

Foreword by Sridhar Iyengar



eclipse

Modeling Framework

Frank Budinsky • David Steinberg
Ed Merks • Ray Ellersick • Timothy J. Grose

SERIES EDITORS ▶ Erich Gamma • Lee Nackman • John Wiegand

Table of Contents

Copyright.....	1
The Eclipse Series.....	3
Foreword.....	4
Foreword.....	6
Preface.....	8
References.....	13
Part: I EMF Overview.....	14
Chapter 1. Eclipse.....	15
Section 1.1. The Projects.....	15
Section 1.2. The Eclipse Platform.....	16
Section 1.3. More Information.....	18
Chapter 2. Introducing EMF.....	19
Section 2.1. Unifying Java, XML, and UML.....	19
Section 2.2. Modeling vs. Programming.....	21
Section 2.3. Defining the Model.....	22
Section 2.4. Generating Code.....	27
Section 2.5. The EMF Framework.....	31
Section 2.6. EMF and Modeling Standards.....	38
Chapter 3. Model Editing with EMF.Edit.....	40
Section 3.1. Displaying and Editing EMF Models.....	40
Section 3.2. Item Providers.....	44
Section 3.3. Command Framework.....	49
Section 3.4. Generating EMF.Edit Code.....	56
Chapter 4. Using EMF—A Simple Overview.....	60
Section 4.1. Example Model: The Primer Purchase Order.....	60
Section 4.2. Creating EMF Models and Projects.....	61
Section 4.3. Generating Code.....	81
Section 4.4. Running the Application.....	83
Section 4.5. Continuing Development.....	86
Part: II Defining EMF Models.....	88
Chapter 5. Ecore Modeling Concepts.....	89
Section 5.1. Core Model Uses.....	89
Section 5.2. The Ecore Kernel.....	90
Section 5.3. Structural Features.....	91
Section 5.4. Behavioral Features.....	95
Section 5.5. Classifiers.....	96
Section 5.6. Packages and Factories.....	100
Section 5.7. Annotations.....	101
Section 5.8. Modeled Data Types.....	102
Chapter 6. Java Source Code.....	104
Section 6.1. Java Specification for Packages.....	104
Section 6.2. Java Specification for Classes.....	105
Section 6.3. Java Specification for Enumerations.....	109
Section 6.4. Java Specification for Data Types.....	110
Section 6.5. Java Specification for Maps.....	111
Chapter 7. XML Schema.....	114
Section 7.1. Schema Definition of Packages.....	115
Section 7.2. Schema Definition of Classes.....	116
Section 7.3. Schema Definition of Attributes.....	118
Section 7.4. Schema Definition of References.....	120
Section 7.5. Schema Simple Types.....	121
Chapter 8. UML.....	123
Section 8.1. UML Packages.....	123
Section 8.2. UML Specification for Classifiers.....	124
Section 8.3. UML Specification for Attributes.....	126
Section 8.4. UML Specification for References.....	129
Section 8.5. UML Specification for Operations.....	133
Part: III Using the EMF Generator.....	135
Chapter 9. EMF Generator Patterns.....	136
Section 9.1. Modeled Classes.....	136
Section 9.2. Attributes.....	139
Section 9.3. References.....	148
Section 9.4. Operations.....	158
Section 9.5. Class Inheritance.....	159
Section 9.6. Reflective Methods.....	161
Section 9.7. Factories and Packages.....	168
Section 9.8. Switch Classes and Adapter Factories.....	170

Section 9.9. Customizing Generated Classes.....	172
Chapter 10. EMF.Edit Generator Patterns.....	175
Section 10.1. Item Providers.....	176
Section 10.2. Item Provider Adapter Factories.....	186
Section 10.3. Editor.....	189
Section 10.4. Action Bar Contributor.....	191
Section 10.5. Wizard.....	192
Section 10.6. Plug-Ins.....	193
Chapter 11. Running the Generators.....	195
Section 11.1. EMF Code Generation.....	195
Section 11.2. The Generator GUI.....	198
Section 11.3. The Command-Line Generator Tools.....	205
Section 11.4. The Template Format.....	208
Chapter 12. Example—Implementing a Model and Editor.....	211
Section 12.1. Getting Started.....	211
Section 12.2. Generating the Model.....	213
Section 12.3. Implementing Volatile Features.....	213
Section 12.4. Implementing Data Types.....	215
Section 12.5. Running the ExtendedPO2 Editor.....	217
Section 12.6. Restricting Reference Targets.....	219
Section 12.7. Splitting the Model into Multiple Packages.....	222
Section 12.8. Editing Multiple Resources Concurrently.....	227
Part: IV Programming with EMF.....	233
Chapter 13. EMF Client Programming.....	234
Section 13.1. Packages and Factories.....	234
Section 13.2. The EMF Persistence API.....	238
Section 13.3. EMF Resource Implementations.....	245
Section 13.4. Adapters.....	256
Section 13.5. Working with EMF Objects.....	265
Section 13.6. Dynamic EMF.....	270
Chapter 14. EMF.Edit Programming.....	275
Section 14.1. Overriding Commands.....	275
Section 14.2. Customizing Views.....	280
Part: V EMF API.....	299
Chapter 15. The org.eclipse.emf.common Plug-In.....	300
Section 15.1. The org.eclipse.emf.common Package.....	300
Section 15.2. The org.eclipse.emf.common.command Package.....	302
Section 15.3. The org.eclipse.emf.common.notify Package.....	309
Section 15.4. The org.eclipse.emf.common.util Package.....	312
Chapter 16. The org.eclipse.emf.common.ui Plug-In.....	332
Section 16.1. The org.eclipse.emf.common.ui Package.....	332
Section 16.2. The org.eclipse.emf.common.ui.celleditor Package.....	334
Section 16.3. The org.eclipse.emf.common.ui.viewer Package.....	337
Chapter 17. The org.eclipse.emf.ecore Plug-In.....	339
Section 17.1. The org.eclipse.emf.ecore Package.....	339
Section 17.2. The org.eclipse.emf.ecore.plugin Package.....	355
Section 17.3. The org.eclipse.emf.ecore.resource Package.....	356
Section 17.4. The org.eclipse.emf.ecore.util Package.....	362
Chapter 18. The org.eclipse.emf.ecore.xmi Plug-In.....	396
Section 18.1. The org.eclipse.emf.ecore.xmi Package.....	396
Part: VI EMF.Edit API.....	405
Chapter 19. The org.eclipse.emf.edit Plug-In.....	406
Section 19.1. The org.eclipse.emf.edit Package.....	406
Section 19.2. The org.eclipse.emf.edit.command Package.....	407
Section 19.3. The org.eclipse.emf.edit.domain Package.....	424
Section 19.4. The org.eclipse.emf.edit.provider Package.....	427
Section 19.5. The org.eclipse.emf.edit.provider.resource Package.....	447
Section 19.6. The org.eclipse.emf.edit.tree Package.....	449
Section 19.7. The org.eclipse.emf.edit.tree.provider Package.....	451
Section 19.8. The org.eclipse.emf.edit.tree.util Package.....	452
Chapter 20. The org.eclipse.emf.edit.ui Plug-In.....	454
Section 20.1. The org.eclipse.emf.edit.ui Package.....	454
Section 20.2. The org.eclipse.emf.edit.ui.action Package.....	455
Section 20.3. The org.eclipse.emf.edit.ui.celleditor Package.....	461
Section 20.4. The org.eclipse.emf.edit.ui.dnd Package.....	463
Section 20.5. The org.eclipse.emf.edit.ui.provider Package.....	466
Appendix A: UML Notation.....	472
Classes and Interfaces.....	472
Enumerations and Data Types.....	473
Class Relationships.....	473
Appendix B: Summary of Example Models.....	476
SimplePO.....	476
PrimerPO.....	478

ExtendedPO1.....	483
ExtendedPO2.....	483
ExtendedPO3.....	484

Copyright

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

U.S. Corporate and Government Sales

(800) 382-3419

<corpsales@pearsontechgroup.com>

For sales outside of the U.S., please contact:

International Sales

(317) 581-3793

<international@pearsontechgroup.com>

Visit Addison-Wesley on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

Eclipse modeling framework : a developer's guide / Frank Budinsky ... [et al.]

p. cm. -- (The eclipse series)

ISBN 0-13-142542-0

1. Computer software--Development. 2. Java (Computer program language) I.

Budinsky, Frank. II. Series.

QA76.76.D47E295 2003

005.1--dc21

2003052408

Copyright © 2004 by Pearson Education, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.

Rights and Contracts Department

75 Arlington Street, Suite 300

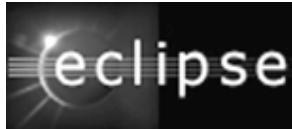
Boston, MA 02116

Fax: (617) 848-7047

First printing, August 2003

Licensed by
Isaac Griffith
1942365

The Eclipse Series



SERIES EDITORS Erich Gamma • Lee Nackman • John Wiegand

Eclipse is a universal tool platform, an open extensible integrated development environment (IDE) for anything and nothing in particular. Eclipse represents one of the most exciting initiatives hatched from the world of application development in a long time, and it has the considerable support of the leading companies and organizations in the technology sector. Eclipse is gaining widespread acceptance in both the commercial and academic arenas.

The Eclipse Series from Addison-Wesley is the definitive series of books dedicated to the Eclipse platform. Books in the series promise to bring you the key technical information you need to analyze Eclipse, high-quality insight into this powerful technology, and the practical advice you need to build tools to support this evolutionary Open Source platform. Leading experts Erich Gamma, Lee Nackman, and John Wiegand are the series editors.

Titles in the Eclipse Series

Kent Beck and Erich Gamma, *Contributing to Eclipse*, 0-321-20575-8

Frank Budinsky David Steinberg Ed Merks Ray Ellersick and Timothy J. Grose, *Eclipse Modeling Framework*, 0-131-42542-0

Eric Clayberg and Dan Rubel, *Eclipse: Building Commercial-Quality Plug-Ins*, 0-321-22847-2

For more information on books in this series visit www.awprofessional.com/series/eclipse

Foreword

By Sridhar Iyengar*

The past few years have seen exciting advances in software that promise to revolutionize how software applications and tools will be conceived, designed, modeled, constructed, integrated, monitored, and managed using open source technologies (such as Eclipse and Apache) and open standards (such as Java, J2EE, UML, XML, and MDA). While individual best-of-breed tools (a compiler, modeling tool, testing tool, business integration tool, and so on) are still important, how these tools work together to enable rapid design, development, management, and monitoring of applications will be even more important as developers and customers will continue to be challenged to “do more with less.” In this challenging environment we need integration technology that enables these tools and applications to be used together to improve the software development process. This book details such an integration technology, one that unifies open source and open standards: the Eclipse Modeling Framework (EMF) unifies Java, XML, and UML technologies so that they can be used together to build better integrated software tools.

In less than two years, Eclipse has grown from a tools integration framework for Java development tools to a rich platform that is spurring innovation across the software development and integration life cycle: from Aspect-Oriented Programming (AOP) to Model Driven Architecture (MDA) development and testing, from Java development to Web services development, and more. The year 2002 saw an important extension to Eclipse: a powerful new framework that enables code generation, model and metadata management, and promises to accelerate a new generation of better integrated tools and applications.

EMF is not simply a powerful framework for code generation. It also plays a central role in how various tools will extend the plug-in architecture of Eclipse to better share data and semantics using a set of shared models across the software development life cycle. These models are all described using the EMF Ecore model, which can be described with just a few simple well understood Java and UML concepts used in the class model. Several Eclipse consortium members, including IBM, have used EMF to build integrated tools that support J2EE, Web services, and more recently OMG MDA. See www.eclipse.org for a complete listing of projects and the large number of plug-ins that are being added regularly.

While the challenges in accomplishing this vision of an end-to-end integrated software life cycle are enormous, we have learned a few lessons and best practices that I believe are worth reinforcing. I occasionally call these “metalessons” of the emerging Model Driven Architecture (see www.omg.org/mda for more information):

*Sridhar Iyengar, a Distinguished Engineer in the IBM Software Group, is focused on using models and metadata for the integration software life cycle tools, application, data, and frameworks that enable flexible integration of business applications and the underlying software infrastructure. Sridhar works with software development teams inside IBM and in the software community at large—including IBM partners, systems integrators, customers, and the Eclipse community—to accelerate the use of modeling and metadata based on open standards (from OMG, W3C, and OASIS) and open source frameworks (specifically EMF and derivative applications). He is the original architect of the OMG Meta Object Facility (MOF) and the OMG XML Metadata Interchange (XMI), which together with UML form the core of OMG Model Driven Architecture (MDA). Sridhar represents IBM at the OMG where he serves on the OMG Architecture Board and the OMG Board of Directors. He holds several patents in modeling, metadata, and tool integration frameworks and has a master's degree in computer science.

- The use of models (UML being a prime example) to document, share, and communicate application designs across the stakeholders in an organization. UML helps us better understand, design, and communicate about software and systems. It enables portable designs.
- The use of metadata (OMG MOF and XMI, which build on W3C XML, are prime examples) to represent these software artifacts as well as to exchange them between tools and applications on a desktop or across the Internet to enable dynamic integration. The use of reflection in Java and EMF Ecore is an excellent example of how metadata can be used to enable flexible development and integration of tools. The use of XML to define and manipulate portable metadata is now an accepted fact.
- The use of model transformations and model-to-code transformations using design patterns and templates. Several common Java and UML design patterns are used to generate Java interfaces, Java code, XML documents, and XML Schemas. The whole idea of using models and metadata to enable predictable model and code transformations is emerging as a central tenet of MDA.
- The use of models, metadata, and model/code transformations to improve software development, integration, and management at the business level (these models are called platform-independent models) or at the technology level (these models are called platform-specific models) while enabling predictable traceability and transformation between these levels.

The Eclipse Modeling Framework project is a significant step that promises to speed up this revolution by showing how pragmatic use of the key concepts described above (without getting bogged down in the inherent complexity of some of the standards) can lead to significant benefits as we strive to extend Eclipse into a more complete software development platform. In reading this book (and downloading the accompanying software), you will learn to use modeling, metadata, and code generation, as well as model transformation techniques. Once you have used EMF, explore other Eclipse projects such as the XSD project (www.eclipse.org/xsd) or the Hyades project (www.eclipse.org/hyades) to see how EMF is used to build powerful and practical tools. Innovate by extending Eclipse into an even more complete software environment.

The authors Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, and Tim Grose have deep experience in building J2EE and Web services software tools, metadata frameworks, and code generators and show by example that "less is better"! The use of a few powerful features of Java and UML to bootstrap a code-generation framework to build better integrated tools has now been proven in commercial software such as IBM WebSphere tools. Learn how to use EMF to extend Eclipse and build tools that will work better together. This book comes at a time when there is a thirst for knowledge about Eclipse technologies.

I hope you learn and gain as much from reading this book as I did.

Sridhar Iyengar

IBM Distinguished Engineer, WebSphere Software

Member OMG Architecture Board

July 2003

Foreword

By Dr. Lee R. Nackman*

EMF is a powerful, practical technology for implementing object models from formal model definitions that has had a profound effect on IBM runtimes and tools, especially the new suite of IBM tools based on Eclipse.

When IBM started work on the technology that ultimately became Eclipse, extensibility and integration were paramount goals. We wanted Eclipse to become the platform of choice for tool providers, be they tool or software vendors, academics, or commercial enterprises. We wanted these many different people to be able independently to extend Eclipse in ways we couldn't even imagine. And we wanted it to be possible for these independently developed extensions to cooperate to give the appearance that the extensions were designed to work together.

We knew that achieving these goals required user interface integration—the ability for independent extensions to contribute to a single user interface—and asset integration—the ability for the persistent files produced and used by various extensions to be managed by common resource management mechanisms. Eclipse was therefore designed from the ground up to be extended via a powerful plug-in mechanism and an associated plug-in development environment. In fact, the base Eclipse platform does very little beyond hosting plug-ins. These plug-ins provide nearly all of Eclipse's many capabilities. For example, the Eclipse Java Development Tools (JDT) are implemented entirely as a collection of plug-ins. Moreover, the JDT defines extension points so that the JDT itself can be extended.

Eclipse has been remarkably successful in achieving our goals. The JDT is widely used as a Java IDE, there are dozens of commercial products built by extending Eclipse and the JDT, there are many commercial products that extend other commercial products, and there are hundreds of Eclipse plug-ins being developed in open source both at eclipse.org and elsewhere. A vibrant Eclipse ecosystem is developing.

Despite this success, there has been something missing: data integration. How can independently developed plug-ins define and share common data? The answers are obvious: plug-in developers can define a common XML Schema, or they can write some Java classes, or maybe they can define a UML model. There are too many answers and, regardless of which you choose, it is hard work to implement the shared data, figure out how to persist it, and write editors and viewers for it. All of these practical difficulties distract attention from the important work of defining the semantics of the data to be shared. EMF—the Eclipse Modeling Framework—comes to the rescue.

*Dr. Lee R. Nackman leads development of IBM's application development tool, compiler, and Java infrastructure products. He initiated the project that became Eclipse and he initiated and led development of IBM's WebSphere Studio tool products. He serves as a founding member of the Board of Stewards of eclipse.org, and has published 50 papers and a book.

With EMF, you can model shared data as XML Schema, as Java classes, or as UML models. Whichever you choose to use, EMF can generate the others. EMF also generates a Java implementation that persists the data as (linked) XML files. With a bit more work, EMF can generate Eclipse editors and viewers for the models you specify. And, of course, the EMF tools are themselves Eclipse plug-ins. Bottom line: EMF takes you past the mechanics of sharing data and lets you focus on getting the semantics right and standardized. In fact, several open source projects are defining and implementing EMF models now.

EMF is not some pie-in-the-sky, untested idea. EMF and its precursors have been used in several generations of IBM's WebSphere Studio products and to share data between WebSphere Studio and other products. It is eminently practical.

The authors write with authority. They have all been deeply involved in building and applying EMF and its precursors. They know the code and they know how it should be used. I urge you to download Eclipse, install the EMF plug-ins, and put it to use. This book will be your guide.

Lee R. Nackman, Ph.D.

Vice President, Application Development Tools, IBM Software Group

Member, eclipse.org Board of Stewards

Preface

This book is a comprehensive introduction to and developer's quick reference for the Eclipse Modeling Framework (EMF). EMF is a powerful framework and code-generation facility for building Java applications based on simple model definitions. Designed to make modeling practical and useful to the mainstream Java programmer, EMF unifies three important technologies: Java, XML, and UML. Models can be defined using a UML modeling tool, an XML Schema, or by specifying simple annotations on Java interfaces, whereby programmers write the abstract interfaces (a small subset of what they would normally need to write), and the rest is generated automatically and merged back into their existing code.

By relating modeling concepts to the simple Java representations of those concepts, EMF has successfully bridged the gap between modelers and Java programmers. It serves as a gentle introduction to modeling for Java programmers and at the same time as a reinforcement of the modeler's theory that plenty of Java coding can be automated, given an appropriate tool. This book shows how EMF is such a tool. At the same time, it also shows how using EMF gives you much more than just automatic code generation.

While Eclipse provides a powerful platform for integration at the UI and file level, EMF builds on this capability to enable applications to integrate at a much finer granularity than would otherwise be possible. EMF-based modeling is the foundation for fine-grained interoperability and data sharing among tools and applications in Eclipse. All of the features provided by the EMF framework, combined with an intrinsic property of modeling—that it provides a higher-level description that can more easily be shared—provide the needed ingredients to foster such data integration. A number of companies are already using both Eclipse and the EMF modeling technology as the foundation for commercial products. IBM's WebSphere Studio, for example, is completely based on Eclipse, and most of its tools use EMF to model their data.

This book assumes the reader is familiar with object-oriented programming concepts, and specifically with the Java programming language. Previous exposure to modeling techniques such as UML class diagrams, although helpful, is not required. Part I (Chapters 1 to 4) provides a basic overview of the most important concepts in EMF and modeling. This part teaches someone with basic Java programming skills everything needed to start using EMF to model and build an application. Part II (Chapters 5 to 8) presents a thorough overview of EMF's metamodel, Ecore, followed by details of the mappings between Ecore and the other supported model-definition forms: annotated Java, XML Schema, and UML. Part III (Chapters 9 to 12) includes detailed analyses of EMF's code-generator patterns and tools, followed by an end-to-end example of a non-trivial EMF application. Part IV (Chapters 13 and 14) provides a more in-depth analysis of the EMF and EMF.Edit frameworks, including discussions of design alternatives and examples of common framework customizations and programming techniques. Part V (Chapters 15 to 18) and Part VI (Chapters 19 and 20) finish off the book with a complete API quick reference for all of the classes and methods in the 1.1 versions of the core EMF and EMF.Edit frameworks.

Conventions Used in This Book

Parts I through IV constitute a comprehensive introduction to EMF. The following formatting conventions are used throughout these parts:

Bold—Used for the names of model elements, such as packages, classes, attributes, and references; and of user-interface elements, including menus, buttons, tabs, and text boxes.

Italic—Used for filenames and URLs, as well as for placeholder text that is meant to be replaced by a particular name. Also, new terms are often italicized for emphasis.

Courier—Used for all code samples and for in-text references to code elements, including the names of Java packages, classes, interfaces, methods, fields, variables, and keywords. Plug-in names and elements of non-Java files, including XML, also appear in this font.

Courier Bold—Used to emphasize portions of code samples, usually new insertions or changes.

Courier ~~Strikethrough~~—Used in code samples to indicate that text should be deleted.

Parts V and VI provide a quick reference for the EMF API, comprising quick-reference entries for each client-accessible class and interface in the six most significant EMF plug-ins. Each entry includes a synopsis that is automatically generated from source and class files, as well as a description that explains the significance of the class or interface and highlights its most important members. The following formatting conventions are used in these parts:

Bold—Used in descriptions for the names of model elements, such as packages, classes, attributes, and references; and of user-interface elements, including menus, buttons, tabs, and text boxes.

Helvetica Narrow—Used for synopses and in descriptions for references to code elements.

Helvetica Narrow Bold—Used in synopses to highlight class and member names.

Helvetica Narrow Italic—Used for entry section headings and in synopses for flags on classes and members.

The format for the quick reference is based quite closely on the one introduced by David Flanagan in *Java in a Nutshell*[a]. Though details have been tweaked, primarily to incorporate Eclipse's notion of plug-ins, readers familiar with Flanagan's series should be very comfortable with it.

Each chapter covers one plug-in in the EMF API, including the corresponding package and its sub-packages. It begins with an overview of the plug-in and a summary of its dependencies and the extension points that it offers and extends, as declared by its plug-in manifest file. Note that these are Eclipse concepts; they are explained in greater detail in Chapter 1. This summary may include the following three headings, as appropriate:

Requires—Lists the plug-in dependencies, which must be available in order for the plug-in to run under Eclipse. The IDs of such plug-ins are specified in a comma-separated list. Any dependencies of those plug-ins are also listed, recursively, within parenthesis. An arrow indicates that the preceding plug-in is exported.

Extension Points—Lists the IDs of extension points that the plug-in declares in order to offer functionality to other plug-ins.

Extensions—Lists the IDs of extension points that the plug-in extends, qualified by the IDs of the plug-ins that declare them.

The chapters are divided into sections corresponding to the packages defined in a plug-in, which are ordered alphabetically. Each section includes a short description of the package, followed by a quick-reference entry for every public class and interface in the package. These entries are, again, ordered alphabetically by name.

