its knowledge of Ports, so there is no easy way to loop on all the ports, or to connect an OutputPort, or a FaultPort port to an InputPort with an Edge.
2. The Java way of manually implementing the model would have required the `inputs` and `outputs` associations to be left at the same place and to be private. The corresponding accessor methods handling the many cardinalities of the reference would be private or protected. All subclasses would have to redefine the methods accessing the collection in order to enforce the constraints. Unfortunately, this solution cannot be implemented easily in EMF, because the serialization process requires the reference to be publicly accessible. There is no way to have a private reference in EMF.
3. The existence of a constraint language, integrated with the code generation tools taking could have been a good solution. We could have kept the associations at the WorkflowNode level and be able to express the constraints.
4. The solution we implemented has the following goals:
   a. To keep the model as designed in order to minimize the number of association and to benefit of the polymorphism for the ports

b. To not to use the default methods generated, including the one giving direct access to the underlying collection

c. To use the method we provided to support and enforce the application constraints

Figure 7-2 shows the resulting WorkFlowNode hierarchy.

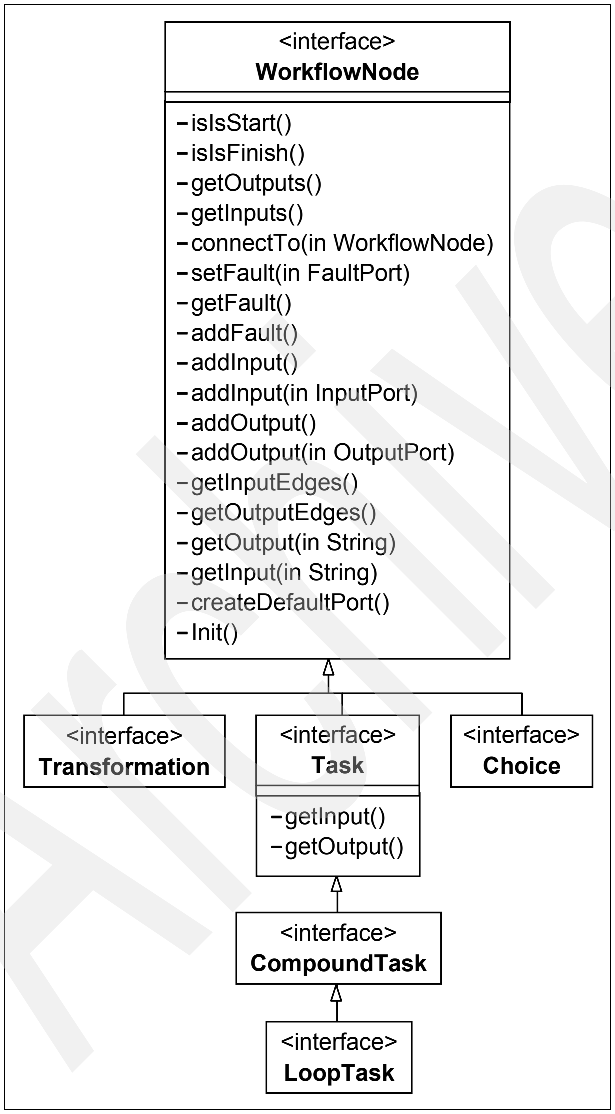


Figure 7-2 WorkflowNode hierarchy

Note: At the moment, the design can be split into three simple implementation lanes. The first is one input and one output; the second is with many inputs and one output; and the third is the one input and many outputs. Once we have more than one class in a lane, basically two classes with no direct inheritance in between, it would be nice to create an abstract intermediate class in order to provide one-one, many-one, or one-many behavior.

The connectTo method

The algorithm to connect an OutputPort port to an InputPort port with an Edge consists of these steps:

1. Checking if the link does not already exist.
2. Adding the Edge to the Workflow in order to have the object created in the Workflow entity context, because of the containment reference between them.
3. Linking the OutputPort to the Edge.
4. Linking the InputPort to the Edge.

The Java code for the connectTo method is found in the WorkflowImpl class as shown in Example 7-9:

Example 7-9 TaskImpl connectTo method

```
/**
* Connects the output port to the given input port.
* From an edge standpoint, the source is an output port
* and the target an input port.
*
* @param outputPort
* @param inputPort
* @param res
*/
public Edge connectTo(OutputPort outputPort, InputPort inputPort) {

   // Check to see if input and output are not already
   // connected by an edge.
   Edge edge = outputPort.findEdgeTo(inputPort);

   if (edge == null) {
      // No connection found
      WorkflowFactory workflowFactory = WorkflowModelManager.getFactory();

      // Create an edge
      edge = workflowFactory.createEdge();
      // Add the edge to the workflow, to benefit
```

```
        // of the containment link between workflow and edge
        this.getEdges().add(edge);

        // Link input and output to the edge
        inputPort.getEdges().add(edge);
        outputPort.getEdges().add(edge);
    }
    return edge;
}
```

The EMF eOpposite attribute of the eReferences entity is very helpful when making a connection between two ports with an edge, because for all the references with an eOpposite attribute, EMF keeps track of the changes on the other side of the reference. This means, for example, that if you add an Edge to an OutputPort:

```
outputPort.getEdges().add(edge);
```

Then EMF will do the opposite setup automatically and transparently for you:

```
edge.setSource(outputPort);
```

When creating an association in the EMF Class Diagram in the UML plug-in.

The Navigable checkbox (see Figure 1-12 on page 20) drives the access to the association features. Once an association is navigable on both ends, a change on one side is reflected on the other side, because the eReferences' eOpposite attributes are used.

7.4 Implementing the multi-page editor

When implementing a multi-page editor, there are several issues which have to be considered before and during development. This section gives you an introduction to a possible multi-page editor implementation. It also discusses some issues encountered during our development of the sample application.

Note: The source code of our sample application is available along with this book. See Appendix A, "Additional material" on page 225 for details on obtaining the sample code. We suggest that you study the code for implementation details. We tried to document it as often as possible. Because of that, we are not going to reproduce a lot of example code within this section. Instead, we give you an overview of the implementation and explain what and why implementation decisions were made.

Our multi-page editor consists of only two pages One page is for editing a whole workflow and the second page is for editing compound tasks of the same workflow. This provides an alternative way of editing compound tasks because in-place editing might not be suitable in all situations.

7.4.1 Getting started

First we start creating our multi-page editor by extending org.eclipse.ui.parts.MultiPageEditorPart. Thus, our main editor class is the class WorkflowEditor, which has to be registered as an Eclipse extension in the plugin.xml in the regular way.

The MultiPageEditorPart uses regular IEditorPart implementations or simple controls as editor pages. Because we expected that there will be some code that is shared between our editor pages, we created an abstract editor page AbstractEditorPage.

Note: In general, we reuse as much code as possible from the concepts described in Chapter 3, “Introduction to GEF” on page 87.

7.4.2 Sharing an EditDomain

One of the most important questions is what to share within your pages. However you decide to do this, you will have to consider some issues.

We decided not to share a single EditDomain within our editor pages. Our reasons were clear, because our editor would only have two pages. The functionality of each page might be similar, but the concept of each page is different.

On one page you should be able to edit the workflow, and on the second page you should edit the content of compound tasks. We thought that changes on one page should not affect the other page except for updating the UI. Thus, we wanted to have completely different undo/redo stacks for each page.

If you want all your pages to be using the same undo/redo stack (CommandStack), you will have to share the EditDomain between your pages.

Because of some current limitations in GEF, you have to think about solutions for the following issues:

- If you share an EditDomain within several pages, you have to remember that an EditDomain can have several EditPartViewers but only one palette.
- Thus, you might consider a concept of sharing one palette or attaching a new palette with every page switch.

7.4.3 The editor’s dirty state

You have two options for resolving this for a multi-page editor. Either you delegate this to every page or you implement this only once for the whole editor.

We decided to implement this directly into the multi-page editor because we think it might be less expensive to calculate this once for all pages rather than letting each page calculate this itself and asking each page.

The concept is basically the same as we would have used for a simple editor. The editor listens for CommandStack changes and updates its dirty state according to the state of the CommandStack.

Our WorkflowEditor provides a MultiPageCommandStackListener, which is capable of listening to multiple CommandStacks. All CommandStacks that need to be observed can be registered to it. We do this at the same time we create our pages.

7.4.4 Actions

Our multi-page editor provides one ActionRegistry for the whole editor. Thus, all actions are available on all pages. We don’t need to have different actions for different pages. Again the concept is similar to a single editor. Actions are registered to an ActionRegistry.

The ActionBarContributor

The GEF ActionBarContributor is not able to provide support for tracking page changes in a multi-page editor. If you need this, you can either implement the functionality from org.eclipse.ui.part.MultiPageEditorActionBarContributor or inherit from this class. But if you inherit from this class, you don’t have the action handling support provided by the GEF ActionBarContributor.

7.4.5 Support for the properties view

The base concept is similar to a single editor. Our multi-page editor uses the undoable property sheet root entry provided by GEF. But this is only capable of committing to one CommandStack. If you share only one EditDomain within all pages, there is no special work necessary and you can stop here.

Due to internal caching in Eclipse, it is not possible to have a separate property sheet page for every single page. There can be only one for the whole editor. But somehow the property sheet page needs to keep track of the active page to commit to the correct CommandStack.