Each entry begins with a two-line heading that gives the class (or interface) name, the package to which it belongs, and zero or more of the following flags, as appropriate:

adapter—The class is derived from the EMF `Adapter` interface, allowing it to receive notifications.

checked—The class is derived from Java's `Exception` class, but not `RuntimeException`. In other words, it is an exception that must be declared in the `throws` clause of any method that may throw it.

collection—The class is derived from the Java `Collection` interface.

eobject—The class is derived from the EMF `EObject` interface, indicating that it represents a modeled object.

notifier—The class is derived from the EMF `Notifier` interface, allowing it to send change notifications.

unchecked—The class is derived from Java's `RuntimeException` class. In other words, it is an exception that a method may throw without declaring in its `throws` clause.

Following the heading is a description of the class. The description usually explains where it fits in the overall design of EMF, points out any common patterns that it follows, refers to other closely related classes, and highlights its most important methods and fields.

The synopsis, which follows the description, describes the signature of the class and its members in a format resembling an interface definition. Java programmers should find it quite easy to read.

The first line contains information about the class, itself. It begins with its visibility, `public` or `protected` (note that no classes or members with `private` or `default/package` visibility are included in the API). The visibility is followed by any other applicable modifiers from among the following: `static`, `abstract`, and `final`. Next, the class name is given, and then any `extends` and `implements` clauses that it declares.

The remainder of the synopsis lists the public and protected members of the class: its inner classes, methods, and fields. These members are broken into groups and listed alphabetically within those groups. Each group is introduced by a Java comment. The different groups are as follows:

Constructors—Contains the constructors for a class, further subdivided into two groups according to visibility. Note that a class may declare a private default constructor to enforce non-instantiability; if so, a special empty grouping called “No Constructor” is included.

Constants—Includes fields declared as both `static` and `final`, which constitute constant values, typically used as valid parameter or return values for certain methods. Again, they are subdivided by visibility into two groups.

Inner Classes—Includes member classes and interfaces, divided by visibility. Each inner class also receives its own complete quick-reference entry, which follows that of the containing class.

Static Methods—Contains class, or `static`, methods, which are typically utilities for dealing with instances of the class. They are also subdivided by visibility into two groups.

Property Accessors—Contains public accessor methods for the properties of a class, which typically correspond in EMF-modeled classes to attributes and references. Public methods that follow the usual naming pattern—beginning with “get” or “is” and “set”—and have appropriate parameter and return types are grouped and alphabetized by property name, rather than method name.

Public Instance Methods—Contains all public instance methods that do not appear elsewhere.

Overriding Methods—Multiple groups, each corresponding to a base class, contain the public methods that override base implementations. If such a method is also a property accessor, it appears in that group instead, and a flag is used to identify it as an overriding method.

Implementing Methods—Multiple groups, each corresponding to an implemented interface, contain the methods for which the class provides public implementations. If such a method is also a property accessor, it appears in that group instead, and a flag is used to identify it as an implementing method.

Protected Instance Methods—Contains all protected instance methods that do not appear elsewhere.

Class Fields—Contains all static, non-final fields, subdivided by visibility into two groups.

Instance Fields—Contains all instance fields, again subdivided by visibility.

Groups appear in the above order. Any empty groups, except for “No Constructors,” are omitted.

Each member is summarized in one line. For a method, this line can include the visibility, additional modifiers, type, method name, parameters, `throws` clause, and flags. For a class or field, there can be no parameters or `throws` clause. A member’s flags may include the following:

Overrides:—The method overrides a base class implementation. This flag is followed by the name of that base class. It is shown only when the method is a property accessor or a protected method; otherwise, the same information is conveyed by its grouping.

Implements:—The method implements an interface method definition. The flag is followed by the name of the interface. Again, it is shown only when the method is a property accessor or a protected method.

empty—The method has an empty body. Such methods are usually intended to be overridden in subclasses.

constant—The method’s body contains only a single `return` statement, with a simple value: a primitive or string literal or `null`. Such methods are usually intended to be overridden in subclasses.

default:—The method is a “get” or “is” property accessor with a default value, which is given following the flag. This value is determined by reflectively instantiating the class with the default constructor, if possible, and then calling the method.

=—The field is constant and has a simple value, which is given following the flag. Primitive and string literals are considered simple values, as is `null`. In addition, some simple expressions involving other primitive constant fields are evaluated to obtain a value.

A quick-reference entry ends with a *Hierarchy* section, which gives all of the supertypes for the class or interface. A class inheritance hierarchy is shown from the top down, as an arrow-separated list. If a class or interface extends or implements any interfaces, they are shown immediately following it, in a parenthesized, comma-separated list. If the hierarchy is trivial, this section is not included.

Online Examples

The Web site for this book is located at <http://www.aw.com/budinsky/>. All of the example models and code used throughout this book can be downloaded from there. The site will also provide an errata list, and other news related to the book.

To use the examples, the Eclipse SDK (Version 2.1) and EMF (Version 1.1) are required. You can find a build of the Eclipse SDK for your platform by visiting www.eclipse.org/downloads/, and EMF is available from www.eclipse.org/emf/.

Acknowledgments

Many people have helped with this book, and with the design and implementation of EMF itself. We'd like to thank Martin Nally for starting the EMF project a few years ago (before it was called EMF, or even had a name at all, for that matter), after which he gave us the mandate and support to make it happen. Equal thanks go to our colleague Steve Brodsky who, along with Martin, provided us with many hours of stimulating design discussions that, although exhausting at times, led to a much better design in the end. Steve also contributed to the implementation of EMF and provided us with material for the "EMF and Modeling Standards" section of Chapter 2.

There have been many others, both inside IBM and within the wider open source community, who have contributed ideas, bug fixes, and enhancements to EMF. We can't mention all their names, but we are nevertheless most grateful for all the input we've received. We are especially grateful to Sridhar Iyengar, one of our biggest supporters, who has been very active in helping to align EMF with related standards activities. Steve Brodsky, Martin Nally, and Barbara Price have also made significant contributions in this regard.

More to the point, we'd like to mention all of the people that helped with reviewing and preparing the book itself. Bob Schloss reviewed several chapters and provided us with many thoughtful suggestions and comments. Rich Kulp also reviewed chapters and has, from the beginning, helped with EMF by providing many suggestions and contributions to its design. Herb Derby, Jim Des Rivieres, Kenn Hussey, Marcelo Paternostro, and Hendra Suwanda also reviewed one or more chapters each, for which we are very grateful. We'd also like to thank Kelvin Chan and Kimberley Peter, graphic designers in the IBM Toronto Laboratory, who rushed to design new icons for EMF, just in time to add a touch of class to the book's screen shots.

Finally we'd like to thank the people at Addison Wesley, especially Vanessa Moore, John Neidhart, Jeff Pepper, and Debby VanDijk for helping to make it all come together.

References

- [fm03entry01] [a] D. Flanagan. *Java in a Nutshell*, 3rd Edition. O'Reilly & Associates, Sebastopol, CA, 1999.
- [fm03entry02] [1] "Eclipse Platform Technical Overview." Object Technology International, Inc., February 2003, <http://www.eclipse.org/whitepapers/eclipse-overview.pdf>.
- [fm03entry03] [2] "XML Schema Part 0: Primer." W3C, May 2001, <http://www.w3.org/TR/xmlschema-0/>.
- [fm03entry04] [3] "Downloading the JavaBeans Spec," (Section 8.3). Sun Microsystems, Inc., <http://java.sun.com/products/javabeans/docs/spec.html>.
- [fm03entry05] [4] "Document Object Model (DOM)." W3C, 2002, <http://www.w3.org/dom/>.
- [fm03entry06] [5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [fm03entry07] [6] J. Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, Reading, MA, 2001.
- [fm03entry08] [7] "JavaServer Pages." Sun Microsystems, Inc, <http://java.sun.com/products/jsp/>.
- [fm03entry09] [8] "XML Schema Part 1: Structures." W3C, May 2001, <http://www.w3.org/TR/xmlschema-1/>.

Part 1. EMF Overview

Chapter 1. Eclipse

Before we can begin to describe EMF, the Eclipse Modeling Framework, a basic introduction to Eclipse would seem in order. If you're already familiar with Eclipse, you can skip this chapter and go straight to Chapter 2, which is the "real" Chapter 1 from the EMF perspective.

Eclipse is an open source software development project, the purpose of which is to provide a highly integrated tool platform. The work in Eclipse consists of a core project, which includes a generic framework for tool integration, and a Java development environment built using it. Other projects extend the core framework to support specific kinds of tools and development environments. The projects in Eclipse are implemented in Java and run on many operating systems including Windows and Linux.

By involving committed and enthusiastic developers in an environment organized to facilitate the free exchange of technology and ideas, Eclipse is hoping to create the best possible integration platform. The software produced by Eclipse is made available under the Common Public License (CPL), which contains the usual legalese, but basically says that you can use, modify, and redistribute it for free, or include it as part of a proprietary product. The CPL is Open Source Initiative (OSI)-approved and recognized by the Free Software Foundation as a free software license. Any software contributed to Eclipse must also be licensed under the CPL.

Eclipse.org is a consortium of a number of companies that have made a commitment to provide substantial support to the Eclipse project in terms of time, expertise, technology, or knowledge. The projects within Eclipse operate under a clearly defined charter that outlines the roles and responsibilities of the various participants including the board, Eclipse users, developers, and the project management committees.

The Projects

The development work in Eclipse is divided into three main projects: the Eclipse Project, the Tools Project, and the Technology Project. The Eclipse Project contains the core components needed to develop using Eclipse. Its components are essentially fixed, and downloadable as a single unit referred to as the Eclipse SDK (Software Development Kit). The components of the other two projects are used for specific purposes and are generally independent and downloaded individually. New components are added to the Tools and Technology projects on an ongoing basis.

The Eclipse Project

The Eclipse Project supports the development of a platform, or framework, for the implementation of integrated development environments (IDEs). The Eclipse framework is implemented using Java but is used to implement development tools for other languages as well (for example, C++, XML, and so on).

The Eclipse project itself is divided into three subprojects: the Platform, the Java Development Tools (JDT), and the Plug-in Development Environment (PDE). Collectively, the three subprojects provide everything needed to extend the framework and develop tools based on Eclipse.

The Platform is the core component of Eclipse and is often considered to be Eclipse. It defines the frameworks and services required to support the plugging in and integration of tools. These services include, among others, a standard workbench user interface and mechanisms for managing projects, files, and folders. We'll describe the platform in more detail in Section 1.2.

The JDT is a full-function Java development environment built using Eclipse. Its tools are highly integrated and representative of the power of the Eclipse platform. It can be used to develop Java programs for Eclipse or other target platforms. The JDT is even used to develop the Eclipse project itself.

The PDE provides views and editors to facilitate the creation of plug-ins for Eclipse. The PDE builds on and extends the JDT by providing support for the non-Java parts of the plug-in development activity, such as registering plug-in extensions, and so on.

The Tools Project

The Eclipse Tools Project defines and coordinates the integration of different sets or categories of tools based on the Eclipse platform. The C/C++ Development Tools (CDT) subproject, for example, comprises the set of tools that define a C++ IDE. Graphical editors using the Graphical Editing Framework (GEF), and model-based editors using EMF (the subject of this book) represent categories of Eclipse tools for which common support is provided in Tools subprojects.

The Technology Project

The Eclipse Technology Project provides an opportunity for researchers, academics, and educators to become involved in the ongoing evolution of Eclipse. Of particular interest in the context of this book is the XML Schema InfoSet Model subproject, which is an EMF-based library for reading and manipulating an XML Schema definition (XSD).

The Eclipse Platform

The Eclipse Platform is a framework for building IDEs. It's been described as "an IDE for anything, and nothing in particular" [1], which is pretty much the way you could think of any framework. It simply defines the basic structure of an IDE. Specific tools extend the framework and are plugged into it to define a particular IDE collectively.

Plug-In Architecture

A basic unit of function, or a component, in Eclipse is called a *plug-in*. The Eclipse Platform itself and the tools that extend it are both composed of plug-ins. A simple tool may consist of a single plug-in, but more complex tools are typically divided into several.

From a packaging perspective, a plug-in includes everything needed to run the component, such as Java code, images, translated text, and the like. It also includes a manifest file, named *plugin.xml*, that declares the interconnections to other plug-ins. It declares, among other things, the following:

- **Requires**—its dependencies on other plug-ins.
 - **Exports**—the visibility of its public classes to other plug-ins.
 - **Extension Points**—declarations of functionality that it makes available to other plug-ins.
 - **Extensions**—its use (implementation) of other plug-ins' extension points.
-

On startup, the Eclipse Platform discovers all the available plug-ins and matches extensions with their corresponding extension points. A plug-in, however, is only activated when its code actually needs to run, avoiding a lengthy startup sequence. When activated, a plug-in is assigned its own class loader, which provides and enforces the visibility declared in the manifest files.

Workspace Resources

Integrated Eclipse tools work with ordinary files and folders, but they use a higher-level API based on resources, projects, and a workspace. A *resource* is the Eclipse representation of a file or folder that provides the following additional capabilities:

1. *Change listeners* can be registered to receive resource change notifications (called resource deltas).
2. *Markers*, such as error messages or to-do lists, can be added to a resource.
3. The previous content, or history, of a resource can be tracked.

A *project* is a special folder-type resource that maps to a user-specified folder in the underlying file system. The subfolders of the project are the same as in the physical file system, but projects are top-level folders in the user's virtual project container, called the *workspace*. Projects can also be tagged with a particular personality, called a *nature*. For example, the Java nature is used to indicate that a project contains the source code for a Java program.

The UI Framework

The Eclipse UI Framework consists of two general-purpose toolkits, SWT and JFace, and a workbench UI that defines the overall structure of an Eclipse IDE.

SWT

SWT (Standard Widget Toolkit) is an OS-independent widget set and graphics library, implemented using native widgets wherever possible. This is unlike Java's AWT, which provides a native implementation only for low-level widgets like lists, text fields, and buttons (that is, the lowest common denominator of all the OSs), and defers to Swing's emulated widgets for the rest. In SWT, emulated widgets are only used where no native implementation is possible, resulting in a portable API with as much native look-and-feel as possible.

JFace

JFace is a higher-level toolkit, implemented using SWT. It provides classes to support common UI programming tasks such as managing image and font registries, dialogs, wizards, progress monitors, and so on. The JFace API does not hide SWT, but rather works with and expands on it.

A particularly valuable part of JFace is its set of standard viewer classes. Viewer classes for lists, trees, and tables work with corresponding SWT widgets, but provide a higher-level connection to the data they're displaying. For example, they include convenient mechanisms for populating themselves from a data model and for keeping themselves in sync with changes to the model.

Another widely used part of JFace is its action framework, which is used to add commands to menus and toolbars. The framework allows you to create an action (to implement a user command), and then contribute the action to a toolbar, menu bar, or context menu. You can even reuse the same action in all three places.

Workbench

The workbench is the main window that the user sees when running Eclipse; it's sometimes referred to as the Eclipse desktop.¹ It is itself implemented using SWT and JFace. As the primary user interface, the workbench is often considered to be the Platform.

The main workbench window consists of an arrangement of *views* and *editors*. A standard Navigator view displays workspace projects, folders, and files. Other views, some standard and others that are tool specific, display information about a resource being edited or possibly the current selection in another view. For example, the standard Properties view is often used to display the properties of an object selected in the Navigator view.

An editor can be launched on a resource from the Navigator view. Eclipse editors work in the usual way, but they are integrated into the workbench window instead of launched externally in their own window. When activated, an editor will contribute its actions to the workbench's menus and toolbar.

The arrangement of views and editors in the workbench window can be customized to suit a particular task. A particular default arrangement is called a *perspective* in Eclipse. The Debug perspective, for example, decreases the size of the main source window in favor of views displaying variables, breakpoints, and other debugging information.

The primary way to extend the Eclipse platform is using extension points provided by the workbench. These extension points allow tools to add new editors, views, or perspectives to the workbench. Tools can also customize existing editors, views, or perspectives for their own purposes.

More Information

If you want to learn more about Eclipse, you can visit the Eclipse Web site at www.eclipse.org. There you will find plenty of detailed information—technical, organizational, and legal. You can download drivers, sign up to user newsgroups, or even find out how to participate in the projects. Eclipse is an open source project whose very success depends on the active participation of a vibrant user community.

¹Although primarily intended for GUI-based development, the Eclipse platform can also be used to implement non-GUI applications by running a "headless" workbench.

Chapter 2. Introducing EMF

Simply put, the Eclipse Modeling Framework (EMF) is a modeling framework for Eclipse. By now, you probably know what Eclipse is, given that you either just read Chapter 1, or you skipped it, presumably because you already knew what it was. You also probably know what a framework is, since you know what Eclipse is, and Eclipse is itself a framework. So, to understand what EMF really is, all you need to know is one more thing: What is a model? Or better yet, what do we mean by a model?

If you're familiar with things like class diagrams, collaboration diagrams, state diagrams, and so on, you're probably thinking that a model is a set of those things, probably defined using UML (Unified Modeling Language), a (the) standard notation for them. You might be imagining a higher-level description of an application from which some, or all, of the implementation can be generated. Well, you're right about what a model is, but not exactly about EMF's spin on it.

Although the idea is the same, a model in EMF is less general and not quite as high-level as the commonly accepted interpretation. EMF doesn't require a completely different methodology or any sophisticated modeling tools. All you need to get started with EMF are the Eclipse Java Development Tools. As you'll see in the following sections, EMF relates modeling concepts directly to their implementations, thereby bringing to Eclipse—and Java developers in general—the benefits of modeling with a low cost of entry.

Unifying Java, XML, and UML

To help understand what EMF is about, let's start with a simple Java programming example. Say that you've been given the job of writing a program to manage purchase orders for some store or supplier.¹ You've been told that a purchase order includes a "bill to" and "ship to" address, and a collection of (purchase) items. An item includes a product name, a quantity, and a price. "No problem," you say, and you proceed to create the following Java interfaces:

```
public interface PurchaseOrder
{
    String getShipTo();
    void setShipTo(String value);

    String getBillTo();
    void setBillTo(String value);

    List<Item> getItems(); // List of Item
}

public interface Item
{
    String getProductName();
    void setProductName(String value);

    int getQuantity();
```

¹If you've read much about XML Schema, you'll probably find this example quite familiar, since it's based on the well-known example from XML Schema Part 0: Primer [2]. We've simplified it here, but in Chapter 4 we'll step up to the real thing.

```

void setQuantity(int value);

float getPrice();
void setPrice(float value);
}

```

Starting with these interfaces, you've got what you need to begin writing the application UI, persistence, and so on.

Before you start to write the implementation code, your boss asks you, "Shouldn't you create a 'model' first?" If you're like other Java programmers we've talked to, who didn't think that modeling was relevant to them, then you'd probably claim that the Java code is the model. "Describing the model using some formal notation would have no value-add," you say. Maybe a class diagram or two to fill out the documentation a bit, but other than that it simply doesn't help. So, to appease the boss, you produce this UML diagram (see Figure 2.1)²:



Figure 2.1. UML diagram of interfaces.

Then you tell the boss to go away so you can get down to business. (As you'll see below, if you had been using EMF, you would already have avoided this unpleasant little incident with the boss.)