We are handling this with a workaround. Our undoable property sheet page root entry gets a delegating CommandStack. The DelegatingCommandStack is a CommandStack that delegates work to a current CommandStack, which can be changed. Thus, we only need to update the DelegatingCommandStack when the current page changes, and this can be easily done from within our multi-page editor.

7.4.6 The outline view

We had a little bit more work to do for the outline view, but basically the concept is the same as seen before. The outline view is updated every time the page changes. Although it is strongly connected to our multi-page editor, we tried to keep the implementation as generic as possible to allow you to reuse it for your projects.

The implementation can be found in WorkflowEditorOutlinePage. It provides both a tree outline and an overview figure. You only need to call one method on each page change to reinitialize the outline view with a new content. This can be easily done from within our multi-page editor.

7.4.7 The palette

Each page has its own PaletteViewer. You can’t share one PaletteViewer instance within several pages. It is possible to have only one PaletteViewer for the whole editor, but this must be implemented in the multi-page editor class because the SWT control needs to be created there.

However, having multiple PaletteViewers is no issue because you can share a single PaletteRoot between them, like we did. Our multi-page editor provides the same PaletteRoot for every page. If you would like to have different PaletteRoots for your pages, this is no problem either. You just have to implement it that way.

A

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG246302`

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246302.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

File name	*Description*
sg246302.zip	Zipped Code Samples

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	15 MB minimum
Operating System:	Windows
Processor:	1Ghz or higher
Memory:	1GB or higher

How to use the Web material

Unzip the contents of the Web material zip files into the plug-in folder of your Eclipse SDK. The material is organized around the chapters and sections of our redbook, and the source code can be imported into your Eclipse SDK as an existing project.

The additional material is organized by chapter and sections within a chapter. Most of the material is in the form of Eclipse projects that you can import into your Eclipse workbench. After you unzip the sg246302.zip file, you will have four main folders created:

1. **emf-examples:**

   This folder contains Eclipse projects for the examples described in Chapter 2, “EMF examples” on page 29.

   Each major section of Chapter 2, “EMF examples” on page 29 has a matching Eclipse project. The projects are cumulative and they also depend on your having completed the modelling and code generation described in Chapter 1, “Introduction to EMF” on page 3. You will need to make sure that you have created the Java build path variables described in 1.3.9, “Compiling the code” on page 27, otherwise you may get classpath errors when importing the sample projects.

2. **gef-intro:**

   This folder contains Eclipse projects for the examples described in Chapter 3, “Introduction to GEF” on page 87. If you import the sample projects you will need to set up the Eclipse environment as described in “Eclipse Classpath settings for sample projects” on page 227.

3. **emf-with-gef:**

   This folder contains Eclipse projects for the examples described in Chapter 5, “Using GEF with EMF” on page 165.

   Each major section of Chapter 5, “Using GEF with EMF” on page 165 has a matching Eclipse project. Also be sure to import the appropriate model project for the editor project you want to work with. For example, to work with the NetworkEditor project, you must also import the NetworkEditorModel project.

Some of the sample projects in this chapter also expect that you have the SAL330RWorkflowModel project in your workspace. You may have created this project by working through the examples described in Chapter 1, "Introduction to EMF" on page 3, or you can import this from our redbook sample material. If you import the SAL330RWorkflowModel project, you will need to set up the Eclipse environment as described in "Eclipse Classpath settings for sample projects" on page 227. If you create the SAL330RWorkflowModel project, you will need to make sure that you have created the Java build path variables described in 1.3.9, "Compiling the code" on page 27, otherwise you may get classpath errors when importing the sample projects.

4. **sample-application:**

   This folder contains code for the sample application described in Chapter 7, "Implementing the sample" on page 203. We provide two zip files:

   – workflow-sample-plugins-1.0.0.zip:

     This is our redbook sample application packaged as a plug-in for install in Eclipse. To us this plug-in, unzip the archive to the directory where you installed Eclipse. You can then experiment with the functions of our sample workflow editor by create a new file resource with a .workflow extension.

   – workflow-sample-src-1.0.0.zip:

     This is the zipped source code for our redbook sample application. When you unzip this archive, two Eclipse project folders are created with projects that can be imported into Eclipse: SAL330RGEFDemoApplication, which is the sample editor code; and SAL330RWorkflowModel, which is the associated workflow model used by our redbook sample editor. If you import the sample projects, you will need to set up the Eclipse environment as described in "Eclipse Classpath settings for sample projects" on page 227.

Eclipse Classpath settings for sample projects

When you add our redbook sample application to your Eclipse workbench you need to make sure that all the required plug-ins can be found. There are two alternate ways to set up your environment to do this:

1. Import external features and plug-ins:

   a. Set up your workspace with plugins as the last folder.
      For example, use d:\sampleapp\plugins

   b. Start Eclipse.

c. Choose **File -> Import**.

d. Check **External Features** and click **Next**.

e. Accept the default **Choose from features in run-time workbench** and click **Next**.

f. Select the features you need to import, including:

- org.eclipse.platform
- org.eclipse.platform.win32
- org.eclipse.jdt
- org.eclipse.emf
- org.eclipse.gef

g. Click **Finish**.

h. Import any missing plug-ins by choosing **File -> Import... -> External Plug-ins and Fragments**.

i. Make sure to choose **Copy plug-in contents into workspace location**.

j. Select all the required plug-ins and click **Finish**.

Note: The tasks view will list all plug-ins that are missing from the required classpath of the sample project you import.

2. Configure the target platform:

a. Workspace folder has no special naming requirement.

b. Start Eclipse.

c. Choose **Window -> Preferences**.

d. Choose **Plug-in Development -> Target Platform**.

e. Select **this application** and click either **Not In Workspace** or **Select All**.

f. Click **OK**.

g. Select the plugin.xml file of the imported sample project, right-click and choose **Update Classpath**.

h. Select all the plug-ins that need their classpath updated.

i. Click **Finish**.

Abbreviations and acronyms

ADL	Architectural Description Language
DTD	Document Type Definition
EAI	Enterprise Application Integration
EJB	Enterprise Java Bean
EMF	Eclipse Modeling Framework
FAQ	Frequently Asked Questions
GEF	Graphical Editing Framework
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IBM	International Business Machines Corporation
IDE	Integrated Development Environment
IDL	Interface Definition Language
ITSO	International Technical Support Organization
JET	Java Emitter Templates
MDA	Model Driven Architecture
MDE	Model Driven Environment
MOF	Meta Object Facility
MVC	model-view-controller
NLS	National Language Support
NLS	National Language Support
OMG	Object Modelling Group
OVID	Object View and Interaction Diagram
SWT	Standard Widget Toolkit
SWT	Standard Widget Toolkit
URI	Universal Resource Identifier
XSD	XML Schema definition

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

Other publications

These publications are also relevant as further information sources:

- *The Java Developer's Guide to Eclipse, Sherry Shavor et al,* Addison Wesley, ISBN: 0-321-15964-0
- *Eclipse Modeling Framework, Frank Budinsky et al,* Addison Wesley, ISBN: 0131425420

Online resources

These Web sites and URLs are also relevant as further information sources:

- eclipse.org main page:

  `http://www.eclipse.org`
- Eclipse Modeling Framework home page:

  `http://www.eclipse.org/emf`
- Graphical Editing Framework home page:

  `http://www.eclipse.org/gef`
- Omondo EclipseUML page:

  `http://www.eclipseuml.com`
- Object, view and interaction design:

  `http://www-3.ibm.com/ibm/easy/eou_ext.nsf/Publish/589`
- Eclipse XML Schema Infoset Model:

  `http://www.eclipse.org/xsd`
- XML Metadata Interchange:

  `http://www.omg.org/technology/documents/formal/xmi.htm`
- JET tutorial part 1:

  `http://eclipse.org/articles/Article-JET/jet_tutorial1.html`

- JET tutorial part 2:

  `http://eclipse.org/articles/Article-JET2/jet_tutorial2.html`
- Eclipse Wiki:

  `http://eclipsewiki.swiki.net`
- Metanology MDE:

  `http://www.metanology.com`
- Eclipse designer plug-in:

  `http://eclipsedesigner.sourceforge.net`
- eSuite project:

  `http://jeez.sourceforge.net`

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads:

ibm.com/support

IBM Global Services:

ibm.com/services

Index

IBM

Redbooks

Eclipse Development using the Graphical Editing Framework & the Eclipse Modeling Framework

(0.5" spine)
0.475"<->0.875"
250 <-> 459 pages

Eclipse Development

using the Graphical Editing Framework and the Eclipse Modeling Framework

Understanding the GEF and EMF frameworks

Developing with GEF and EMF

Code examples

Eclipse Development using the Graphical Editing Framework and the Eclipse Modelling Framework is written for developers who use the Eclipse SDK to develop plug-in code. This IBM Redbook is intended for a technical readership and for developers who already have good knowledge and experience in Eclipse plug-in development.

In this book, we examine two frameworks that are developed by the Eclipse Tools Project for use with the Eclipse Platform: the Graphical Editing Framework (GEF), and the Eclipse Modeling Framework (EMF). We cover both the Graphical Editing Framework and the Eclipse Modeling Framework, but these frameworks can be used separately, and there is no dependency between them.

This book provides a high level introduction to these frameworks so that Eclipse plug-in developers can consider whether the frameworks will be useful for the requirements of their particular development environment. Next, tips and techniques are provided for writing code that uses GEF and EMF. Also, a detailed example is developed to illustrate a GEF editor that uses an EMF model.

SG24-6302-00 ISBN 0738453161