Next, you start to think about how to persist this "model." You decide that storing the model in an XML file would be a good solution. Priding yourself on being a bit of an XML expert, you decide to write an XML Schema to define the structure of your XML document:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.com/SimplePO"
    xmlns:PO="http://www.example.com/SimplePO">
    <xsd:complexType name="PurchaseOrder">
        <xsd:sequence>
            <xsd:element name="shipTo" type="xsd:string"/>
            <xsd:element name="billTo" type="xsd:string"/>
            <xsd:element name="items" type="PO:Item"
                minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="Item">
        <xsd:sequence>
            <xsd:element name="productName" type="xsd:string"/>
            <xsd:element name="quantity" type="xsd:int"/>
            <xsd:element name="price" type="xsd:float"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>

```

Before going any further, you notice that you now have three different representations of what appears to be pretty much (actually, exactly) the same thing: the "data model" of your application. Looking at it, you start to wonder if you could have written only one of the three (that is, Java interfaces, UML diagram, or XML Schema), and generated the others from it. Even better, you start to wonder if maybe there's even enough information in this "model" to generate the Java implementation of the interfaces.

²If you're unfamiliar with UML and are wondering what things like the little black diamond mean, Appendix A provides a brief overview of the notation.

This is where EMF comes in. EMF is a framework and code generation facility that lets you define a model in any of these forms, from which you can then generate the others and also the corresponding implementation classes. Figure 2.2 shows how EMF unifies the three important technologies: Java, XML, and UML. Regardless of which one is used to define it, an EMF model is the common high-level representation that “glues” them all together.

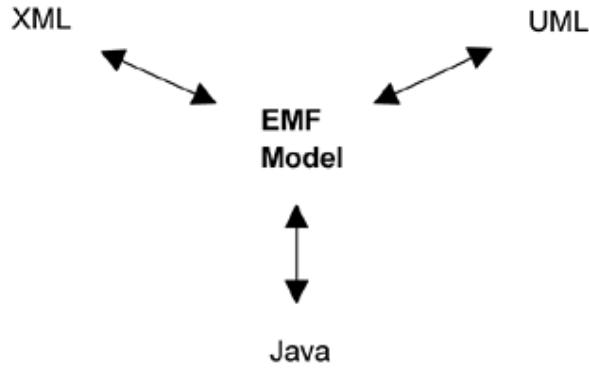


Figure 2.2. EMF unifies Java, XML, and UML.

Imagine that you want to build an application to manipulate some specific XML message structure. You would probably be starting with a message schema, wouldn't you? Wouldn't it be nice to be able to take the schema, press a button or two, and get a UML class diagram for it? Press another button, and you have a set of Java implementation classes for manipulating the XML. Finally, press one more button, and you can even generate a working editor for your messages. All this is possible with EMF, as you'll see when we walk through an example similar to this in Chapter 4.

If, on the other hand, you're not an XML Schema expert, you may choose to start with a UML diagram, or simply a set of Java interfaces representing the message structure. The EMF model can just as easily be defined using either of them. If you want, you can then have an XML Schema generated for you, in addition to the implementation code. Regardless of how the EMF model is provided, the power of the framework and generator will be the same.

Modeling vs. Programming

So is EMF simply a framework for describing a model and then generating other things from it? Well, basically yes, but there's an important difference. Unlike most tools of this type, EMF is truly integrated with and tuned for efficient programming. It answers the often-asked question, “Should I model or should I program?” with a resounding, “both.”

“To model or to program, that is not the question.”

How's that for a quote? With EMF, modeling and programming can be considered the same thing. Instead of forcing a separation of the high-level engineering/modeling work from the low-level implementation programming, it brings them together as two well-integrated parts of the same job. Often, especially with large applications, this kind of separation is still desirable, but with EMF the degree to which it is done is entirely up to you.

Why is modeling interesting in the first place? Well, for starters it gives you the ability to describe what your application is supposed to do (presumably) more easily than with code. This in turn can give you a solid, high-level way both to communicate the design and to generate part, if not all, of the implementation code. If you're a hard-core programmer without a lot of faith in the idea of high-level modeling, you should think of EMF as a gentle introduction to modeling, and the benefits it implies. You don't need to step up to a whole new methodology, but you can enjoy some of the benefits of modeling. Once you see the power of EMF and its generator, who knows, we might even make a modeler out of you yet!

If, on the other hand, you have already bought into the idea of modeling, and even the “MDA (Model Driven Architecture) Big Picture,”³ you should think of EMF as a technology that is moving in that direction, but more slowly than immediate widespread adoption. You can think of EMF as MDA on training wheels. We’re definitely riding the bike, but we don’t want to fall down and hurt ourselves by moving too fast. The problem is that high-level modeling languages need to be learned, and since we’re going to need to work with (for example, debug) generated Java code anyway, we now need to understand the mapping between them. Except for specific applications where things like state diagrams, for example, can be the most effective way to convey the behavior, in the general case, good old-fashioned Java programming is the simplest and most direct way to do the job.

From the last two paragraphs, you’ve probably surmised that EMF stands in the middle between two extreme views of modeling: the “I don’t need modeling” crowd, and the “modeling rules!” crowd. You might be thinking that being in the middle implies that EMF is a compromise and is reduced to the lowest common denominator. You’re right about EMF being in the middle and requiring a bit of compromise from those with extreme views. However, the designers of EMF truly feel that its exact position in the middle represents the right level of modeling, at this point in the evolution of software development technology. We believe that it mixes just the right amount of modeling with programming to maximize the effectiveness of both. We must admit, though, that standing in the middle and arguing out of both sides of our mouths can get tiring!

What is this right balance between modeling and programming? An EMF model is essentially the Class Diagram subset of UML. That is, a simple model of the classes, or data, of the application. From that, a surprisingly large percentage of the benefits of modeling can be had within a standard Java development environment. With EMF, there’s no need for the user, or other development tools (like a debugger, for example), to understand the mapping between a high-level modeling language and the generated Java code. The mapping between an EMF model and Java is natural and simple for Java programmers to understand. At the same time, it’s enough to support fine-grain data integration between applications; next to the productivity gain resulting from code generation, this is one of the most important benefits of modeling.

Defining the Model

Let’s step back and take a closer look at what we’re really describing with an EMF model. We’ve seen in Section 2.1 that our conceptual model could be defined in several different ways, that is, in Java, UML, or XML Schema. But what exactly are the common concepts we’re talking about when describing a model? Let’s look at our purchase order example again. Recall that our simple model included the following:

1. **PurchaseOrder** and **Item**, which in UML and Java map to class definitions, but in XML Schema map to complex type definitions.
2. **shipTo**, **billTo**, **productName**, **quantity**, and **price**, which map to attributes in UML, `get()` / `set()` method pairs (or Bean properties, if you want to look at it that way) in Java, and in the XML Schema are nested element declarations.
3. **items**, which is a UML association end or reference, a `get()` method in Java, and in XML Schema, a nested element declaration of another complex type.

³MDA is described in Section 2.6.4.

As you can see, a model is described using concepts that are at a higher level than simple classes and methods. Attributes, for example, represent pairs of methods, and as you'll see when we look deeper into the EMF implementation, they also have the ability to notify observers (such as UI views, for example) and be saved to, and loaded from, persistent storage. References are more powerful yet, because they can be bidirectional, in which case referential integrity is maintained. References can also be persisted across multiple resources (documents), where demand-load and proxy resolution come into play.

To define a model using these kinds of "model parts" we need a common terminology for describing them. More importantly, to implement the EMF tools and generator, we also need a model for the information. We need a model for describing EMF models, that is, a metamodel.

The Ecore (Meta) Model

The model used to represent models in EMF is called Ecore. Ecore is itself an EMF model, and thus is its own metamodel. You could say that makes it also a meta-metamodel. People often get confused when talking about meta-metamodels (metamodels in general, for that matter), but the concept is actually quite simple. A metamodel is simply the model of a model, and if that model is itself a metamodel, then the metamodel is in fact a meta-metamodel.⁴ Got it? If not, I wouldn't worry about it, since it's really just an academic issue anyway.

A simplified subset of the Ecore model is shown in Figure 2.3. This diagram only shows the parts of Ecore needed to describe our purchase order example, and we've taken the liberty of simplifying it a bit to avoid showing base classes. For example, in the real Ecore model the classes **EClass**, **EAttribute**, and **EReference** share a common base class, **ENamedElement**, which defines the **name** attribute which here we've shown explicitly in the classes themselves.

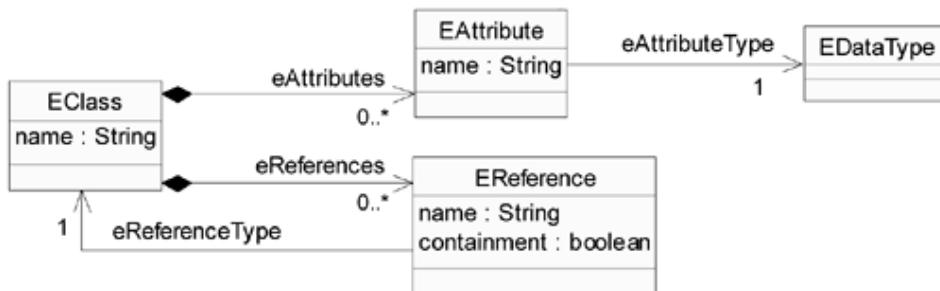


Figure 2.3. A simplified subset of the Ecore model.

As you can see, there are four Ecore classes needed to represent our model:

1. **EClass** is used to represent a modeled class. It has a name, zero or more attributes, and zero or more references.
2. **EAttribute** is used to represent a modeled attribute. Attributes have a name and a type.
3. **EReference** is used to represent one end of an association between classes. It has a name, a boolean flag to indicate if it represents containment, and a reference (target) type, which is another class.
4. **EDataType** is used to represent the type of an attribute. A data type can be a primitive type like `int` or `float` or an object type like `java.util.Date`.

⁴This concept can recurse into meta-meta-metamodels, and so on, but we won't go there.

Notice that the names of the classes correspond most closely to the UML terms. This is not surprising since UML stands for Unified Modeling Language. In fact, you might be wondering why UML isn't "the" EMF model. Why does EMF need its own model? Well, the answer is quite simply that Ecore is a small and simplified subset of full UML. Full UML supports much more ambitious modeling than the core support in EMF. UML, for example, allows you to model the behavior of an application, as well as its class structure. We'll talk more about the relationship of EMF to UML and other standards in Section 2.6.

We can now use instances of the classes defined in Ecore to describe the class structure of our application models. For example, we describe the purchase order class as an instance of **EClass** named "PurchaseOrder". It contains two attributes (instances of **EAttribute** that are accessed via **eAttributes**) named "shipTo" and "billTo", and one reference (an instance of **EReference** that is accessed via **eReferences**) named "items", for which **eReferenceType** (its target type) is equal to another **EClass** instance named "Item". These instances are shown in Figure 2.4.

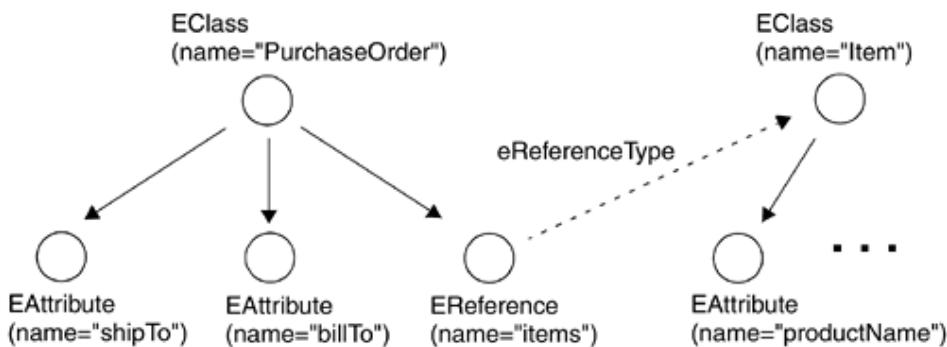


Figure 2.4. The purchase order Ecore instances.

When we instantiate the classes defined in Ecore to define the model for our own application, we are creating what we call a *core model*.

Creating and Editing the Model

Now that we have these Ecore objects to represent a model in memory, the EMF framework can read from them to, among other things, generate implementation code. You might be wondering, though, how do we create the model in the first place? The answer is that we need to build it from whatever input form you start with. If you start with Java interfaces, the EMF generator will introspect them and build the core model. If, instead, you start with an XML Schema, then the model will be built from that. If you start with UML, there are three possibilities:

1. **Direct Ecore Editing**—EMF's simple tree-based sample Ecore editor and Omndo's (free) EclipseUML graphical editor⁵ are examples.
2. **Import from UML**—The EMF Project Wizard provides this option for Rational Rose (.mdf/files) only. The reason Rose has this special status is because it's the tool that we used to "bootstrap" the implementation of EMF itself.
3. **Export from UML**—This is essentially the same as option 2, except the conversion is invoked from the UML tool, instead of from the EMF Project Wizard.

As you might imagine, option 1 is the most desirable. With it, there is no import or export step in the development process. You simply edit the model and then generate. Also, unlike the other options, you don't need to worry about the core model being out of sync with the tool's own native model. The other two approaches require an explicit reimport or reexport step whenever the UML model changes.

⁵You can download EclipseUML from the Omndo Web site at www.omndo.com.

The advantage of options 2 or 3 is that you can use the UML tool to do more than just your EMF modeling. You can use the full power of UML and whatever fancy features the particular tool has to offer. If it supports its own code generation, for example, you can use the tool to define your core model, and also to both define and generate other parts of your application. As long as the tool is able to export a serialized core model, then that tool will also be usable as an input source for the EMF framework/generator.⁶

XMI Serialization

By now you might be wondering what is the serialized form of a core model? Previously, we've observed that the "conceptual" model is represented in at least three physical places: Java code, XML Schema, or a UML diagram. Should there be just one form that we use as the primary, or standard, representation? If so, which one should it be?

Believe it or not, we actually have yet another (that is, a fourth) persistent form that we use as the canonical representation: XMI (XML Metadata Interchange). Why did we need another one? We weren't exactly short of ways to represent the model persistently.

The reason we use XMI is because it is a standard for serializing metadata, which Ecore is. Also, except for the Java code, the other forms are all optional. If we were to use Java code to represent the model, we would need to introspect the whole set of Java files every time we want to reproduce the model. It's definitely not a very concise form of the model.

So, XMI does seem like a reasonable choice for the canonical form of Ecore. It's actually the closest to choosing door number 3 (UML) anyway. The problem is that every UML tool has its own persistent model format. An Ecore XMI file is a "Standard" XML serialization of the exact metadata that EMF uses.

Serialized as XMI, our purchase order model looks something like this:

```
<?xml version="1.0" encoding="ASCII"?>
<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="http://com/example/po.ecore"
    nsPrefix="com.example.po">
  <eClassifiers xsi:type="ecore:EClass" name="PurchaseOrder">
    <eReferences name="items"
      eType="#/Item" upperBound="-1" containment="true"/>
    <eAttributes name="shipTo"
      eType="ecore:EDataType
        http://www.eclipse.org/emf/2002/Ecore//EString"/>
    <eAttributes name="billTo"
      eType="ecore:EDataType
        http://www.eclipse.org/emf/2002/Ecore//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Item">
    <eAttributes name="productName"
      eType="ecore:EDataType
        http://www.eclipse.org/emf/2002/Ecore//EString"/>
    <eAttributes name="quantity"
      eType="ecore:EDataType
        http://www.eclipse.org/emf/2002/Ecore//EInt"/>
    <eAttributes name="price"
      eType="ecore:EDataType
        http://www.eclipse.org/emf/2002/Ecore//EFloat"/>
  </eClassifiers>
</ecore:EPackage>
```

⁶For option 3, the UML tool in question needs to support exporting to "Ecore XMI," not simply export to XMI. This issue is discussed in Section 2.6.3.

Notice that the XML elements correspond directly to the Ecore instances back in Figure 2.4, which makes perfect sense seeing that this is a serialization of exactly those objects.

Java Annotations

Let's revisit the issue of defining a core model using Java interfaces. Previously we implied that when provided with ordinary Java interfaces, EMF "would" introspect them and deduce the model properties. That's not exactly the case. The truth is that given interfaces containing standard `get()` methods,⁷ EMF "could" deduce the model attributes and references. EMF does not, however, blindly assume that every interface and method in it is part of the model. The reason for this is that the EMF generator is a code-merging generator. It generates code that not only is capable of being merged with user-written code, it's *expected* to be.

Because of this, our `PurchaseOrder` interface isn't quite right for use as a model definition. First of all, the parts of the interface that correspond to model elements whose implementation should be generated need to be indicated. Unless explicitly marked with an `@model` annotation in the Javadoc comment, a method is not considered to be part of the model definition. For example, `interface PurchaseOrder` needs the following annotations:

```
/**
 * @model
 */
public interface PurchaseOrder
{
    /**
     * @model
     */
    String getShipTo();

    /**
     * @model
     */
    String getBillTo();

    /**
     * @model type="Item" containment="true"
     */
    List getItems();
}
```

Here, the `@model` tags identify `PurchaseOrder` as a modeled class, with two attributes, `shipTo` and `billTo`, and a single reference, `items`. Notice that both attributes, `shipTo` and `billTo`, have all their model information available through Java introspection, that is, they are simple attributes of type `String`. No additional model information appears after their `@model` tags, because only information that is different from the default needs to be specified.

There is some non-default model information needed for the `items` reference. Because the reference is multiplicity-many, indicated by the fact that it returns a `List`, we need to specify the target type of the reference. We also need to specify `containment="true"` to indicate that we want purchase orders to be a container for their items and serialize them as children.

Notice that the `setShipTo()` and `setBillTo()` methods are not required in the annotated interface. With the annotations present on the `get()` method, we don't need to include them; once we've identified the attributes (which are settable by default), the `set()` methods will be generated and merged into the interface if they're not already there. Whether the `set()` method is generated in the interface or not, both `get` and `set` implementation methods will be generated.

⁷EMF uses a subset of the JavaBean simple property accessor naming patterns [3].

The Ecore “Big Picture”

Let's recap what we've covered so far.

1. Ecore, and its XMI serialization, is the center of the EMF world.
2. A core model can be created from any of at least three sources: a UML model, an XML Schema, or annotated Java interfaces.
3. Java implementation code and, optionally, other forms of the model can be generated from a core model.

We haven't talked about it yet, but there is one important advantage to using XML Schema to define a model: given the schema, instances of the model can be serialized to conform to it. Not surprisingly, in addition to simply defining the model, the XML Schema approach is also specifying something about the persistent form of the model.

One question that comes to mind is whether there are other persistent model forms possible. Couldn't we, for example, provide a relational database (RDB) Schema and produce a core model from it? Couldn't this RDB Schema also be used to specify the persistent format, similar to the way XML Schema does? The answer is, quite simply, yes. This is one type of function that EMF is intended to support, and certainly not the only kind. The “big picture” is shown in Figure 2.5.

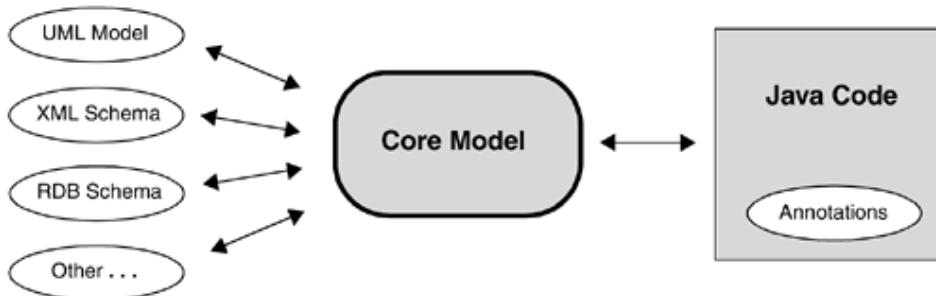


Figure 2.5. The core model and its sources.

Generating Code

The most important benefit of EMF, as with modeling in general, is the boost in productivity that results from automatic code generation. Let's say that you've defined a model, for example the purchase order core model shown in Section 2.3.3, and are ready to turn it into Java code. What do you do now? In Chapter 4, we'll walk through this scenario, and others where you start with other forms of the model (for example, Java interfaces). For now, suffice to say that it only involves a few mouse clicks. All you need to do is create a project using the EMF Project Wizard, which automatically launches the generator, and select **Generate Model Code** from a menu.

Generated Model Classes

So what kind of code does EMF generate? The first thing to notice is that an Ecore class (that is, an **EClass**) actually corresponds to two things in Java: an interface and a corresponding implementation class. For example, the **EClass** for **PurchaseOrder** maps to a Java interface:

```
public interface PurchaseOrder ...
```

and a corresponding implementation class:

```
public class PurchaseOrderImpl extends ... implements PurchaseOrder {
```

This interface/implementation separation is a design choice imposed by EMF. Why do we require this? The reason is simply that we believe it's a pattern that any good model-like API would follow. For example, DOM [4] is like this and so is much of Eclipse. It's also a necessary pattern to support multiple inheritance in Java.

The next thing to notice about each generated interface is that it extends directly or indirectly from the base interface `EObject` like this:

```
public interface PurchaseOrder extends EObject {
```

`EObject` is the EMF equivalent of `java.lang.Object`, that is, it's the base of all modeled objects. Extending from `EObject` introduces three main behaviors:

1. `eClass()` returns the object's metaobject (an `EClass`).
2. `eContainer()` and `eResource()` return the object's containing object and resource.
3. `eGet()`, `eSet()`, `eIsSet()`, and `eUnset()` provide an API for accessing the objects reflectively.

The first and third items are interesting only if you want to generically access the objects instead of, or in addition to, using the type-safe generated accessors. We'll look at how this works in Sections 2.5.3 and 2.5.4. The second item is an integral part of the persistence API that we will describe in Section 2.5.2.

Other than that, `EObject` has only a few convenience methods. However, there is one more important thing to notice; `EObject` extends from yet another interface:

```
public interface EObject extends Notifier {
```

The `Notifier` interface is also quite small, but it introduces an important characteristic to every modeled object; model change notification as in the Observer Design Pattern [5]. Like object persistence, notification is an important feature of an EMF object. We'll look at EMF notification in more detail in Section 2.5.1.

Let's move on to the generated methods. The exact pattern that is used for any given feature (that is, attribute or reference) implementation depends on the type and other user-settable properties. In general, the features are implemented as you'd expect. For example, the `get()` method for the `shipTo` attribute simply returns an instance variable like this:

```
public String getShipTo() {
    return shipTo;
}
```

The corresponding `set()` method sets the same variable, but it also sends a notification to any observers that may be interested in the state change:

```
public void setShipTo(String newShipTo) {
    String oldShipTo = shipTo;
    shipTo = newShipTo;
    if (eNotificationRequired())
        eNotify(new ENotificationImpl(this,
            Notification.SET,
            POPackage.PURCHASE_ORDER__SHIP_TO,
            oldShipTo, shipTo));
}
```

Notice that to make this method more efficient when the object has no observers, the relatively expensive call to `eNotify()` is avoided by the `eNotificationRequired()` guard.

More complicated patterns are generated for other types of features, especially bidirectional references where referential integrity is maintained. In all cases, however, the code is generally as efficient as possible, given the intended semantic. We'll cover the complete set of generator patterns in Chapter 9.

The main message you should go away with is that the generated code is clean, simple, and efficient. EMF does not pull in large base classes, or generate inefficient code. The EMF framework is lightweight, as are the objects generated for your model. The idea is that the code that's generated should look pretty much like what you would have written, had you done it by hand. But because it's generated, you know it's correct. It's a big time saver, especially for some of the more complicated reference handshaking code, which might otherwise be fairly difficult to get right.

Before moving on, we should mention two other important classes that are generated for a model: a factory and a package. The generated factory (for example, `POFactory`) includes a `create` method for each class in the model. The EMF programming model strongly encourages, but doesn't require, the use of factories for creating objects. Instead of simply using the `new` operator to create a purchase order, you should do this:

```
PurchaseOrder aPurchaseOrder =  
    POFactory.eINSTANCE.createPurchaseOrder();
```

The generated package (for example, `POPackage`) provides convenient accessors for all the Ecore metadata for the model. You may already have noticed, in the code fragment on page 23, the use of `POPackage.PURCHASE_ORDER__SHIP_TO`, a static `int` constant representing the `shipTo` attribute. The generated package also includes convenient accessors for the `EClasses`, `EAttributes`, and `EReferences`. We'll look at the use of these accessors in Section 2.5.3.

Other Generated “Stuff”

In addition to the interfaces and classes described in the previous section, the EMF generator can optionally generate the following:

1. A skeleton adapter factory⁸ class (for example, `POAdapterFactory`) for the model. This convenient base class can be used to implement adapter factories that need to create type-specific adapters. For example, a `PurchaseOrderAdapter` for `PurchaseOrders`, an `ItemAdapter` for `Items`, and so on.
2. A convenience switch class (for example, `POSwitch`) that implements a “switch statement”-like callback mechanism for dispatching based on an object's type (that is, its `EClass`). The adapter factory class, as just described, uses this switch class in its implementation.
3. A plug-in manifest file, so that the model can be used as an Eclipse plug-in.
4. An XML Schema for the model.

If all you're interested in is generating a model, then this is the end of the story. However, as we'll see in Chapters 3 and 4, the EMF generator can, using the `EMF.Edit` extensions to the base EMF framework, generate adapter classes that enable viewing and command-based, undoable editing of a model. It can even generate a working editor for your model. We will talk more about `EMF.Edit` and its capabilities in the following chapter. For now, we just stick to the basic modeling framework itself.

⁸Adapters and adapter factories are described in Section 2.5.1.

Regeneration and Merge

The EMF generator produces files that are intended to be a combination of generated pieces and hand-written pieces. You are expected to edit the generated classes to add methods and instance variables. You can always regenerate from the model as needed and your additions will be preserved during the regeneration.

EMF uses `@generated` markers in the Javadoc comments of generated interfaces, classes, methods, and fields to identify the generated parts. For example, `getShipTo()` actually looks like this:

```
/**  
 * @generated  
 */  
public String getShipTo() { ... }
```

Any method that doesn't have this `@generated` tag (that is, anything you add by hand) will be left alone during regeneration. If you already have a method in a class that conflicts with a generated method, then your version will take precedence and the generated one will be discarded. You can, however, redirect a generated method if you want to override it but still call the generated version. If, for example, you rename the `getShipTo()` method with a Gen suffix:

```
/**  
 * @generated  
 */  
public String getShipToGen() { ... }
```

Then if you add your own `getShipTo()` method without an `@generated` tag, the generator will, upon detecting the conflict, check for the corresponding Gen version and, if it finds one, redirect the generated method body there.

The merge behavior for other things is generally fairly reasonable. For example, you can add extra interfaces to the extends clause of a generated interface (or the implements clause of a generated class) and they will be retained during regeneration. The single extends class of a generated class, however, would be overwritten by the model's choice. We'll look at code merging in more detail in Chapter 9.

The Generator Model

Most of the data needed by the EMF generator is stored in the core model. As we've seen in Section 2.3.1, the classes to be generated and their names, attributes, and references are all there. There is, however, more information that needs to be provided to the generator, such as where to put the generated code and what prefix to use for the generated factory and package class names, that isn't stored in the core model. All this user-settable data also needs to be saved somewhere so that it will be available if we regenerate the model in the future.

The EMF code generator uses a generator model to store this information. Like Ecore, the generator model is itself an EMF model. Actually, a generator model provides access to all of the data needed for generation, including the Ecore part, by wrapping the corresponding core model. That is, generator model classes are Decorators [5] of Ecore classes. For example, `GenClass` decorates `EClass`, `GenFeature` decorates `EAttribute` and `EReference`, and so on.

The significance of all this is that the EMF generator runs off of a generator model instead of a core model; it's actually a generator model editor.⁹ When you use the generator, you'll be editing a generator model, which in turn indirectly accesses the core model from which you're generating. As you'll see in Chapter 4 when we walk through an example of using the generator, there are two

⁹It is, in fact, an editor generated by EMF, like the ones we'll be looking at in Chapter 4 and later in the book.

model resources (files) in the project: a `.ecore` file and a `.genmodel` file. The `.ecore` file is an XMI serialization of the core model, as we saw in Section 2.3.3. The `.genmodel` file is a serialized generator model with cross-document references to the `.ecore` file. Figure 2.6 shows the conceptual picture.

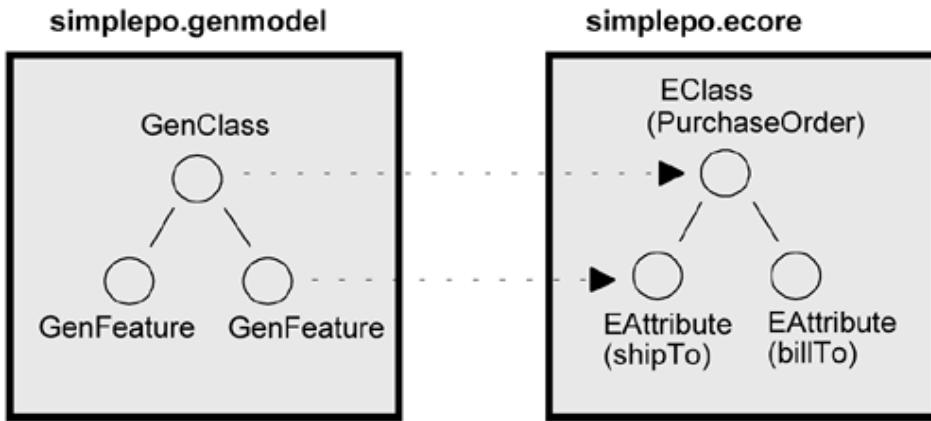


Figure 2.6. The `.genmodel` and `.ecore` files.

Separating the generator model from the core model like this has the advantage that the actual Ecore metamodel can remain pure and independent of any information that is only relevant for code generation. The disadvantage of not storing all the information right in the core model is that a generator model may get out of sync if the referenced core model changes. To handle this, the generator model classes include methods to reconcile a generator model with changes to its corresponding core model. Using these methods, the two files are kept synchronized automatically by the framework and generator. Users don't need to worry about it.

The EMF Framework

In addition to simply increasing your productivity, building your application using EMF provides several other benefits, such as model change notification, persistence support including default XMI serialization, and a most efficient reflective API for manipulating EMF objects generically. Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications.

Notification and Adapters

In Section 2.4.1, we saw that every generated EMF class is also a `Notifier`, that is, it can send notification whenever an attribute or reference is changed. This is an important property, allowing EMF objects to be observed, for example, to update views or other dependent objects.

Notification observers (or listeners) in EMF are called adapters because in addition to their observer status, they are often used to extend the behavior (that is, support additional interfaces without subclassing) of the object they're attached to. An adapter, as a simple observer, can be attached to any `EObject` (for example, `PurchaseOrder`) by adding to its `adapter` list like this:

```
Adapter poObserver = ...
aPurchaseOrder.eAdapters().add(poObserver);
```

After doing this, the `notifyChanged()` method will be called, on `poObserver`, whenever a state change occurs in the purchase order (for example, if the `setBillTo()` method is called), as shown in Figure 2.7.

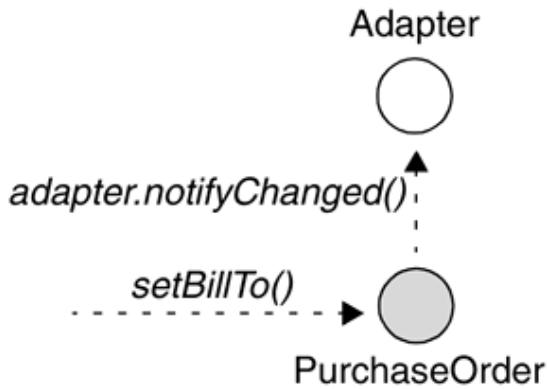


Figure 2.7. Calling the `notifyChanged()` method.

Unlike simple observers, attaching an adapter as a behavior extension is normally done using an adapter factory. An adapter factory is asked to adapt an object with an extension of the required type, something like this:

```

PurchaseOrder aPurchaseOrder = ...
AdapterFactory somePOAdapterFactory = ...
Object poExtensionType = ...
if (somePOAdapterFactory.isFactoryForType(poExtensionType)) {
    Adapter poAdapter =
        somePOAdapterFactory.adapt(aPurchaseOrder, poExtensionType);
...
}
  
```

Often, the `poExtensionType` represents some interface supported by the adapter. For example, the argument could be the actual `java.lang.Class` for an interface of the chosen adapter. The returned adapter can then be downcast to the requested interface, like this:

```

POAdapter poAdapter =
    (POAdapter)somePOAdapterFactory.adapt(someObject,
                                         POAdapter.class);
  
```

If the adapter of the requested type is already attached to the object, then `adapt()` will return the existing adapter; otherwise it will create a new one. In EMF, the adapter factory is the one responsible for creating the adapter; the EMF object itself has no notion of being able to adapt itself. This approach allows greater flexibility to implement the same behavioral extension in more than one way. If instead the object were asked to adapt itself, it could only ever return one implementation for a given extension type.

As you can see, an adapter must be attached to each individual `EObject` that it wants to observe. Sometimes, you may be interested in being informed of state changes to any object in a containment hierarchy, a resource, or even any of a set of related resources. Rather than requiring you to walk through the hierarchy and attach your observer to each object, the EMF framework provides a very convenient adapter class, `EContentAdapter`, which can be used for this purpose. It can be attached to a root object, a resource, or even a resource set, and it will automatically attach itself to all the contents. It will then receive notification of state changes to any of the objects and it will even respond to content change notifications itself, by attaching or detaching itself as appropriate.

Adapters are used extensively in EMF as observers and to extend behavior. They are the foundation for the UI and command support provided by the EMF.Edit framework, as we will see in Chapter 3. We'll also look at how they work in much more detail in Chapter 13.

Object Persistence

The ability to persist, and reference other persisted model objects, is one of the most important benefits of EMF modeling; it's the foundation for fine-grain data integration between applications. The EMF framework provides simple, yet powerful, mechanisms for managing object persistence.

As we've seen earlier, core models are serialized using XMI. Actually, EMF includes a default XMI serializer that can be used to persist objects generically from any model, not just Ecore. Even better, if your model is defined using an XML Schema, EMF allows you to persist your objects as an XML instance document conforming to that schema. The EMF framework, combined with the code generated for your model, handles all this for you.

Above and beyond the default serialization support, EMF allows you to save your objects in any persistent form you like. In this case you'll also need to write the actual serialization code yourself, but once you do that the model will transparently be able to reference (and be referenced by) objects in other models and documents, regardless of how they're persisted.

When we looked at the properties of a generated model class in Section 2.4.1, we pointed out that there are two methods related to persistence: `eContainer()` and `eResource()`. To understand how they work, let's start with the following example:

```
PurchaseOrder aPurchaseOrder =
    POFactory.eINSTANCE.createPurchaseOrder();
aPurchaseOrder.setBillTo("123 Maple Street");

Item aItem = POFactory.eINSTANCE.createItem();
aItem.setProductName("Apples");
aItem.setQuantity(12);
aItem.setPrice(0.50);

aPurchaseOrder.getItems().add(aItem);
```

Here we've created a **PurchaseOrder** and an **Item** using the generated classes from our purchase order model. We then added the **Item** to the **items** reference by calling `getItems().add()`.

Whenever an object is added to a containment reference, which **items** is, it also sets the container of the added object. So, in our example, if we were to call `aItem.eContainer()` now, it would return the purchase order, `aPurchaseOrder`.¹⁰ The purchase order itself is not in any container, so calling `eContainer()` on it would return `null`. Note also that calling the `eResource()` method on either object would also return `null` at this point.

Now, to persist this pair of objects, we need to put them into a resource. Interface **Resource** is used to represent a physical storage location (for example, a file). To persist our objects all we need to do is add the root object (that is, the purchase order) to a resource like this:

```
Resource poResource = ...
poResource.getContents().add(aPurchaseOrder);
```

After adding the purchase order to the resource, calling `eResource()` on either object will return `poResource`. The item (`aItem`) is in the resource via its container (`aPurchaseOrder`).

Now that we've put the two objects into the resource, we can save them by simply calling `save()` on the resource. That seems simple enough, but where did we get the resource from in the first place? To understand how it all fits together we need to look at another important interface in the EMF framework: **ResourceSet**.

¹⁰Notice how this implies that a containment association is implicitly bidirectional, even if, like the **items** reference, it is declared to be one-way. We'll discuss this issue in more detail in Chapter 9.

A `ResourceSet`, as its name implies, is a set of resources that are accessed together, in order to allow for potential cross-document references among them. It's also the factory for its resources. So, to complete our example, we would create the resource, add the purchase order to it, and then save it something like this¹¹:

```
ResourceSet resourceSet = new ResourceSetImpl();
URI fileURI =
    URI.createFileURI(new File("mypo.xml").getAbsolutePath());
Resource poResource = resourceSet.createResource(fileURI);
poResource.getContents().add(aPurchaseOrder);
poResource.save(null);
```

Class `ResourceSetImpl` chooses the resource implementation class using an implementation registry. Resource implementations are registered, globally or local to the resource set, based on a URI scheme, file extension, or other possible criteria. If no specific resource implementation applies for the specified URI, then EMF's default XMI resource implementation will be used.

Assuming that we haven't registered a different resource implementation, then after saving our simple resource, we'd get an XMI file, `mypo.xml`, that looks something like this:

```
<simplepo:PurchaseOrder xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:simplepo="http://simplepo.ecore"
    billTo="123 Maple Street">
    <items productName="Apples" quantity="12" price="0.5"/>
</simplepo:PurchaseOrder>
```

Now that we've been able to save our model instance, let's look at how we would load it again. Loading is also done using a resource set like this:

```
ResourceSet resourceSet = new ResourceSetImpl();
URI fileURI =
    URI.createFileURI(new File("mypo.xml").getAbsolutePath());
Resource poResource = resourceSet.getResource(fileURI, true);
PurchaseOrder aPurchaseOrder =
    (PurchaseOrder)poResource.getContents().get(0);
```

Notice that because we know that the resource has our single purchase order at its root, we simply get the first element and downcast.

The resource set also manages demand-load for cross-document references, if there are any. When loading a resource, any cross-document references that are encountered will use a proxy object instead of the actual target. These proxies will then be resolved lazily when they are first used.

In our simple example, we actually have no cross-document references; the purchase order contains the item, so they are both in the same resource. Imagine, however, that we had modeled `items` as a non-containment reference like this (Figure 2.8):



Figure 2.8. `items` as a simple reference.

Notice the missing black diamond on the `PurchaseOrder` end of the association, indicating a simple reference as opposed to a by-value aggregation (containment reference). If we make this change using Java annotations instead of UML, the `getItems()` method would need to change to this:

¹¹If you're wondering about the call to `File.getAbsolutePath()`, it's used to ensure that we start with an absolute URI that will allow any cross document references that we may serialize to use relative URIs, guaranteeing that our serialized document(s) will be location independent. URLs and cross-document referencing are described in detail in Chapter 13.

```
/**
 * @model type="Item"
 */
List getItems();
```

Now that `items` is not a containment reference, we'll need to explicitly call `getContents().add()` on a resource for the item, just like we previously did for the purchase order. Now, however, we have the option of adding it to the same resource as the purchase order, or to a different one. If we choose to put the items into separate resources, then demand loading would come into play, as shown in Figure 2.9.

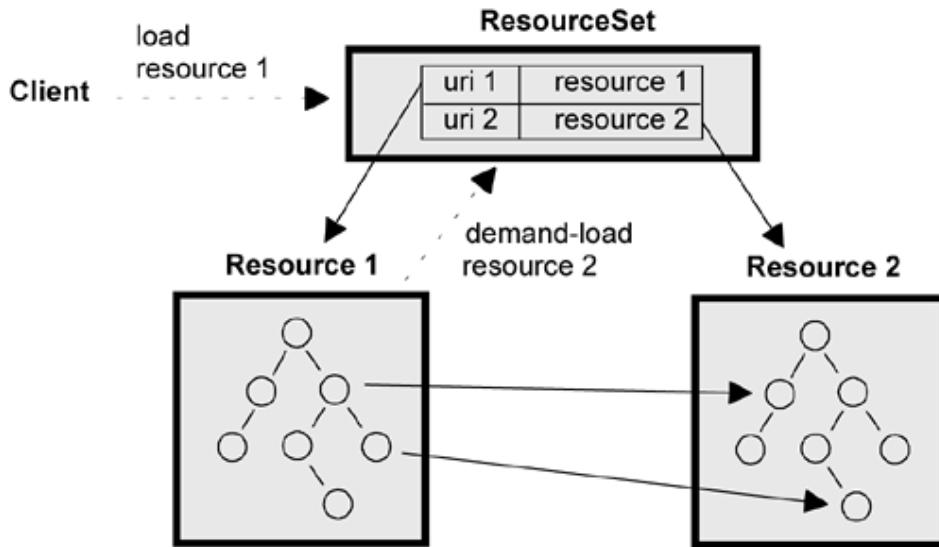


Figure 2.9. Resource set demand-loading of resources.

In the diagram, **Resource 1** (which could contain our purchase order, for example) contains cross-document references to **Resource 2** (for example, containing our item). When we load **Resource 1** by calling `getResource()` for "uri 1", any references to objects in **Resource 2** (that is, "uri 2") will simply be set to proxies. A proxy is an uninitialized instance of the target class, but with the actual object's URI stored in it. Later, when we access the object, for example by calling `aPurchaseOrder.getItems().get(0)`, **Resource 2** will be demand loaded and the proxy will be resolved (that is, replaced with the target object).

Demand loading, proxies, and proxy resolution are very important features of the EMF framework. We'll explore them in greater detail in Chapters 9 and 13.

The Reflective EObject API

As we observed in Section 2.4.1, every generated model class implements the EMF base interface, `EObject`. Among other things, `EObject` defines a generic, reflective API for manipulating instances:

```
public interface EObject
{
    Object eGet(EStructuralFeature feature);
    void eSet(EStructuralFeature feature, Object newValue);

    boolean eIsSet(EStructuralFeature feature);
    void eUnset(EStructuralFeature feature);

    ...
}
```

We can use this reflective API, instead of the generated methods, to read and write the model. For example, we can set the `shipTo` attribute of the purchase order like this:

```
aPurchaseOrder.eSet(shipToAttribute, "123 Maple Street");
```

We can read it back like this:

```
String shipTo = (String)aPurchaseOrder.eGet(shipToAttribute);
```

We can also create a purchase order reflectively, by calling a generic `create` method on the factory like this:

```
EObject aPurchaseOrder =
    poFactory.create(purchaseOrderClass);
```

If you're wondering where the metaobjects, `purchaseOrderClass` and `shipToAttribute`, and the `poFactory` come from, the answer is that you can get them using generated static accessors like this:

```
POPackage poPackage = POPackage.eINSTANCE;
POFactory poFactory = POFactory.eINSTANCE;
EClass purchaseOrderClass = poPackage.getPurchaseOrder();
EAttribute shipToAttribute =
    poPackage.getPurchaseOrder_ShipTo();
```

The EMF code generator also generates efficient implementations of the reflective methods. They are slightly less efficient than the generated `getShipTo()` and `setShipTo()` methods (the reflective methods dispatch to the generated ones through a generated switch statement), but they open up the model for completely generic access. For example, the reflective methods are used by the EMF.Edit framework to implement a full set of generic commands (for example, `AddCommand`, `RemoveCommand`, `SetCommand`) that can be used on any model. We'll talk more about this in Chapter 3.

Notice that in addition to the `eGet()` and `eSet()` methods, the reflective `EObject` API includes two more methods: `eIsSet()` and `eUnset()`. The `eIsSet()` method can be used to find out if an attribute is set or not, while `eUnset()` can be used to unset or reset it. The generic XMI serializer, for example, uses `eIsSet()` to determine which attributes need to be serialized during a resource save operation. We'll talk more about the "unset" state, and its significance on certain models, in Chapters 5 and 9.

Dynamic EMF

Until now, we've only ever considered the value of EMF in generating implementations of models. Sometimes, we would like to simply share objects without requiring generated implementation classes to be available. A simple interpretive implementation would be good enough.

A particularly interesting characteristic of the reflective API is that it can also be used to manipulate instances of dynamic, non-generated, classes. Imagine if we hadn't created the purchase order model or run the EMF generator to produce the Java implementation classes in the usual way. Instead, we simply create the core model at runtime, something like this:

```
EPackage poPackage = EcoreFactory.eINSTANCE.createEPackage();

EClass purchaseOrderClass = EcoreFactory.eINSTANCE.createEClass();
purchaseOrderClass.setName("PurchaseOrder");
poPackage.getEClassifiers().add(purchaseOrderClass);

EClass itemClass = EcoreFactory.eINSTANCE.createEClass();
itemClass.setName("Item");
```

```

poPackage.getEClassifiers().add(itemClass);

EAttribute shipToAttribute =
    EcoreFactory.eINSTANCE.createEAttribute();
shipToAttribute.setName("shipTo");
shipToAttribute.setEType(EcorePackage.eINSTANCE.getEString());
purchaseOrderClass.getEAttributes().add(shipToAttribute);

// and so on ...

```

Here we have an in-memory core model, for which we haven't generated any Java classes. We can now create a purchase order instance and initialize it using the same reflective calls as we used in the previous section:

```

EFactory poFactory = poPackage.getEFactoryInstance();
EObject aPurchaseOrder = poFactory.create(purchaseOrderClass);
aPurchaseOrder.eSet(shipToAttribute, "123 Maple Street");

```

Because there is no generated `PurchaseOrderImpl` class, the factory will create an instance of `EObjectImpl` instead. `EObjectImpl` provides a default dynamic implementation of the reflective API. As you'd expect, this implementation is slower than the generated one, but the behavior is exactly the same.

An even more interesting scenario involves a mixture of generated and dynamic classes. For example, assume that we had generated class `PurchaseOrder` in the usual way and now we'd like to create a dynamic subclass of it.

```

EClass subPOClass = EcoreFactory.eINSTANCE.createEClass();
subPOClass.setName("SubPO");
subPOClass.getESuperTypes().add(POPackage.getPurchaseOrder());
poPackage.getEClassifiers().add(subPOClass);

```

If we now instantiate an instance of our dynamic class `SubPO`, then the factory will detect the generated base class and will instantiate it instead of `EObjectImpl`. The significance of this is that any accesses we make to attributes or references that come from the base class will call the efficient generated implementations in class `PurchaseOrderImpl`:

```
String shipTo = aSubPO.eGet(shipToAttribute);
```

Only features that come from the derived (dynamic) class will use the slower dynamic implementation.

The most important point of all this is that, when using the reflective API, the presence (or lack thereof) of generated implementation classes is completely transparent. All you need is the core model in memory. If generated implementation classes are (later) added to the class path, they will then be used. From the client's perspective, the only thing that will change will be the speed of the code.

Foundation for Data Integration

The last few sections have shown various features of the EMF framework that support sharing of data. Section 2.5.1 described how change notification is an intrinsic property of every EMF object, and how adapters can be used to support open-ended extension. In Section 2.5.2, we showed how the EMF persistence framework uses `Resources` and `ResourceSets` to support cross-document referencing, demand-loading of documents, and arbitrary persistent forms. Finally, in Sections 2.5.3 and 2.5.4 we saw how EMF supports generic access to EMF models, including ones that may be partially or completely dynamic (that is, without generated implementation classes).

In addition to these features, the EMF framework provides a number of convenience classes and utility functions to help manage the sharing of objects. For example, a utility class for finding object cross-references (`EcoreUtil.CrossReferencer` and its subclasses) can be used to find any uses of an object (for example, to cleanup references when deleting the object) and any unresolved proxies in a resource, among other things.

All these features, combined with an intrinsic property of modeling—that it provides a higher-level description that can more easily be shared—provide all the needed ingredients to foster fine-grain data integration. While Eclipse itself provides a wonderful platform for integration at the UI and file level, EMF builds on this capability to enable applications to integrate at a much finer granularity than would otherwise be possible. We've seen how EMF can even be used to share data reflectively, even without using the EMF code generation support. Whether dynamic or generated, EMF models are the foundation for fine-grain data integration in Eclipse.

EMF and Modeling Standards

EMF is often discussed together with several important modeling standards of the Object Management Group (OMG), including UML, MOF, XMI, and MDA. This section introduces these standards and describes EMF's relationships with them.

UML

Unified Modeling Language is the most widely used standard for describing systems in terms of object concepts. UML is very popular in the specification and design of software, most often software to be written using an object-oriented language. UML emphasizes the idea that complex systems are best described through a number of different views, as no single view can capture all aspects of such a system completely. As such, it includes several different types of model diagrams to capture usage scenarios, class structures, behaviors, and implementations.

EMF is concerned with only one aspect of UML, class modeling. This focus is in no way a rejection of UML's holistic approach. Rather, it is a starting point, based on the pragmatic realization that the task of translating the ideas that can be expressed in various UML diagrams into concrete implementations is very large and very complex.

UML was standardized by the OMG in 1997. The standard's latest version is 1.5; it is available at www.omg.org/technology/documents/formal/uml.htm.

MOF

Meta-Object Facility (MOF) concretely defines a subset of UML for describing class modeling concepts within an object repository. As such, MOF is comparable to Ecore. However, with a focus on tool integration, rather than metadata repository management, Ecore avoids some of MOF's complexities, resulting in a widely applicable, optimized implementation.

MOF and Ecore have many similarities in their ability to specify classes and their structural and behavioral features, inheritance, packages, and reflection. They differ in the area of life cycle, data type structures, package relationships, and complex aspects of associations.

MOF was standardized in 1997, at the same time as UML. The standard, also at version 1.4, is available at www.omg.org/technology/documents/formal/mof.htm.

The next versions of MOF and UML—2.0—are currently works-in-progress. Requests for proposals have been issued, with a goal of unifying the specification cores, increasing the scope, and addressing existing deficiencies. The experience of EMF has substantially influenced the current status of the working drafts, in terms the layering of the architecture and the structure of the semantic core.

XMI

XML Metadata Interchange is the standard that connects modeling with XML, defining a simple way to specify model objects in XML documents. An XMI document's structure closely matches that of the corresponding model, with the same names and an element hierarchy that follows the model's containment hierarchy. As a result, the relationship between a model and its XMI serialization is easy to understand.

Although XMI can be, and is by default, used as the serialization format for any EMF model, it is most appropriate for serializing models representing metadata—that is, metamodels, like Ecore itself. We refer to a core model, serialized in XMI 2.0, as “Ecore XMI” and consider an Ecore XMI (*.ecore*) file as the canonical form of such a model.

XMI was standardized in 1998, shortly after XML 1.0 was finalized. The XMI 2.0 specification is available at www.omg.org/technology/documents/formal/xmi.htm.

MDA

Model Driven Architecture is an industry architecture proposed by the OMG that addresses full life-cycle application development, data, and application integration standards that work with multiple middleware languages and interchange formats. MDA unifies some of the industry best practices in software architecture, modeling, metadata management, and software transformation technologies that allow a user to develop a modeling specification once and target multiple technology implementations by using precise transformations/mappings.

EMF supports the key MDA concept of using models as input to development and integration tools: in EMF, a model is used to drive code generation and serialization for data interchange.

MDA information and key specifications are available at www.omg.org/mda.

Chapter 3. Model Editing with EMF.Edit

In the previous chapter we saw how EMF can take a model definition and produce a good, easily customizable Java implementation for it. Well, that's just the beginning. Once you decide to use EMF to model your application, you can then use the EMF.Edit framework to build very functional viewers and editors for the model. You can generate an editor that will display and edit (that is, copy, paste, drag-and-drop, and so on) instances of your model using standard JFace viewers and a property sheet, all with unlimited undo/redo. Alternatively, you can use the reflective support in EMF.Edit to do the same kinds of editing reflectively, even with a dynamic EMF model that you didn't generate.

You may be thinking that this is beginning to sound like one of those infomercials: If you buy EMF in the next 24 hours, we'll throw in the free viewers, drag-and-drop, and a bonus icon directory. Well, maybe it does, but the bottom line is that EMF.Edit is simply exploiting the information that is available in the model, along with the mechanisms provided in the EMF core, to provide more and higher-level function. The free function being offered comes naturally from the fact that we have a model, so you can rest assured that there is no "free juicer" about to be offered.

Displaying and Editing EMF Models

Let's return to the simple purchase order model we looked at in Chapter 2. Recall that it consisted of two simple classes with a containment association between them (Figure 3.1):

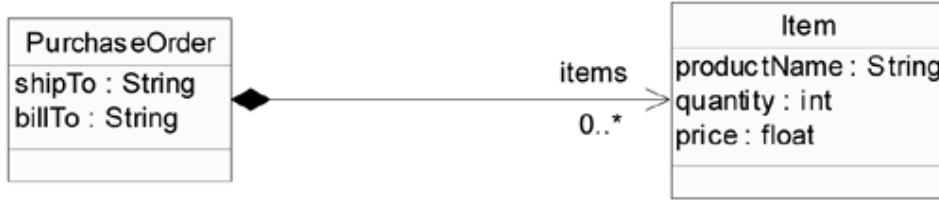


Figure 3.1. Purchase order model with containment association.

In the previous chapter we saw how the base framework of EMF gave you the ability to generate Java implementations for the model classes `PurchaseOrder` and `Item`, as well as other supporting classes. We also saw how using the generated classes along with the framework classes, `Resource` and `ResourceSet`, you could persist a purchase order instance, and its items. For example, assuming you used the framework default (XMI) serializer, a purchase order with two items could be serialized something like this:

```

<simplepo:PurchaseOrder xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:simplepo="http://simplepo.ecore"
    billTo="123 Maple Street">
    <items productName="Apples" quantity="12" price="0.5"/>
    <items productName="Oranges" quantity="24" price="0.3"/>
</simplepo:PurchaseOrder>
  
```

Let's assume you save this into a file, *My.po*, somewhere in your Eclipse workspace. The next thing you might like to do is display and edit it using a purchase order editor launched in the Eclipse workbench. You could then, for example, display the containment structure in a tree view and edit the attributes in a property sheet as shown in Figure 3.2. To really leverage the power of Eclipse, you would want to "integrate" the purchase order implementation with the Eclipse desktop this way.

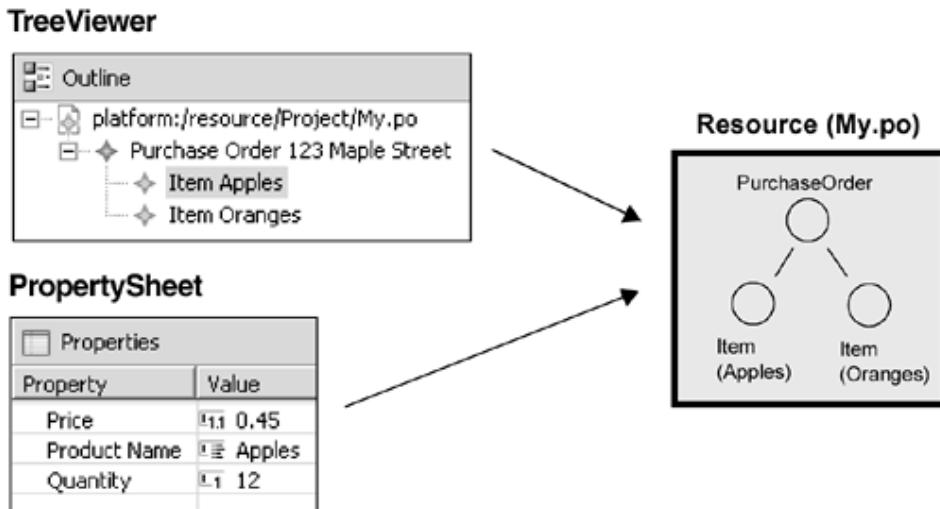


Figure 3.2. Outline and Properties views of a purchase order.

So, what would it take to display a model in a UI like this? To understand that, we first need to understand how Eclipse viewers work in general. The following section gives a brief overview of the Eclipse UI framework's viewer classes, property sheet, and action mechanism. If you already know how they work, you might want to skip it and go straight to Section 3.1.2 where we start to look at what EMF.Edit provides, to help you use the Eclipse framework to display and edit EMF models.

Eclipse UI Basics

Included in JFace, a part of the Eclipse user interface framework, is a set of reusable viewer classes (for example, `TreeViewer`) for displaying structured models. Instead of querying the model objects directly, the JFace viewers use a content provider to navigate the content and a label provider to retrieve the label text and icons for the objects being displayed. Each viewer class uses a content provider that implements a specific provider interface. For example, a `TreeViewer` uses a content provider that implements the interface `ITreeContentProvider`, as shown in Figure 3.3.

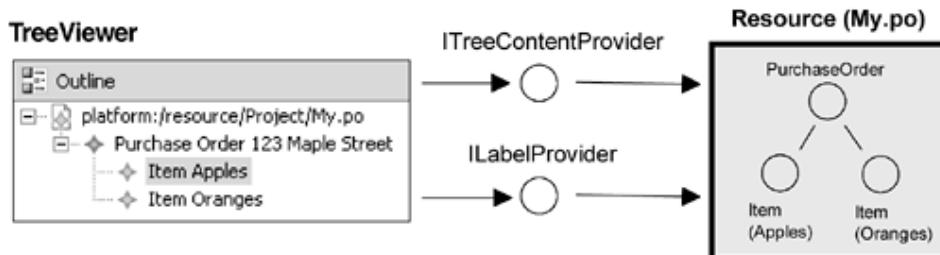


Figure 3.3. JFace viewer access to a model instance.

To display our purchase order resource in a `TreeViewer`, we start by providing the root object (the Resource object, “My.po”, in this example) to the viewer. The viewer will respond by calling the `getText()` and `getImage()` methods on its label provider to retrieve the image and text, respectively. Next, the `TreeViewer` will call the `getChildren()` method on its content provider to retrieve the next level of objects to display in the tree (only the single purchase order in our example). This process of retrieving the text, icon, and children will then repeat for the rest of the tree.

In addition to class `TreeViewer`, JFace also includes `TableViewer` and `ListViewer` classes, which work the same way. They use a different content provider interface, `IStructuredContentProvider`, to retrieve their content. However, the `ITreeContentProvider` interface actually extends from `IStructuredContentProvider`, so any tree content provider implementation class can also conveniently be used to support the other viewers as well.

Now that we understand how viewers are populated from a model, let's look at how a property sheet, which populates the Properties view, works. A property sheet gets the properties of an object by first calling the `getPropertySource()` method on its associated `IPropertySourceProvider`. Figure 3.4 shows how the property source provider produces an `IPropertySource` corresponding to a specific model object, for example, the “Apples” Item. Next, the property sheet calls the `getPropertyDescriptors()` method on the property source to get a list of `IPropertyDescriptor`s for the object's properties (“Price”, “Product Name”, and “Quantity”). The `IPropertyDescriptor` interface is then used by the property sheet to display and edit the properties.

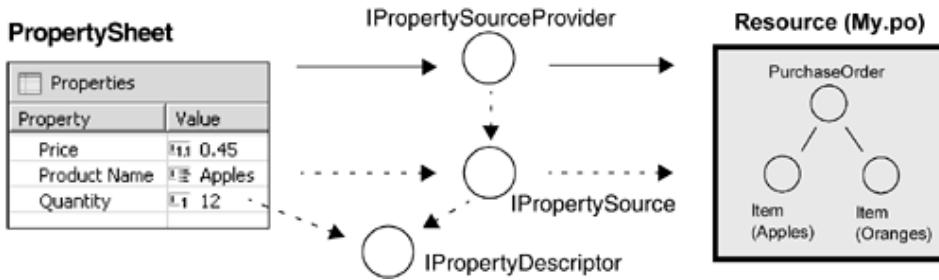


Figure 3.4. Eclipse property sheet.

There is one more important part of the Eclipse UI framework that we should look at: the action mechanism. Actions implement the `IAction` interface and represent the commands that can be run from menu items or toolbar buttons. When a menu item or toolbar button is selected, the framework invokes the associated action by calling its `run()` method. Actions also include methods to retrieve, among other things, the label and icon. For example, the `getText()` method is called by the framework to get the text for a menu item, when it is showing. The `getImageDescriptor()` method is used to get the icon to display both on menu items and toolbar buttons.

An action bar contributor is used to create and manage the actions for an editor. A subclass of `EditorActionBarContributor` is used to contribute the actions on behalf of its associated `EditorPart`. For example, to add a **New Purchase Order** item to the menu bar for a purchase order editor, `POEditor`, we would create an action bar contributor subclass, `POActionBarContributor` for example, and override the `contributeToMenu()` method to add (contribute) the new action. `POActionBarContributor` would be associated with `POEditor` in the workbench “registered editor” extension in the editor plug-in's manifest file.

EMF.Edit Support

Now that we know how views and actions work in Eclipse, we can look at how EMF.Edit helps you implement a UI for a model based on EMF.

To implement a tree viewer, like the one shown in the previous section, we need an implementation of the `ITreeContentProvider` interface that is capable of returning the children of EMF model objects. We also need an implementation of `ILabelProvider` to return a suitable text string for the label, usually based on one of the Ecore attributes. For a property sheet, we need a way of producing a set of `IPropertyDescriptors` for the (subset of) Ecore attributes and references that should be properties. EMF.Edit supports two ways of implementing these things, one using the reflective `EObject` API, and the other using generated classes.

The reflective approach consults the core model at runtime to provide a “best guess” implementation. For example, it implements the `getChildren()` method by returning the contained objects, `eContents()`, of the parent. For a label, it first tries to find a “name” attribute (case insensitive) in the class or failing that, one that includes the string “name” (for example, “productName”). For our simple purchase order example, the reflective implementation will produce pretty much what we want.

The second approach is to use the EMF.Edit generator to generate an implementation. The generated approach will, by default, produce the same behavior as the reflective implementation. This is not surprising, since the code is generated from the same core model that the reflective implementation uses at runtime. With the generated approach, however, you have an opportunity to influence some of the choices before the code is generated. For example, you could pick the “quantity” attribute for the label feature instead of default “productName”.

Generating the implementation classes also results in a much more easily customizable solution. Just like the EMF model classes described in Chapter 2, you can modify and regenerate the EMF.Edit classes any way you like. The generated classes provide convenient override points for the most common types of customizations, unlike the reflective implementation, which would require a monolithic override with lots of `instanceof` checks.

EMF.Edit also includes support for Eclipse actions and model modification in general. Changing the state of an EMF model object, by running an action or using a property descriptor, is implemented (in EMF.Edit) by delegating to a command [5]. EMF.Edit includes generic implementations of a number of common commands, as well as framework support for customizing their behavior or for implementing your own specialized commands.

As shown in Figure 3.5, EMF.Edit is the bridge between the Eclipse UI framework and the EMF core framework.



Figure 3.5. EMF.Edit connects the Eclipse UI and EMF core frameworks.

A large amount of EMF.Edit function is actually independent of the UI. To support reuse of the UI-independent parts, the EMF.Edit framework is divided into two separate plug-ins:

1. `org.eclipse.emf.edit` is the low-level UI-independent portion.
2. `org.eclipse.emf.edit.ui` contains the Eclipse UI-dependent implementation classes.

Most of the real editing work is done in the UI-independent plug-in. The UI part handles the actual connection to the display, with an implementation tied to the Eclipse UI framework. As we'll see in the following two sections, the bulk of the work is actually done by delegating to two very important UI-independent mechanisms: item providers and commands.

Item Providers

Item providers are the single most important objects in EMF.Edit; they are used to adapt model objects so the model object can provide all of the interfaces that it needs to be viewed or edited. If you think back to Chapter 2, where we saw how EMF adapters can be used as both behavioral extensions and as change observers, you can see how adapters would be just right for implementing item providers. As behavioral extensions, they can adapt the model objects to implement whatever interfaces the editors and views need, and at the same time, as observers, they will be notified of state changes which they can then pass on to listening views.

Although item providers are usually EMF adapters, they are not always. An item provider that is “providing” for an EMF object will be an adapter, but other item providers may represent non-modeled objects, mixed in to a view with modeled items. This is an important feature of the EMF.Edit framework. It has been carefully designed to allow you to create views on EMF models that may be structurally different from the model itself (that is, views that suppress objects or include additional, non-modeled, objects). We'll look at this issue in Chapter 14, but for now you should simply think of item providers as adapters on EMF objects, but keep in mind that the framework is actually more flexible.

Their name, item provider, stems from the fact that they “provide” functions on behalf of individual editable model “items” (objects). As you'll see in the following sections, the EMF.Edit framework implements a delegation scheme whereby most functions involving model objects are ultimately implemented in their associated item provider. Consequently, item providers need to perform four major roles:

1. Implement content and label provider functions
2. Provide a property source (property descriptors) for EMF objects
3. Act as a command factory for commands on their associated model objects
4. Forward EMF model change notifications on to viewers

A given item provider can implement all of these functions or just a subset, depending on what editing functions are actually required. Most commonly, however, item providers will simply implement them all by subclassing the very functional EMF.Edit base class, `ItemProviderAdapter`. It implements most of the function generically, so a subclass (which may be generated, as we'll see in Section 3.4.1) only needs to implement a few methods to complete the job. EMF.Edit also provides a full function subclass, `ReflectiveItemProvider`, that implements all of the roles using the reflective EObject API. We'll talk about these and other implementation issues in Section 3.2.5, but first, the next four sections will describe each of the roles of an item provider.

Content and Label Item Providers

The first role of an item provider is to support the implementation of content and label providers for the viewers. In Section 3.1.1, we saw how Eclipse viewers use a content provider (for example, `ITreeContentProvider`) and a label provider (for example, `ILabelProvider`) to get the information they need from the model. EMF.Edit provides generic content and label provider implementation classes, `AdapterFactoryContentProvider` and `AdapterFactoryLabelProvider`, that delegate their implementation to item providers as shown in Figure 3.6. Both of these classes are constructed with an adapter factory (`POIItemProviderAdapterFactory` in this example), which, like any other EMF adapter factory, serves to create and locate EMF adapters of a specific type (item providers for the purchase order model, in this case).

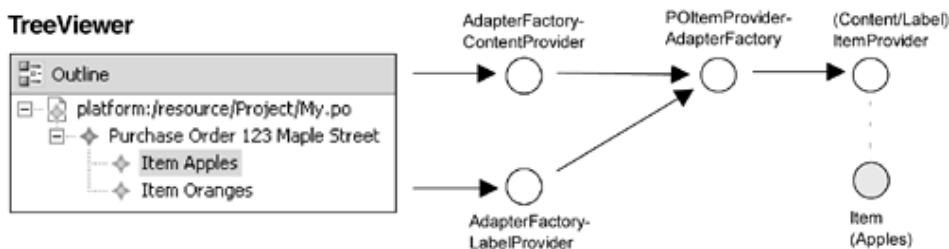


Figure 3.6. Content and label provider role of an item provider.

To service a request like `ITreeContentProvider.getChildren()`, for example, the `AdapterFactoryContentProvider` first calls `adapt()` on the `ItemProviderAdapterFactory`, which will create or return the `ItemProvider` (adapter) for the specified object. It then simply delegates to the `getChildren()` method of a corresponding item provider interface, `ITreeItemContentProvider`. The `getChildren()` method in `AdapterFactoryContentProvider` looks something like this:

```

public Object[] getChildren(Object object) {
    ITreeItemContentProvider adapter = (ITreeItemContentProvider)
        adapterFactory.adapt(object, ITreeItemContentProvider.class);
    return adapter.getChildren(object).toArray();
}
  
```

This same pattern is used for all of the content provider methods, and also by the `AdapterFactoryLabelProvider` to implement `ILabelProvider` methods (like `getText()`, for example). The adapter factory content and label providers do nothing more than simply delegate JFace provider methods to corresponding EMF content and label item provider mixin interfaces; there are four of them in total:

1. `ITreeItemContentProvider` is used to support content providers for `TreeViewers`.
2. `IStructuredItemContentProvider` is used to support content providers for other structured viewers, such as `ListViewers` and `TableViewers`.
3. `ITableItemLabelProvider` is used to support label providers for `TableViewers`.
4. `IIItemLabelProvider` is used to support label providers for other structured viewers.

Notice the similarity of these interface names to those in JFace, only with the word “Item” added. The EMF.Edit interfaces are in fact very similar to the corresponding JFace ones. The main reason for having the parallel set of interfaces is to avoid any dependencies on JFace. Although item providers are primarily used to implement Eclipse (JFace-based) UIs, they are completely UI independent. In addition to their use in support of the JFace implementation classes, `AdapterFactoryContentProvider` and `AdapterFactoryLabelProvider`, they can also be used to implement views for other UI libraries (for example, Swing), or to implement non-UI, command-based utilities for EMF models.

Item Property Source

The second major role of an item provider is to act as a property source for the property sheet. Recall that the Eclipse `PropertySheet` uses an `IPropertySourceProvider` to request an `IPropertySource` for the object whose properties it wants to display and edit. In EMF.Edit, the `AdapterFactoryContentProvider` also implements the `IPropertySourceProvider` interface and is used to provide a property source to the property sheet, as shown in Figure 3.7.

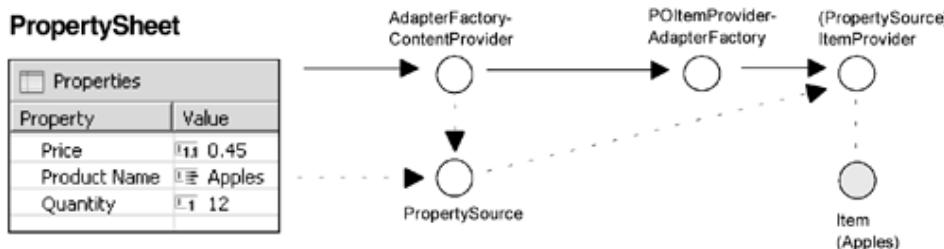


Figure 3.7. Property source role of an item provider.

Following the same pattern that we saw for `ITreeContentProvider` in the previous section, the `AdapterFactoryContentProvider` uses the adapter factory to locate an item provider, only this time one that implements the EMF.Edit item provider mixin interface `IItemPropertySource`. Again, just like the content and label provider interface, this EMF.Edit interface, `IItemPropertySource`, is very similar to its corresponding Eclipse interface, `IPropertySource`, only UI independent. Another EMF.Edit helper class, `PropertySource`, implements the actual `IPropertySource` interface needed by the property sheet. The `AdapterFactoryContentProvider` creates an instance of this class as a wrapper for the selected item provider (that is, the `IItemPropertySource`) and then returns it to the `PropertySheet`.

This same wrapping pattern is used again when the property sheet calls the `IPropertySource.getPropertyDescriptors()` method. Class `PropertySource` services the request by delegating to the `getPropertyDescriptors()` method on the item provider, which returns a set of `IItemPropertyDescriptor`s. `PropertySource` then constructs an `IPropertyDescriptor` wrapper class, `PropertyDescriptor`, for the item property descriptor and then returns it to the property sheet. The complete picture is shown in Figure 3.8.

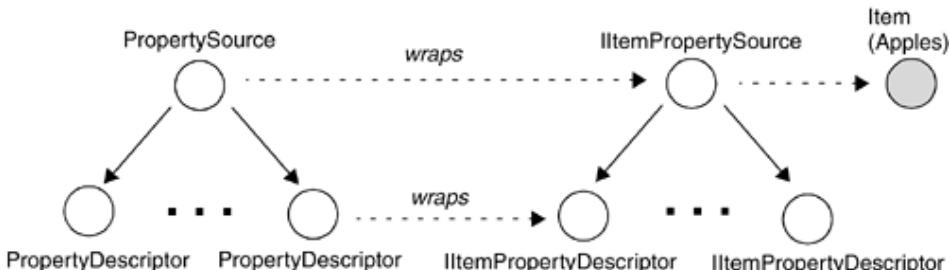


Figure 3.8. PropertySource and PropertyDescriptor delegation.

With this arrangement, property descriptor calls are now delegated to their “Item” equivalents. For example, if a property value is changed in the property sheet, the `PropertySource.setPropertyValue()` method will be called. The property source will then simply delegate to the `setPropertyValue()` method on the `ItemPropertyDescriptor`, which will make the actual model change.

If you’re thinking that all this creating of wrapper objects seems fairly messy, you’re right, but that’s the price we have to pay to keep the item providers UI independent. The good news is that the property descriptors are in fact the worst, and the last, of this double-object pattern.

Command Factory

Item providers act as the factory for commands involving their adapted objects. In this role, item providers play a critical part in the EMF.Edit command framework, which we'll look at in Section 3.3. For now, let's just say that EMF.Edit provides all the mechanisms for modifying EMF objects in an undoable way, including a full set of generic commands. The framework makes it easy to tune the command behavior for specific models by delegating their creation to item providers.

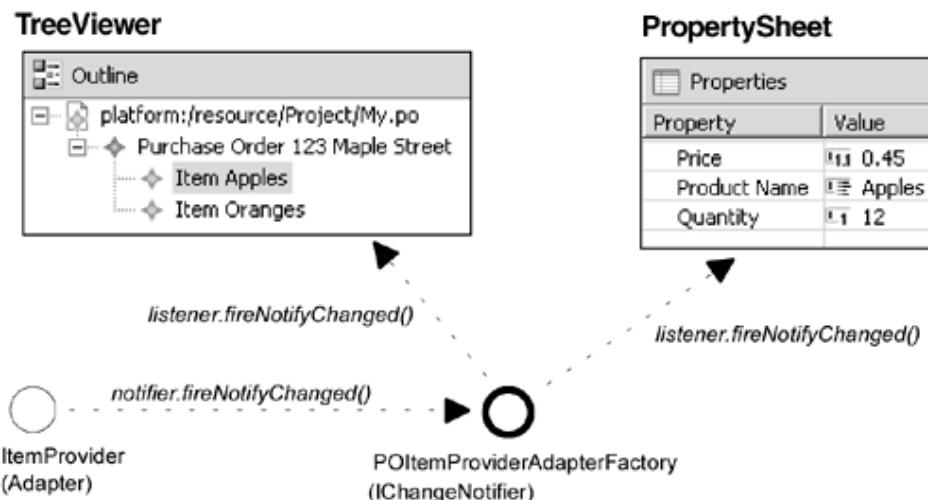
Similar to the way the eclipse UI framework uses “provider” interfaces (for example, `ITreeContentProvider`) to access the model, the EMF.Edit command framework also has an interface, `EditingDomain`, which it uses for the same purpose. Also, just like the content and label providers, a delegating implementation class, `AdapterFactoryEditingDomain`, is used to implement it. As shown in Figure 3.9, `AdapterFactoryEditingDomain` works the same as the other `AdapterFactory` implementation classes, only it delegates its methods to an item provider supporting the editing domain item provider mixin interface `IEditionDomainItemProvider`. We'll look at editing domains and the role of item providers in their implementation in Section 3.3.3.



Figure 3.9. Command creation role of an item provider.

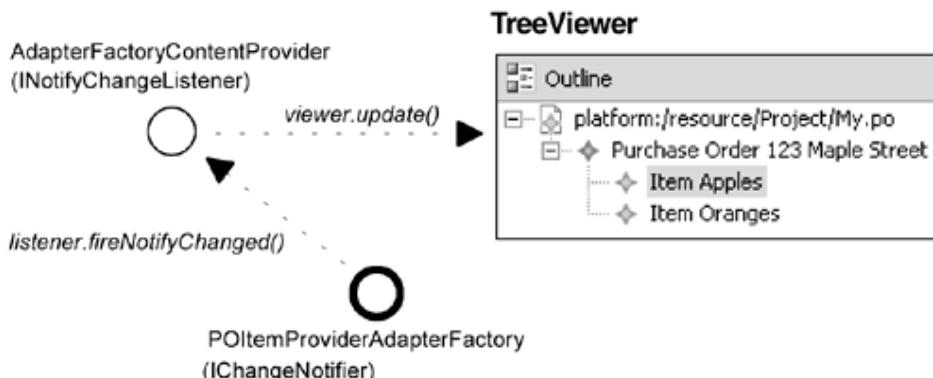
Change Notification

Being a standard EMF adapter, an `ItemProvider` will be notified, with a call to its `notifyChanged()` method, whenever an object that it is adapting changes state. The `ItemProvider`'s responsibility as an observer is to optionally filter uninteresting events and then to pass the remaining ones on to the central change notifier for the model, usually the `ItemProviderAdapterFactory`. The `ItemProviderAdapterFactory` implements the EMF.Edit interface `IChangeNotifier`, which allows views and other interested parties to register as listeners of the model as a whole. The design is illustrated in Figure 3.10.

Figure 3.10. Notification flow from an `ItemProvider` to the views.

The diagram shows the flow of notification after a purchase order `Item` changes state in some way (for example, the `productName` attribute was just set to "Apples"). In its `notifyChanged()` method, the `ItemItemProvider` (that is the `ItemProvider` for the model class `Item`) passes on the change notification by calling the `fireNotifyChanged()` method on the `IChangeNotifier` (the `POItemProviderAdapterFactory` in our example), which in turn calls `fireNotifyChanged()` on all its registered listeners. In our example, the listeners are a `TreeViewer` and the `PropertySheet`, which would now be updated to reflect the model change.

That's all there is to it, but there is one more thing worth mentioning. When notifying a JFace viewer, as in this example, the viewer itself is not the actual listener. Instead, the content provider associated with the viewer registers on behalf of its viewer. As shown in Figure 3.11, it responds to the `fireNotifyChanged()` call by calling the appropriate update method(s) on the viewer. This is another function that is handled automatically for you by the EMF.Edit class `AdapterFactoryContentProvider`. It knows how to most efficiently update all of the standard JFace viewers.

Figure 3.11. `AdapterFactoryContentProvider` updating its associated viewer.

Item Provider Implementation Classes

Now that we understand the various roles that item providers play, the question of how to implement them comes up. The EMF.Edit framework provides lots of flexibility as far as this is concerned, ranging from using a single generic (reflective) item provider to having generated type-specific item providers for every class in the model.

Reflective Item Provider

In Chapter 2, we showed how Java code does not necessarily need to be generated for an EMF model. If you choose to not generate implementation classes for your model, you can still create and manipulate instances of your classes using dynamic EMF and the reflective `EObject` API. The same is true for the editing support in EMF.Edit.

You can decide to generate or to use a reflective implementation for your item providers. In fact, you can choose to use reflective item providers for some of your classes, and generated ones for others. The important thing to consider is how much customization you plan to do.

`Class ReflectiveItemProvider` is the EMF.Edit generic item provider implementation. As you might imagine, it implements essentially the same behavior as generated item providers with default generator settings. Using it is simple, but not easily customizable. Any customization you might want to do will involve `instanceof` checks. If you use typed item provider classes whose inheritance hierarchy mirrors the model's instead, you will be able to specialize the implementation in a clean, object-oriented way.

Typed Item Providers

When using the typed item providers, there are two possible patterns:

1. **The Stateful pattern**, where every object in a model (instance) has a one-to-one correspondence with its item provider. Each item provider has a target pointer to the one-and-only object that it is adapting. This pattern doubles the number of objects in the application, and therefore only makes sense in applications where the instances are needed to carry additional state. That's why we call it the Stateful pattern.
2. **The Singleton pattern** avoids most of the extra objects. With this pattern, we share a single item provider adapter for all objects of the same type. In this case, like the reflective case, each item provider has many targets.

To allow more than one object to share an item provider (that is, for the Singleton or reflective cases), an extra argument (the object) is passed to every item provider interface method. For example, the `getChildren()` method in the `ITreeItemProvider` interface looks like this:

```
public Collection getChildren(Object object);
```

In the Stateful case, this object will always be the same as the adapter's target, so a Stateful item provider implementation could choose to ignore this argument and use the adapter target to access the model object instead. However, doing the opposite (using the argument instead of the target) is a better approach since it leaves you the option of making the item provider a Singleton in the future, without having to change it.

Command Framework

In the previous sections, we've seen how EMF models can be viewed using content and label providers, but we haven't talked much about how to change, or edit, a model. Another very important feature of the EMF.Edit framework is its support for command-based editing of an EMF model, including fully automatic undo and redo.

The command framework in EMF is divided into two parts, the common command framework and the EMF.Edit commands. The common framework defines basic command interfaces and provides some implementation classes like a basic command stack, a compound command for composing commands from other commands, and other convenient command implementations. The commands in the common framework are very general and can be used independently of EMF.Edit. In fact, they don't even depend on EMF modeled objects (that is, `EObjects`); they're completely independent of EMF.

The EMF.Edit Commands, on the other hand, are implementation classes specifically for editing `EObjects`. EMF.Edit includes a full set of generic command implementation classes that, using the reflective `EObject` API, provide support for setting attributes, adding and removing references, copying objects, and other kinds of modifications to a model. In the next two sections we'll look at some of the most interesting commands provided by the framework. Then, in Section 3.3.3, we'll look at the rest of the infrastructure in the command framework, and how it allows you to easily customize a command's behavior for your specific model.

Common Command Framework

The common command framework includes the basic interfaces and implementation classes with which model-change commands can be built and executed. Although the main use of them is in EMF.Edit, they are completely general purpose (that is, they work with `java.lang.Object`s as opposed to `EObjects`) and therefore can be used with any model, EMF or not. The framework consists of the following classes and interfaces.

Command

Interface `Command` is the base interface that all commands must implement. As you might expect, it includes methods like `execute()`, `undo()`, and `redo()` to cause and reverse the command's effect. A command is tested for executability by calling the `canExecute()` method, which is often used to control the enablement of actions bound to the command. Not all commands can be undone; sometimes it is just too hard to maintain all the information needed to reverse the state changes caused by a command. The method `canUndo()` is used to check for undoability of a command. Returning `false` from `canUndo()` indicates that the `undo()` and `redo()` methods are unimplemented.

The command interface includes a few more methods, two of which are particularly interesting: `getResult()` and `getAffectedObjects()`. Implementing these two methods is optional, but they can be quite useful. The `getResult()` method is used by a command implementation to return what should be considered the result of its execution. This allows one to implement compound commands where the result of one command can be the input of another. For example, if a generic object copy command returns the copy as its result, then a copy-and-paste command could simply be composed from the copy command and a generic add command; the add command's input would be the copy command's result.

The second method, `getAffectedObjects()`, is used to return the objects that have been changed during the last `execute()`, `undo()`, or `redo()` call. The EMF.Edit UI framework uses the affected objects to control the UI selection to highlight the effect of the command. The `getAffectedObjects()` method often returns the same thing as `getResult()`, but not always.

AbstractCommand

Class `AbstractCommand` is a convenient partial implementation of the `Command` interface that most commands extend from. It's a small class that doesn't really do a lot; non-trivial commands need to override most of the methods anyway. However, it does provide an important implementation of the `canExecute()` method, which calls out to another protected method, `prepare()`, like this:

```
public boolean canExecute() {
    if (!isPrepared) {
        isExecutable = prepare();
        isPrepared = true;
    }
    return isExecutable;
}
```

Notice that the `prepare()` method will be called only once, regardless of how often `canExecute()` is called. This is particularly significant if the enablement checking of the actual command (subclass) involves a lot of computation. With this design, all a subclass needs to do differently is override and put the enablement checking code in the `prepare()` method, instead of `canExecute()`.

CommandStack

Interface `CommandStack` defines the interface for executing and maintaining commands in an undoable stack. Like `Command` itself, it includes methods to `execute()`, `undo()`, and `redo()` a command, only here it also maintains the command on the stack. The `canUndo()` and `canRedo()` method can be used to determine if there are any commands on the stack to undo or redo, respectively. Other methods are provided to access the command at the top of the stack or the next command to undo or redo, to flush the stack, and to add change listeners (interface `CommandStackListener`) to the stack.

BasicCommandStack

Class `BasicCommandStack` is a basic implementation of the `CommandStack` interface. It is fully functional and can be used, as is, as the command stack for an EMF.Edit-based editor. One interesting observation relates to non-undoable commands. It can be seen when looking at the implementation of the `canUndo()` method in `BasicCommandStack`:

```
public boolean canUndo() {
    return top != -1 && ((Command) commandList.get(top)).canUndo();
}
```

Notice that the command stack will not only return `false` from the `canUndo()` method when there are no commands to undo (that is, `top == -1`), but also if the last executed command cannot be undone (that is, `((Command) commandList.get(top)).canUndo()` returns `false`). This is a very important observation in that it implies that if a non-undoable command is executed, the entire stack of commands before it will no longer be undoable either. The undo list is effectively wiped out at this point, so it's important to consider this before executing a non-undoable command on the command stack.

CompoundCommand

Class `CompoundCommand` is probably the single most commonly used command in the framework. It's a very useful class that allows you to build higher-level commands by composing them from other more basic commands. The `execute()` method simply calls `execute()`, in order, on each of the commands it's composed from; `canExecute()` returns `true` if all the commands can execute; and so on.

In addition to delegating its implementation to the composed commands, `CompoundCommand` provides a few useful convenience methods. For example, the `appendAndExecute()` method can be used to append a command and immediately execute it. This is a particularly good way of recording a set of executed commands, which can later be undone by simply calling `undo` on the compound command. This technique is often used to conditionally execute commands in the `execute()` method of another command. Chapter 14 includes an example of this.

Another useful convenience method is `unwrap()`, which returns the underlying (composed) command if there is only one, or itself (that is, `this`) otherwise. It allows you to optimize away compound commands that only have one command under them.

Other Common Commands

In addition to the classes we've just described, there are a few more common command implementation classes that are less commonly used but nevertheless quite handy. We won't describe them here, but Chapter 15 includes an overview of the whole set of common command classes.

EMF.Edit Commands

EMF.Edit includes a set of generic commands for modifying EMF models. Its commands extend from and build on the interfaces defined in the common command component but impose a dependency on the Ecore model; they use the reflective `EObject` API to operate on EMF modeled objects. The following basic commands are provided:

1. `SetCommand` sets the value of an attribute or reference on an `EObject`.
2. `AddCommand` adds one or more objects to a multiplicity-many feature of an `EObject`.
3. `RemoveCommand` removes one or more objects from a multiplicity-many feature of an `EObject`.
4. `MoveCommand` moves an object within a multiplicity-many feature of an `EObject`.
5. `ReplaceCommand` replaces an object in a multiplicity-many feature of an `EObject`.
6. `CopyCommand` performs a deep copy of one or more `EObjects`.

These commands work on any EMF model, and their implementations fully support undo and redo. All of these commands, except `CopyCommand`, are primitive commands that simply perform their function when called. A `CopyCommand`, however, is actually composed from instances of two other special-purpose primitive commands, `CreateCopyCommand` and `InitializeCopyCommand`, which create and initialize a shallow copy object, respectively. The `CopyCommand` works by building up `CompoundCommands` composed of `CreateCopyCommands` and `InitializeCopyCommands` for the individual objects that need to be copied. It then invokes the compound commands to perform the deep copy. The reason for this approach is to allow easy customization of any part of the copy operation.

In addition to the basic commands, the EMF.Edit command framework includes some higher-level commands that are built using the basic commands along with some of the classes from the common command component:

1. `CreateChildCommand` allows you to create a new object and add it to a feature of an `EObject`. It uses an `AddCommand` or `SetCommand` to add the child, depending on whether the feature is multiplicity-many or not.
2. `CutToClipboardCommand` invokes a `RemoveCommand`, and then saves the removed object on the clipboard.
3. `CopyToClipboardCommand` simply saves a pointer on the clipboard; it doesn't actually change the model.
4. `PasteFromClipboardCommand` uses a `CopyCommand` to copy the object on the clipboard and then an `AddCommand` to add the copy to the target location.
5. `DragAndDropCommand` uses `CopyCommand`, `RemoveCommand`, and `AddCommand` to implement standard drag-and-drop operations.

That's it for the predefined commands, but there are a couple of other interesting features of the `EMF.Edit` command package that we should point out. The first thing has to do with overrideability of the commands.

AbstractOverrideableCommand

Most of the generic commands extend from an `EMF.Edit` abstract base class, `AbstractOverrideableCommand`, which is a subclass of the common `AbstractCommand`. The `EMF.Edit` base class adds the ability to attach another command to override it (via delegation). For example, the `execute()` method looks like this:

```
public final void execute() {
    if (overrideCommand != null)
        overrideCommand.execute();
    else
        doExecute();
}
```

If an `overrideCommand` is attached, the `execute()` method is delegated to it, otherwise the `doExecute()` method is called. This pattern is used for all of the `Command` methods.

You may be wondering why this is needed, given that you can always override a command simply by subclassing it. The key word is “you.” The `EMF.Edit` framework expects you to use ordinary subclassing to customize the generic commands with any model-specific specializations you may need. At the same time, the framework wants to reserve an orthogonal dimension of overrideability for itself.

The implication of this is that if you want to subclass an `EMF.Edit` overrideable command, you need to override the `doExecute()` method instead of `execute()`, `doUndo()` instead of `undo()`, and so on. Other than that, you typically shouldn't need to concern yourself with the `OverrideableCommand` mechanism.

CommandParameter and Static Create Methods

Another special feature of the `EMF.Edit` command package has to do with the creation of commands. We already mentioned in Section 3.2.3 that `EMF.Edit` commands are created using an `EditingDomain`, which, in turn, delegates to item providers. The `EditingDomain` interface contains (among other things) a command factory method, `createCommand()`, that looks like this:

```
Command createCommand(Class commandClass,
                      CommandParameter commandParameter);
```

Notice that class `CommandParameter` is used to pass the command arguments in a generic way. To use this method to create a command you would first need to create a `CommandParameter` object, set the command's parameters into it, and then call the `create` method, passing it the command class (for example, `SetCommand.class`) and the parameters.

Rather than making clients go through all that, the EMF.Edit command framework uses a convention of providing static convenience `create()` methods in every command class. Using the static method, you can create and execute a `SetCommand` like this:

```
Command cmd = SetCommand.create(ed, object, feature, value);
```

The static `create()` method will, in turn, create the `CommandParameter` object and call the `createCommand()` method on the `EditingDomain` for you.

EditingDomain

Similar to the way that content and label providers are used to manage viewer access to a model, the EMF.Edit framework uses an editing domain to manage an editor's command-based modification of a model. It does this by providing three main functions:

1. Creating commands, optionally deducing some of their arguments
2. Managing the command undo stack
3. Providing convenient access to the set of EMF resources being edited

In Section 3.2.3 we saw that EMF.Edit's editing domain implementation class, `AdapterFactoryEditingDomain`, implements the first function, command creation, by delegating to an item provider. The second and third functions are handled by maintaining the editor's `CommandStack` and `ResourceSet`, respectively. The three roles are illustrated in Figure 3.12.

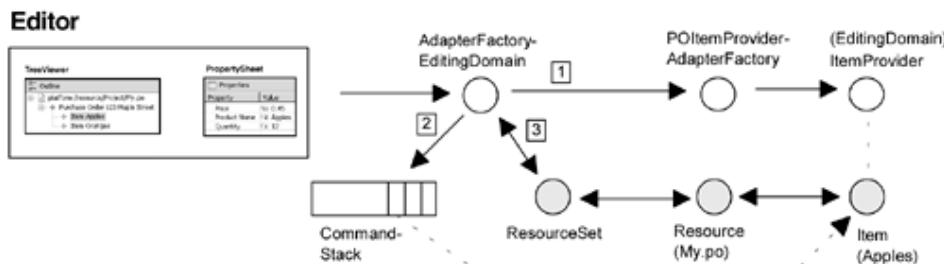


Figure 3.12. The roles of an `EditingDomain`.

Creating Commands

To understand how an editing domain handles its command creation role, let's walk through a simple example. Assume we want to remove one of the items from our purchase order model, `My.po`. The `RemoveCommand`, like most EMF.Edit commands, includes several static `create()` methods. Two of them look like this:

```
public static Command create(EditingDomain domain,
    Object owner,
    Object feature,
    Object value) { ... }

public static Command create(EditingDomain domain,
    Object value) { ... }
```

Notice that the first one takes three arguments in addition to the `EditingDomain`: the `owner` and `feature` being removed from, and the `value` being removed. For our example we would call it like this:

```
Command cmd = RemoveCommand.create(editingDomain,
                                    aPurchaseOrder,
                                    poPackage.getPurchaseOrder_Items(),
                                    aItem);
```

In this example, we provided all the information needed to create the `RemoveCommand`. This would not be the case if we used the second `create()` method instead:

```
Command cmd = RemoveCommand.create(editingDomain, aItem);
```

Notice that here we simply pass the item to be removed, `aItem`, without specifying where to remove it from. This is actually the way an EMF.Edit-based editor implements a **Delete** action; it simply passes the selected object, or objects, to the `RemoveCommand`'s `create()` method. The editing domain is given the added responsibility of filling in the missing arguments. Let's follow through with our example.

As we described in the previous section, the static `create()` methods simply package up their arguments in a `CommandParameter` and then call `createCommand()` on an editing domain. The `AdapterFactoryEditingDomain` simply delegates `createCommand()` to an item provider using the familiar adapter factory delegation pattern:

```
public Command createCommand(Class commandClass,
                             CommandParameter commandParameter) {
    Object owner = ... // get the owner object for the command
    IEditingDomainItemProvider adapter =
        (IEditingDomainItemProvider)
            adapterFactory.adapt(owner, IEditingDomainItemProvider.class);
    return adapter.createCommand(owner, this, commandClass,
                                commandParameter);
}
```

Notice that the `createCommand()` method uses the `owner` object to access the item provider to delegate to (that is, for the object used in the `adapterFactory.adapt()` call). If we used the four-argument `create()` method, then the `owner` is known (`aPurchaseOrder`). However, if we used the two-argument method, then the editing domain will need to compute it.

If you look at the actual implementation, the way `AdapterFactoryEditingDomain` finds the `owner` looks more complicated than it is. This is because it's designed to handle, among other things, removing collections of objects at once. For example, a user might select multiple items from more than one purchase order, and then invoke a **Delete** action. The editing domain handles this case by computing all the owners involved, and then creating a `CompoundCommand` containing a `RemoveCommand` for each owner.

For any given object (to be removed), the `owner` is computed by calling the `getParent()` method on its item provider, another method in the `IEditingDomainItemProvider` interface.¹ The effect of this is that the method `createCommand()` is finally called on the item provider of the purchase order (`aPurchaseOrder`).

¹Actually, there is a `getParent()` method with the same signature in the `ITreeItemContentProvider` interface, which works out fine since we usually have the same item provider implementing both interfaces.

In Section 3.2, we mentioned that most item providers that are also EMF adapters extend from the `EMF.Edit` convenience base class, `ItemProviderAdapter`, which provides a default implementation of many methods. Included among these is an implementation of `createCommand()` that handles all the standard commands provided by the `EMF.Edit` framework. For our example, the item provider will first deduce the final argument, the feature (`items`), and then simply return a new `RemoveCommand` constructed with all four arguments. In Chapter 14, we'll look in detail at how this works and how you can easily override the generic commands with your own model-specific customizations.

Maintaining the Command Stack

The command stack plays a key role in an `EMF.Edit`-based editor. If a single command stack is used pervasively for all changes to the model, editors can also use it to enable the **Save** action (that is, only enable it when the stack is not empty), and to both enable and execute the **Undo** and **Redo** actions.

In general, commands are created and executed in the same place. Since the editing domain serves as the command factory, it would also be an excellent holder for the command stack. Having created the command, the editing domain can then be used to access the command stack to execute it.

So, the only thing that we need to ensure is that the editing domain is available everywhere in the code that needs to change the model (that is, execute commands). For editor actions, the editing domain is readily available from the editor. For property descriptors, the editing domain needs to be located using the resource set, which brings us to the third role of the editing domain.

Accessing the ResourceSet

In its third role, the adapter factory provides convenience methods to load and save resources, as well as convenient access to the resource set. However, the real reason it provides these friendly services is so it can create a special `ResourceSet` that knows about it—one that implements the interface `IEditingDomainProvider`. This is indicated in Figure 3.12 by the arrow on both ends of the line between the editing domain and resource set.

Since a resource is aware of its resource set, and an object is aware of its resource, with this arrangement we can now find the editing domain for any model object. This is important, in that it allows commands to be executed on model objects from anywhere in the code. For example, when a property sheet change is made, the `ItemPropertyDescriptor` `setPropertyValue()` method will locate the editing domain from the object being changed, like this:

```
EditingDomain editingDomain = getEditingDomain(object);
```

Once it has access to the editing domain, it can then create the command and, since it also now has access to the command stack, execute it:

```
editingDomain.getCommandStack().execute(  
    SetCommand.create(editingDomain, object, feature, value));
```

Generating EMF.Edit Code

In Chapter 2, we saw how EMF lets you take a model definition in any of several forms and generate Java implementation code for it. Given the same model definition, we can also use the `EMF.Edit` code generation support to generate item providers and other classes needed to edit the model. The `EMF.Edit` code generator is not a separate tool; it's just another feature of the model generator. As you'll see in Chapter 4, after generating your model, you can generate the `EMF.Edit` parts via the **Generate Edit Code** and **Generate Editor Code** menu items.

When they are needed to hold generated code, the EMF generator will create new projects. We have seen that, by default, model code is generated into the existing project that contains the core and generator models. However, this is not the case for generated EMF.Edit code. As described in Section 3.1.2, the EMF.Edit framework is divided into two separate plug-ins: the UI independent part and the Eclipse UI-dependent part. By default, the code generated by EMF.Edit follows this same pattern. **Generate Edit Code** will generate a plug-in containing the UI independent editing support classes, while **Generate Editor Code** will generate the rest into a separate plug-in that also depends on the Eclipse UI. You can, however, override this and force the generator to put everything into a single plug-in, if that's what you want.

Edit Generation

Invoking **Generate Edit Code** in the EMF generator will generate a complete plug-in containing the UI independent portion of a model editor. It produces the following:

1. A set of typed item provider classes, one for each class in the model.
2. An item provider adapter factory class that creates the generated item providers. It extends from the model-generated adapter factory base class described in Chapter 2.
3. A `Plugin` class that includes methods for locating the plug-in's resource strings and icons.
4. A plug-in manifest file, `plugin.xml`, specifying the required dependencies.
5. A property file, `plugin.properties`, containing the externalized strings needed by the generated classes and the framework.
6. A directory of icons, one for each model class.

The most important of these is the set of item provider implementation classes. As described in Section 3.2.5, there are two possible item provider patterns: Stateful or Singleton. The generator gives you the option on a class-by-class basis to generate a Stateful, a Singleton, or no item provider. The chosen pattern only affects the generated `create()` method in the adapter factory and not the generated item provider itself. The generated item providers are implemented using the pattern-neutral approach described in Section 3.2.5.

Choosing not to generate an item provider for a class would be appropriate if you plan to never display instances of it or if you don't need to customize it and can therefore use the EMF.Edit reflective item provider, class `ReflectiveItemProvider`, for it.

The generated item providers mix in all the interfaces needed for basic support of the standard viewers, commands, and the property sheet:

```
public class PurchaseOrderItemProvider extends ItemProviderAdapter
    implements
        IEditingDomainItemProvider,
        IStructuredItemContentProvider,
        ITreeItemContentProvider,
        IItemLabelProvider,
        IItemPropertySource { ...
```

It extends, either directly or indirectly, from the EMF.Edit item provider base class, `ItemProviderAdapter`.

If you look at a generated item provider class, you'll notice that most of the methods from the item provider interfaces are actually implemented in the base class generically, or by calling out to a few simple methods that are implemented in the item provider subclasses. We'll cover these methods, along with all the EMF.Edit generator patterns, in detail in Chapter 10.

Editor Generation

Generate Editor Code is used to generate a fully functional editor plug-in that will allow you to view instances of the model using several common viewers and to add, remove, cut, copy, and paste model objects, or to modify the objects in a standard property sheet, all with full undo/redo support. The following is generated in the editor plug-in:

- An integrated Eclipse workbench editor
- A wizard for creating new model instance documents
- An action bar contributor that manages the popup menus, and toolbar and menu bar items
- A `Plugin` class that includes methods for locating the plug-in's resource strings and icons
- A plug-in manifest file, `plugin.xml`, that specifies the required dependencies and extensions of the editor, wizard, and action workbench extension points
- A property file, `plugin.properties`, containing the externalized strings needed by the generated classes and the framework
- A directory containing icons for the editor and model wizard

The generated editor is a very functional multipage editor. The Outline view displays the model file in a tree viewer. Each page of the editor is synchronized with it and demonstrates a different way of displaying the model. The following pages are generated by default:

- **Selection** shows a tree viewer similar to the one in the Outline view.
- **Parent** is an inverted tree showing the container path from the element selected in the Outline view back to the root.
- **List** shows a list viewer containing the children of the selection in the Outline view.
- **Tree** shows another tree viewer, only rooted at the current selection.
- **Table** shows a table viewer containing the children of the current selection.
- **TableTree** is the same, only using a table tree viewer.

The generated wizard allows you to create a new model instance document containing a single root object of one of the model's types. The generated default implementation provides a drop-down list of concrete classes in the model, from which the user selects an appropriate root.

Regenerating EMF.Edit Plug-Ins

When regenerating into existing projects, the EMF generator supports the same kind of merging for EMF.Edit code as it does for model code, which we described in Chapter 2. You can edit the generated classes to add methods and instance variables, or to modify the generated ones. As long as you remove the `@generated` tags from any generated methods that you change, your modifications will be preserved during the regeneration.

As shown in the last two sections, EMF.Edit also generates three types of non-Java content: property files, icons, and manifest files. Generated property files contain the translated text strings (resources) referenced by the generated code. You can manually add new resource strings or edit the generated ones and then later regenerate without losing your changes. Any newly generated strings will be added, but unused ones, whether initially generated or not, will never be removed. You will need to manually remove them.

Every icon generated by EMF.Edit is a uniquely colored version of a generic icon. The generated icons are expected to be replaced by properly designed model-specific ones, and therefore, the generator will never overwrite an existing icon.

The generator also never overwrites manifest files. There is no automatic merge support either, because it's rarely needed. If you've change a generated manifest file by hand (for example, to add a new extension point), and then later change the model in a way that affects the generated *plugin.xml*, then you should rename it (to *plugin.tmp* for example), run the generator to produce the new *plugin.xml*, and then manually merge your changes into the newly generated version.

Chapter 4. Using EMF—A Simple Overview

Now that you have been introduced to EMF, it's time to get personally acquainted.

In Chapter 2, we explained EMF's notion of a model, examining its conceptual and concrete, serialized forms, and discussing the framework and generated Java code that realize it. In Chapter 3, we looked at EMF.Edit's contributions, primarily item providers for model objects and a command framework, and how these can form the basis for a structured editor for any EMF model. Now, let's put all of this into action.

In this chapter, we will walk through the process of creating an EMF application from a data model in each of the four forms—annotated Java, UML, XML Schema, and Ecore itself—discussed in Chapter 2. Each step in the process will be explained, but there is an expectation that you are already fairly familiar with the interface of the Eclipse platform and the Java Development Tools (JDT). If this is not the case, you may wish to consult the *Workbench User Guide* and the *Java Development User Guide*, which are both available through Eclipse's help system.

We hope to show how easy it can be to go from a high-level model description, to a working implementation of that model, to a capable and feature-rich editor for it. You are encouraged to roll up your sleeves, fire up your copy of Eclipse, follow along, and see for yourself just how painless this task can be with EMF.

Example Model: The Primer Purchase Order

In Chapter 2, we saw a simple model of a purchase order that we mentioned was based on the central example from the W3C recommendation for XML Schema Part 0: Primer [2]. Let's expand our model, so that it includes a few more different elements of Ecore and more closely resembles the example from the primer.

Figure 4.1 shows the UML class diagram for our expanded primer purchase order model.

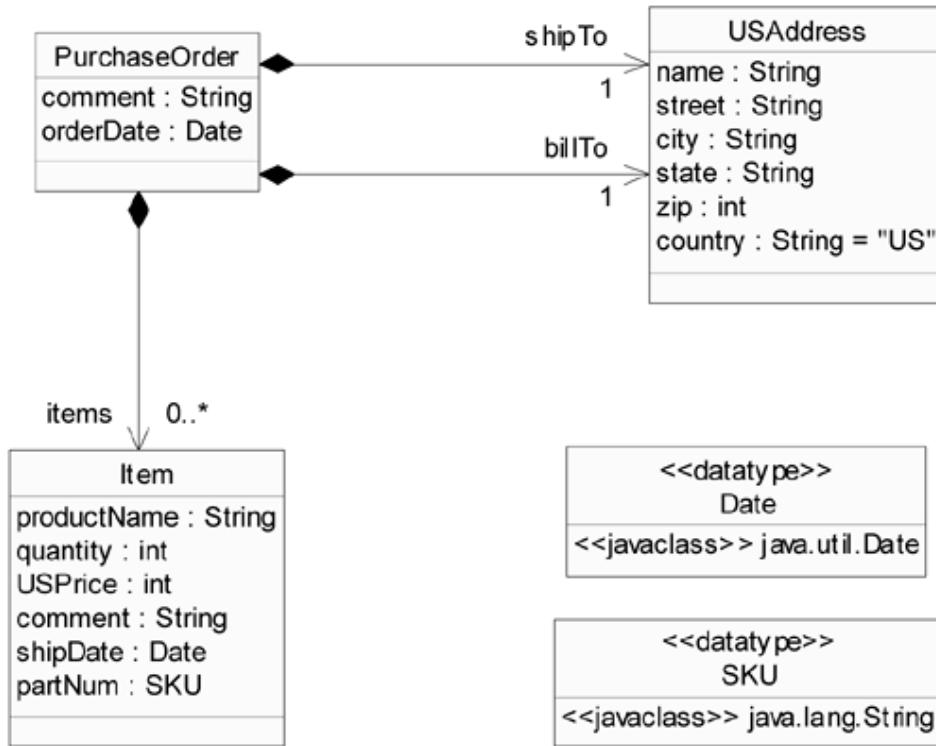


Figure 4.1. Primer purchase model in UML.

Instead of simply representing addresses with a string, we have added a class called **USAddress**, whose attributes are the components of an address in the United States of America. Now that addresses are represented by a class, **shipTo** and **billTo** can no longer be simple attributes of a **PurchaseOrder**. In UML, we show composition associations (also called by-value aggregations) between **PurchaseOrder** and **USAddress**; they correspond to containment references in Ecore. We have also added a number of attributes to the **PurchaseOrder** and **Item** classes. In particular, notice the **orderDate** attribute of **PurchaseOrder** and the **shipDate** and **partNum** attributes of **Item**, for which we have defined new data types. **Dates** will be represented in Java by the `java.util.Date` class and **SKUs** (stock keeping units) by `java.lang.String`. Why, you may ask, are we defining this second type, instead of simply using the built-in type for strings?

The purchase order in the XML Schema primer defines a **SKU** simple type as a restriction on string so that only values matching a certain pattern are accepted. We may wish to implement a similar restriction in the factory methods responsible for serializing and de-serializing values of this type.

The last thing to mention about this purchase order model is that its constituent classes and data types all reside in a package, which we have called **ppo**. EMF requires that all classes and data types belong to a package; Java's default, nameless package is not supported. When a UML class model is converted to Ecore and there are objects that are not contained in a package, a package will be created for them.

Creating EMF Models and Projects

As we discussed in Section 1.2.2, the work that we do in Eclipse lives in the workspace, in a number of projects, or groups of related folders and files. In the context of creating Eclipse plug-ins with the Plug-in Development Environment (PDE), a project corresponds to a single plug-in.

Typically, an EMF model, which may include one or more packages, will live in one plug-in, and hence will be developed in one project, while the UI-independent and UI-dependent portions of the editor will reside in two others. We refer to these as the *model*, *edit*, and *editor* plug-ins.¹

In keeping with the Eclipse philosophy of tool interoperability, the EMF development tools are meant to build on and be used with the JDT and PDE. As a result, an EMF project has a Java nature and includes a plug-in manifest, or *plugin.xml*, file. An EMF project is distinguished from other JDT and PDE projects by a generator model file, with an extension of *.genmodel*, and one or more core model files, with extensions of *.ecore*.²

EMF provides two different wizards to help you create these model files. The EMF Model Wizard creates core model files based on the annotated Java files in a project, an XML Schema or a Rational Rose class model, or copied directly from existing core models. A generator model file is also created to decorate the core models with default code generation settings. The EMF Project Wizard is similar: it creates a new project in the workspace as a home for the models it creates. It does not, however, support annotated Java as a model source—a project must already exist to contain the source files.

In the following subsections, we will demonstrate the use of both of these wizards to create EMF models and projects from the four supported data model sources.

Creating EMF Models from Annotated Java

Of the different sources from which we can create EMF models, annotated Java code certainly offers the lowest cost of entry, both because Java programmers already understand the syntax and because the only tool that it requires is a text editor—whether that's the editor provided by the JDT, a third-party Eclipse plug-in, or the “one true editor” that you have been using since the beginning of time. This makes annotated Java a good place to start this discussion.

Figure 4.2 shows the Eclipse platform in the Java perspective, with a single, very simple Java project in the workspace. This project, named *com.example.ppo*, was created with the Java Project Wizard, and it will soon become our model plug-in. To ensure uniqueness, we are following the recommended convention of beginning plug-in IDs with a reversed domain name—assuming, of course, that we own the *example.com* domain. We use the same convention in our package names to ensure uniqueness in the Java package namespace.

¹It is also possible to combine the two editor-related plug-ins, or indeed all three plug-ins, into one. In Chapter 11, we will describe how.

²Generally, these model files live in the same project as the model source, but there is no reason why they could not form their own project instead.

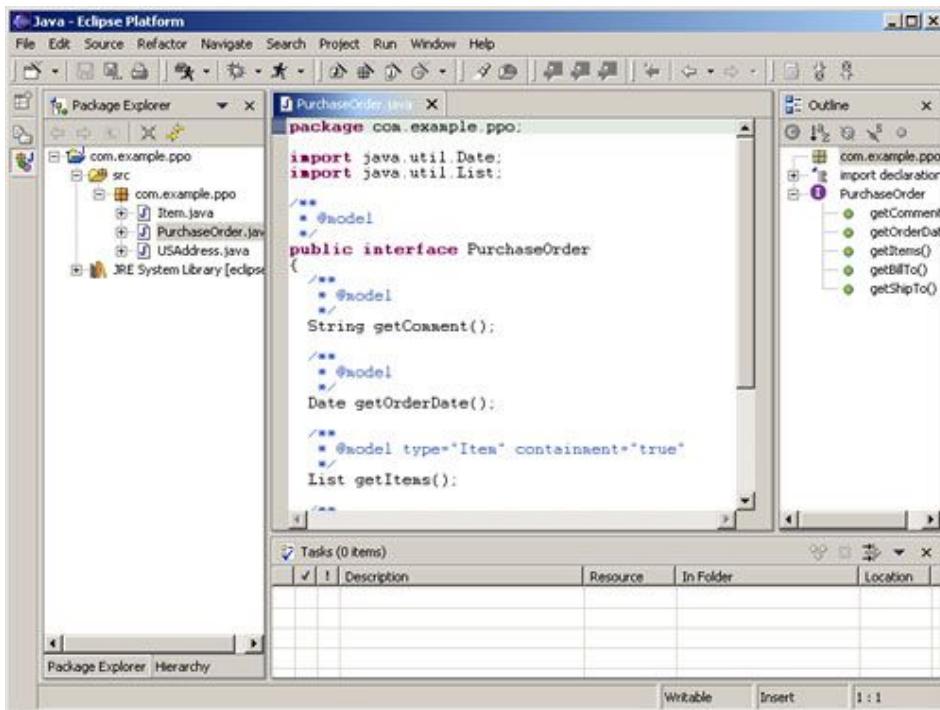


Figure 4.2. Project containing annotated Java files.

The listings of the Java files in the project can be found in Appendix B. Section 2.3.4 introduced the `@model` Javadoc annotations that we use to identify EMF model elements and to specify information that cannot be captured directly in the Java interface declarations. This code demonstrates a little more of the `@model` annotation syntax, which we will now explain briefly. A complete discussion can be found in Chapter 6.

Looking at the `PurchaseOrder` interface, defined in `PurchaseOrder.java`, we find two annotations on `getItems()` that we have already seen:

```
/**
 * @model type="Item" containment="true"
 */
List getItems();
```

Because `items` is a multiplicity-many feature, we use `type` to specify which class to use as its type; `containment` specifies whether the reference should enforce containment semantics.

We find a new annotation on the accessors for `billTo` and `shipTo`:

```
/**
 * @model containment="true" required="true"
 */
USAAddress getBillTo();

/**
 * @model containment="true" required="true"
 */
USAAddress getShipTo();
```

This annotation, `required="true"`, specifies that a valid instance of `PurchaseOrder` must define values for the features that carry it.

In the `IItem` interface, defined in `Item.java`, we specify `dataType="SKU"` on `getPartNum()`, as follows:

```
/**  
 * @model dataType="SKU"  
 */  
String getPartNum();
```

This indicates that we want a new **SKU** data type to be created and used for the `partNum` attribute, instead of **EString**, the existing Ecore data type that maps to `java.lang.String`. As we explained in Section 4.1, we are defining a new data type so that we may supply our own serialization and de-serialization methods. Note that `getShipDate()` does not require a similar annotation because there is no data type in Ecore that corresponds to `java.util.Date`; when a core model is being built, new data types will be demand-created in any such cases.

In the `USAAddress` interface, defined in `USAAddress.java`, we use `default="US"` and `changeable="false"` annotations on `getCountry()`:

```
/**  
 * @model default="US" changeable="false"  
 */  
String getCountry();
```

These annotations define a default value for the `country` attribute and specify that it cannot be changed, suppressing the generation of a `setCountry()` method. Together, these have an effect similar to that of a “fixed” attribute on an “attribute” schema element.

Now, let's see how these annotated Java files are introspected and turned into EMF models.

EMF models, like most other resources in Eclipse, are created using the New Wizard, which can be accessed from the **File** pull-down menu, the **New Wizard** toolbar button, or the context-sensitive menu in the Package Explorer and Navigator views. We'll access it by right-clicking on the `src` folder, and selecting **New**, then **Other**¹⁸⁵ from the pop-up menu. The wizard's first page is shown in Figure 4.3.

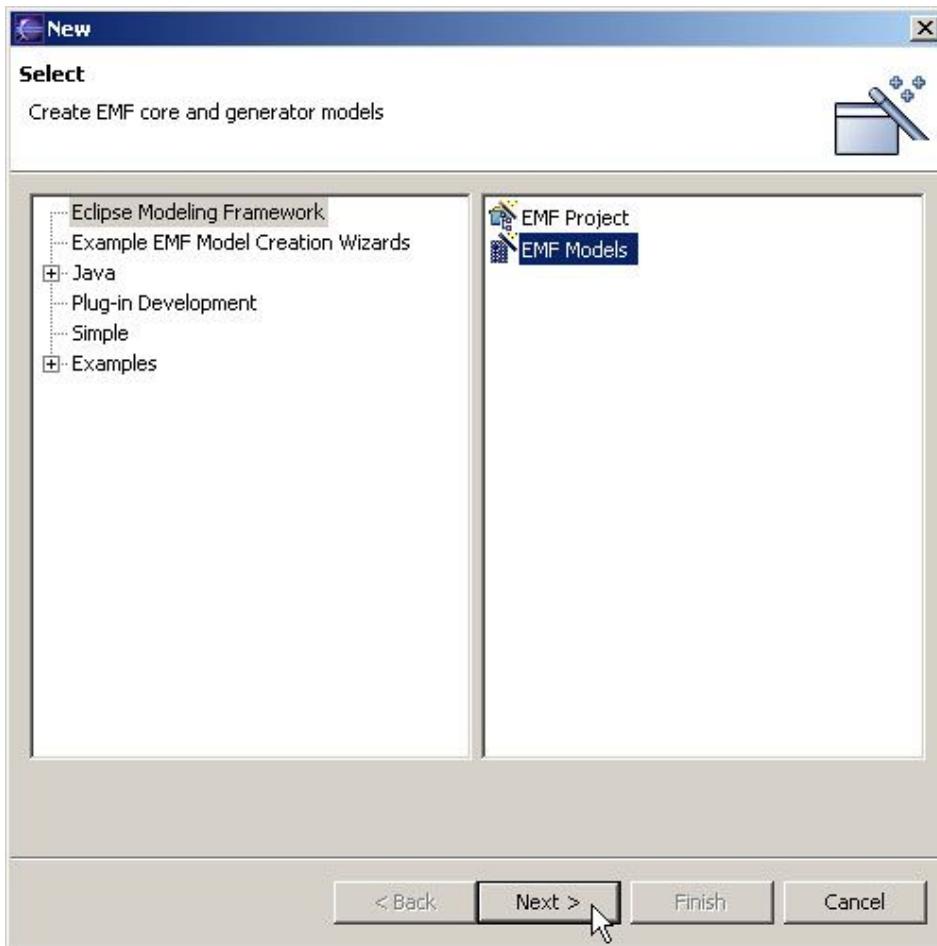


Figure 4.3. The New Wizard's opening page.

In the left column, select “Eclipse Modeling Framework”; on the right, select “EMF Models”; and click the **Next** button to advance to the next page.

This page is illustrated in Figure 4.4. Here, we choose where to create the models and what file name to give the generator model. We will specify a new folder, *com.example.ppo/src/model*, name the model *PrimerPO.genmodel*, and then advance to the next page.

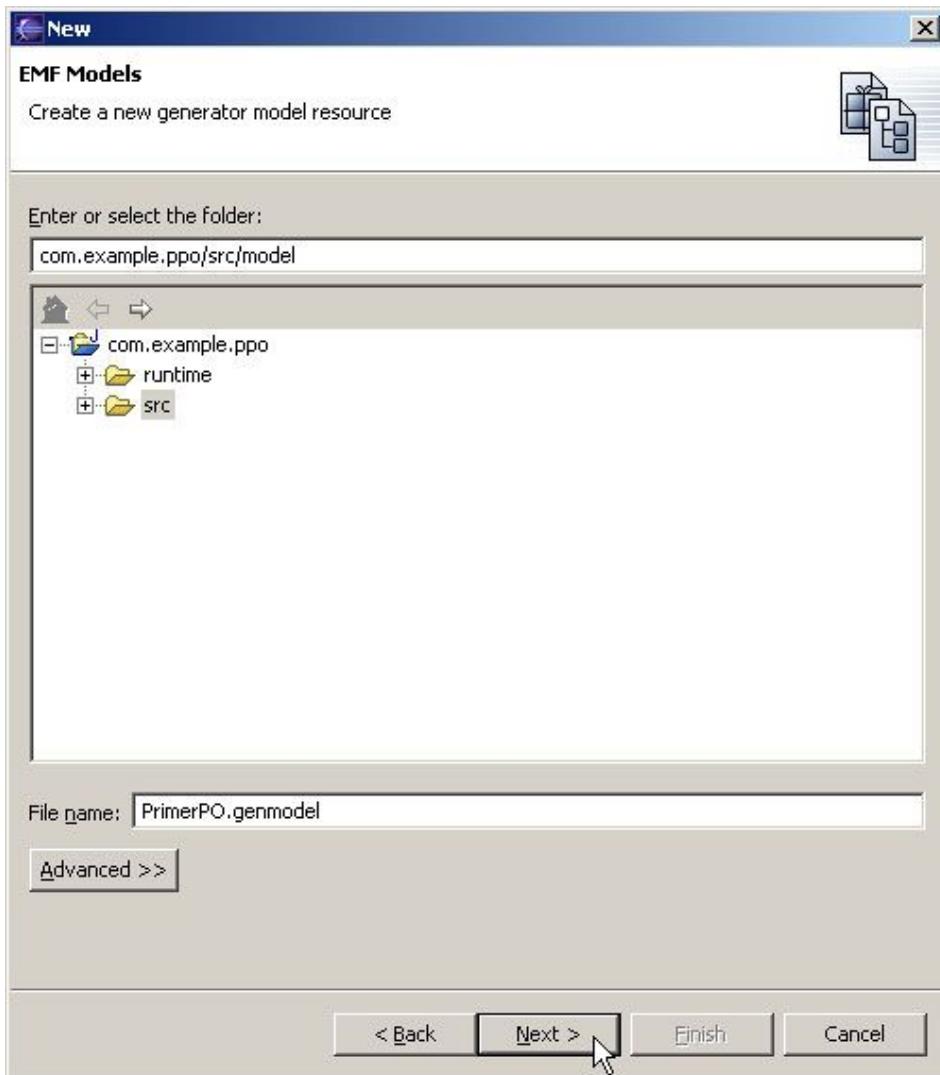


Figure 4.4. Specifying a location for new EMF models.

On the next page, shown in Figure 4.5, we select the source for our model contents: we want to **Load from annotated Java**.

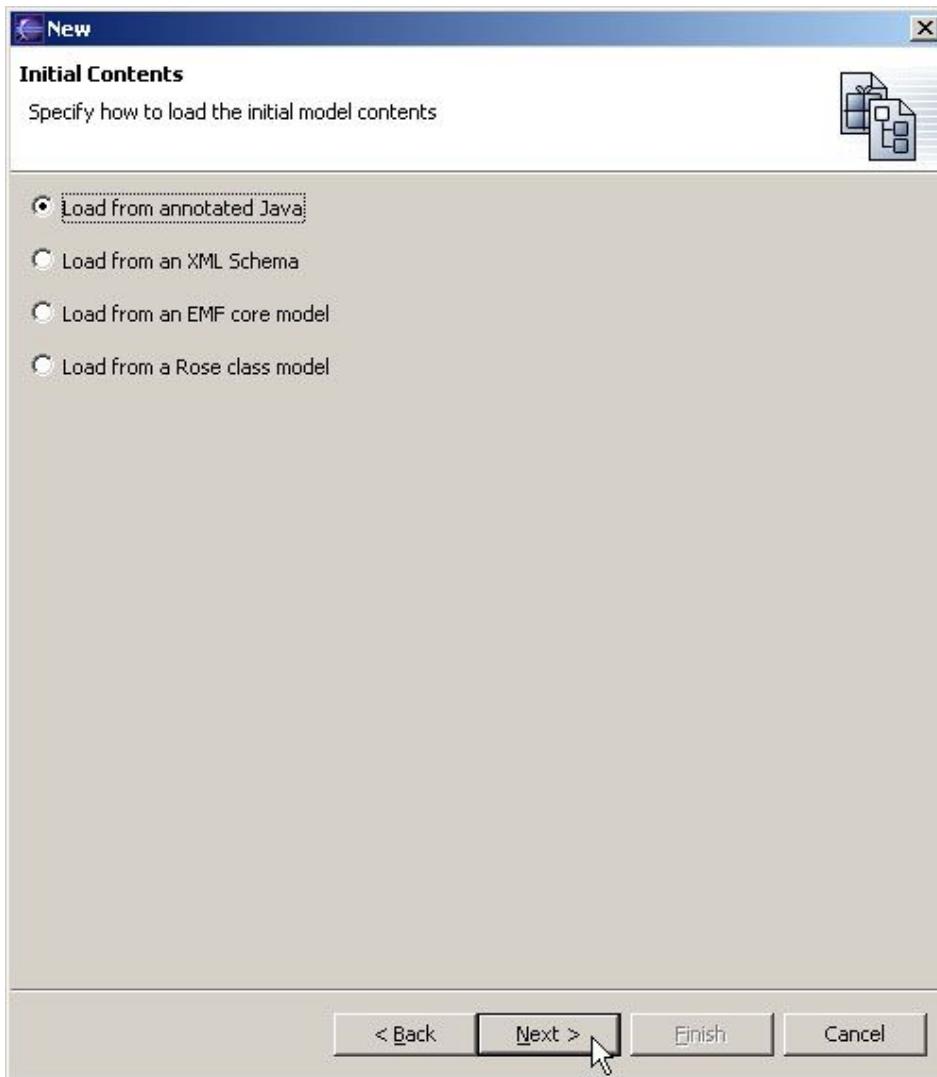


Figure 4.5. Selecting annotated Java as initial contents.

When we advance to the last page of the wizard, the source contained in the project is examined, and we are presented with the list of packages discovered, as illustrated in Figure 4.6. We can select which packages to model and change the file names of the core models that we are creating. Since we only have one package, and the default core model name looks fine, we'll just select that package and click **Finish**.

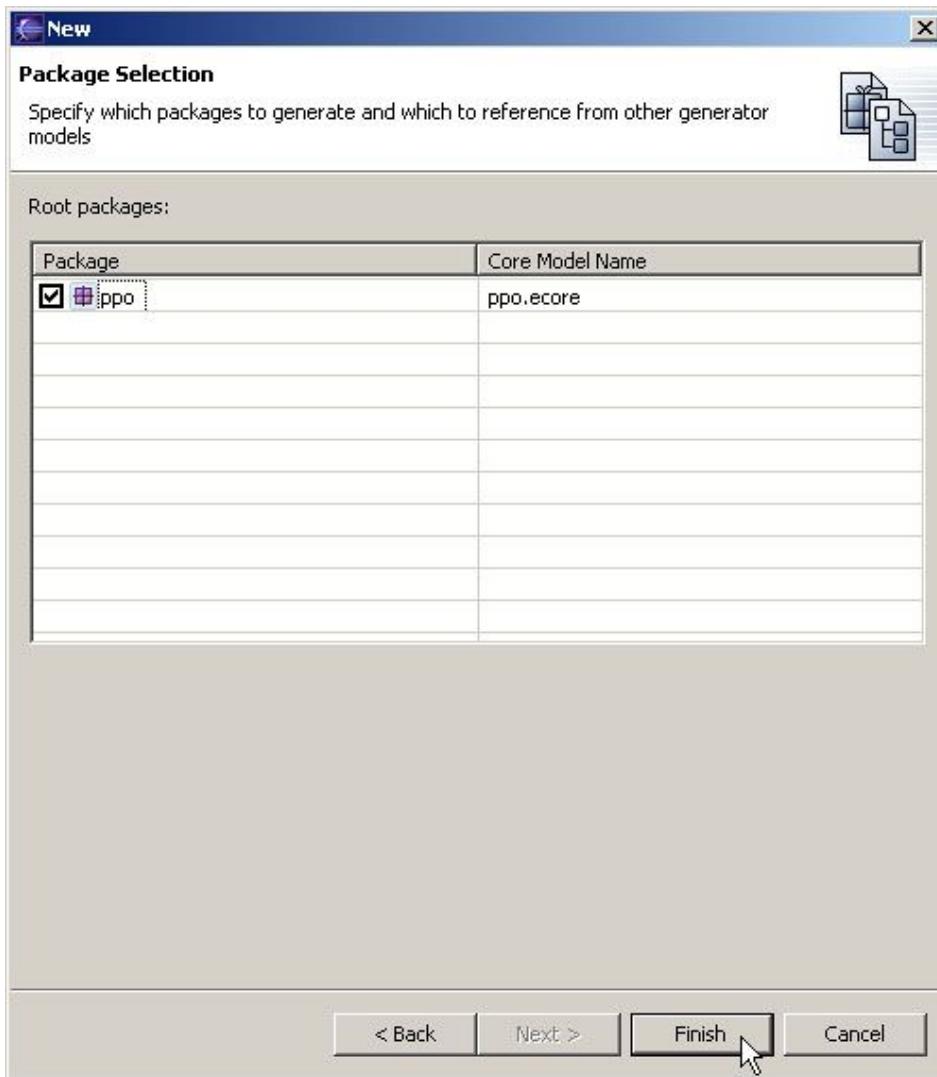


Figure 4.6. Selecting packages discovered in annotated Java.

The wizard creates core and generator models for our package; the latter is opened in the EMF generator, as illustrated in Figure 4.7. Before we proceed to generate code from the models, let's consider how we would create these models from the other sources. Since we are going to recreate the same project in each of the following three sections, you may want to rename them in between, so that you can compare the results.

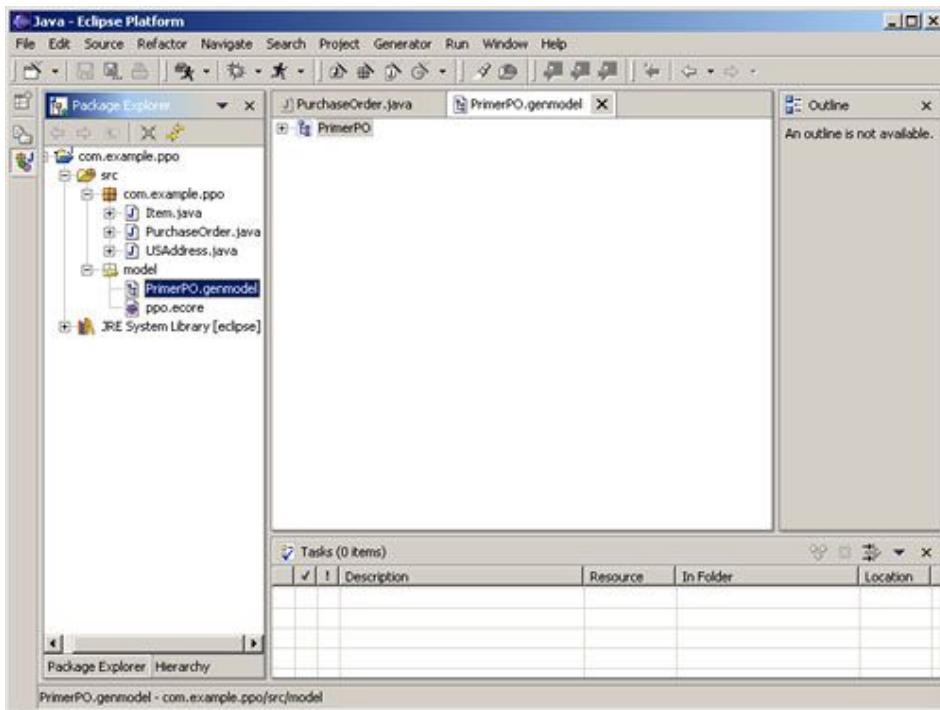


Figure 4.7. Models created and EMF generator opened.

Creating an EMF Project from a Rational Rose Class Model

According to the oft-quoted Chinese proverb, “a picture’s worth a thousand words”—even in this chapter’s simple example, the UML class model of Figure 4.1 would seem to be worth 111 lines of annotated Java code. It’s a notion well understood by those who have bought into modeling. If you fall into this group and already have a visual modeling tool like Rational Rose at your disposal, you should expect to be able to use your Rose data models as input to EMF.

Before we describe how, it is worth briefly mentioning the concept of extensible model properties that is supported by Rose. For the sake of flexibility, Rose provides a mechanism for defining non-UML properties of model elements. EMF uses this mechanism—a property file—to add support for several features of the core and generator models. It is highly recommended that you add the properties from the *ecore.pty* file to Rose models on which you plan to base EMF projects. This file is located in the *rose*/subdirectory of the *org.eclipse.emf.codegen.ecore* plug-in, in the EMF Runtime package. Please see your Rational Rose documentation for information on how to do this. Once added, these properties will be accessible via the **Ecore** tab of the specification dialog box for any class model element.

While these model properties will be discussed thoroughly in Chapter 8, let’s briefly look at two of the model properties for our **ppo** package, illustrated in Figure 4.8.

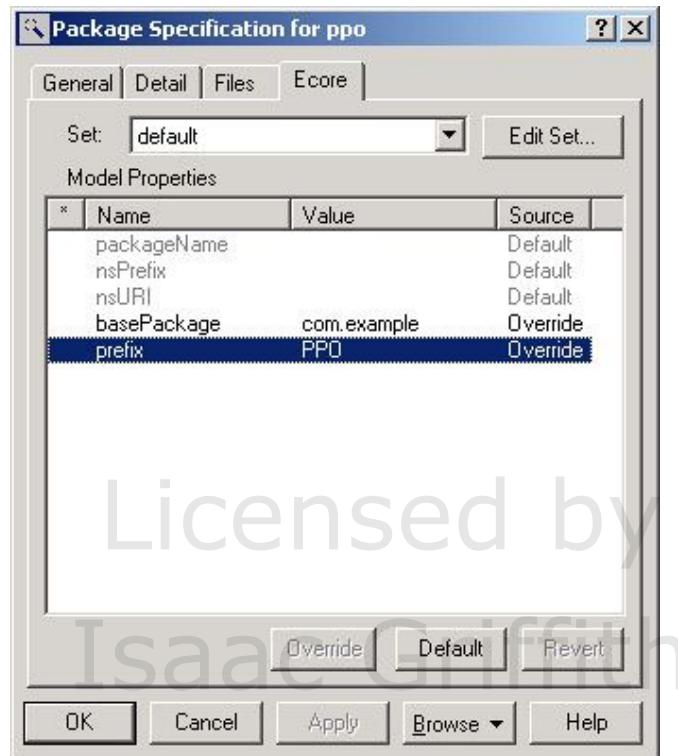


Figure 4.8. Ecore package properties in Rose.

The **basePackage** and **prefix** are properties of the generator model; **basePackage** specifies the Java package of which the generated package will be a subpackage. In other words, it allows us to easily generate code with globally unique package names, without modeling empty, nested packages. For our purchase order package, we set its value to “com.example”. Since our package is named “ppo”, our generated code will go into `com.example.ppo`, the same package we used in our annotated Java example. The other property, **prefix**, is used in forming the names of the generated supporting classes for the package, which, as explained in Sections 2.4.1 and 2.4.2, can include a class representing the package itself, a factory, a switch, and an adapter factory. If we did not set a value here, a default would be generated by capitalizing the first letter of the package name. Since our package name is an abbreviation, we prefer to capitalize it in its entirety; we enter the value “PPO”.

With that detail out of the way, let's now see exactly how we go about creating an EMF project based on this Rose model.

An EMF project is created, like any other project, using the **New Project** Wizard, which can be accessed from the **File** pull-down menu, the **New Wizard** toolbar button, or the context-sensitive menu in the Navigator and Package Explorer views. The wizard's first page is shown in Figure 4.9. In the left column, select “Eclipse Modeling Framework”; on the right, select “EMF Project”; click the **Next** button to advance. On the next page, name the project `com.example.ppo`, leave the default project directory unchanged, and advance again.

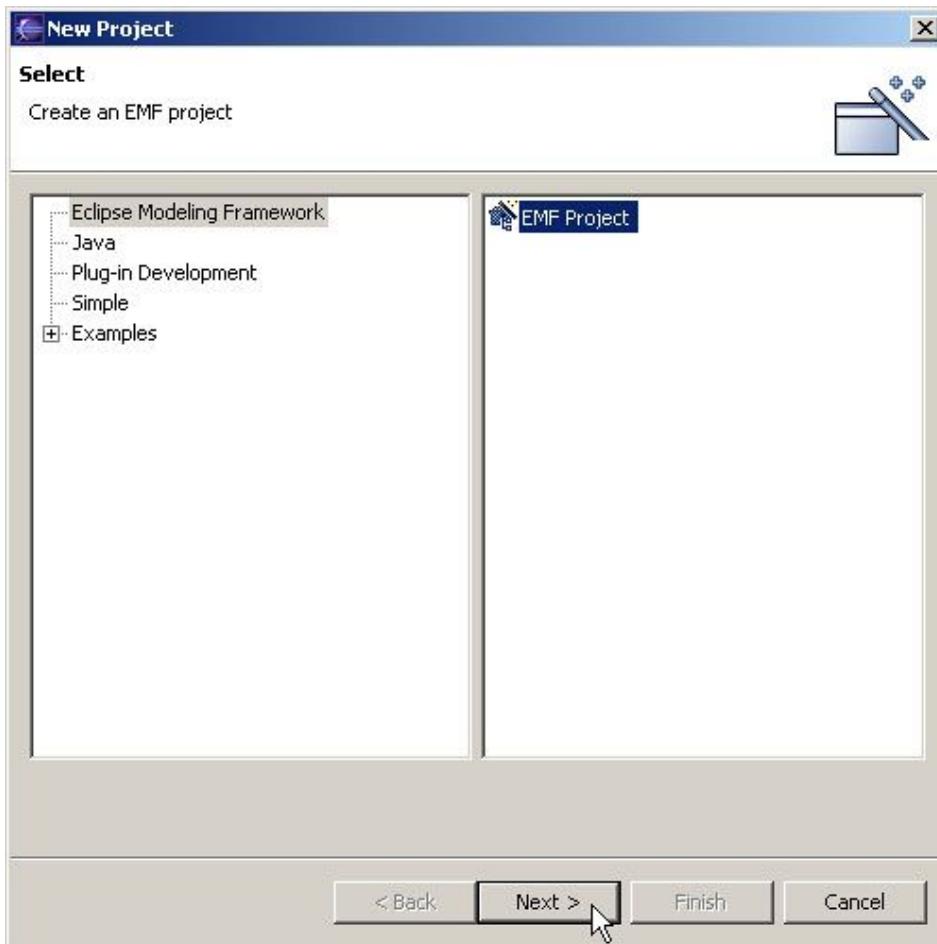


Figure 4.9. The New Project Wizard.

As shown in Figure 4.10, we next select the source for our model contents: we want to **Load from a Rose class model**.



Figure 4.10. Selecting Rose class model as initial contents.

Figure 4.11 shows the next page, where we specify the location of the Rose model file, *PrimerPO.mdl*. An absolute path name can be typed in the text box, or selected by clicking the **Browse** '85 button. If the Rose model had included packages from separate *.cat* files and used path map symbols to locate them, we could define values for those symbols in the **Path map** table. Since our model is contained in a single file, we need not fill in anything here. We can also accept the suggested **Generator model name**, *PrimerPO.genmodel*, and then advance to the last page of the wizard.

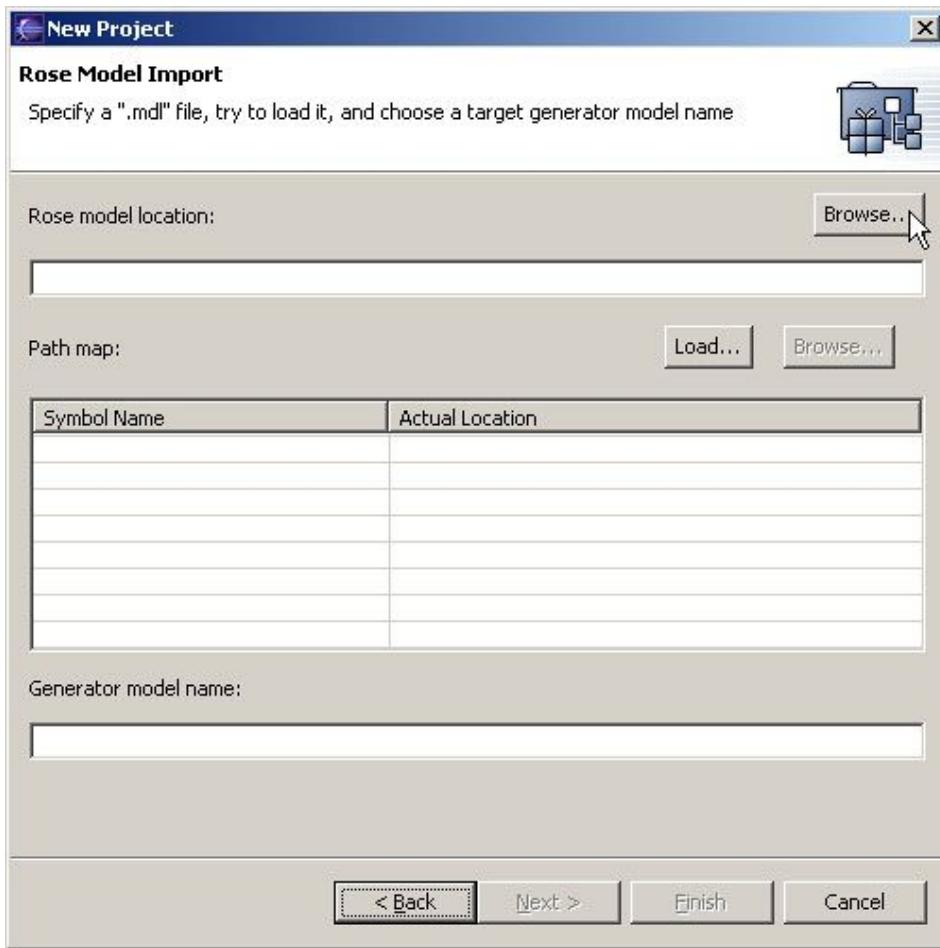


Figure 4.11. Locating the Rose model.

As when we were creating models from annotated Java, this page allows us to select those packages for which we wish to create core models and to specify their file names. This is illustrated in Figure 4.12. We are not shown any nested packages, only those appearing as immediate children of “Logical View” in Rose. If there were more than one of these root packages, we could choose to generate a subset of them and reference dependant packages in existing projects in the workspace.³ Note that we did not have this control when introspecting annotated Java because the references would be made explicitly via `import` statements and fully qualified class names. In any case, we still have only one package, so we will again just select it and press **Finish**.

³We will show exactly how to do this in Chapter 12.

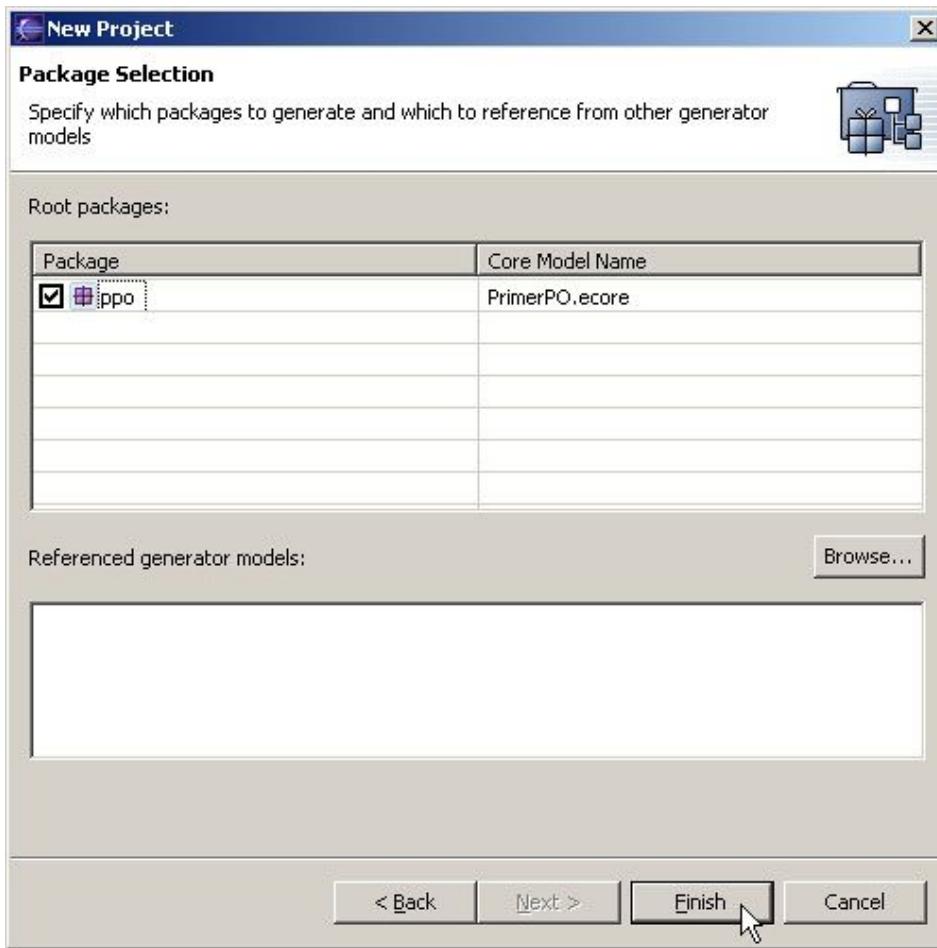


Figure 4.12. Selecting packages from the Rose model.

The wizard creates our project, converts our Rose model into a core model, and creates a generator model for it; once again, this model is opened in the EMF generator. If you compare the core models with those created in Section 4.2.1, you will find that they model the same package, classes, features and data types.⁴ There is a slightly greater difference in the details of the generator models, as reasonable default values can be based only on the clues available in the original model source. In this case, for example, we specified a package prefix as a property in the Rose model, while we knew of no way to control this value with annotated Java.⁵

Creating an EMF Project from an XML Schema

As described in Chapter 2, the design of an application may be centered on the particular XML message structure that it handles. If so, it would make sense to start with the XML Schema that specifies that structure, and from that generate an EMF model for manipulating the data contained in the messages.

⁴There is one small, but notable exception: the **USPrice** attribute of **Item** was named **uSPrice** in the core model created from annotated Java, following the usual convention of beginning feature names with a lowercase character. In the Rose model, we were able to specify the unconventional name used in the XML Schema primer.

⁵In fact, there is a way to do this, and we will describe it in Chapter 6.

Ecore's relationship with XML Schema is considerably less straightforward than its relationship with annotated Java or UML. As a result, trying to apply a simple mapping from an arbitrary XML Schema will often result in more complicated Ecore core models, sometimes unnecessarily so.

In fact, the purchase order schema from the XML Schema primer illustrates this well. It defines **items**, an element of a **PurchaseOrderType**, with a type called **Items**. **Items** is a complex type consisting of a sequence of **item** elements. Here is the relevant portion of the schema:

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    ...
    <xsd:element name="items" type="Items"/>
  </xsd:sequence>
  ...
</xsd:complexType>

...
<xsd:complexType name="Items">
  <xsd:sequence>
    <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
      ...
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

This structure makes perfect sense in a schema: the whole list is delimited by the **items** element, and each member of the list corresponds to a single **item** element. However, when we map complex types in XML Schema to classes in Ecore, the result is unnatural and suboptimal (Figure 4.13):

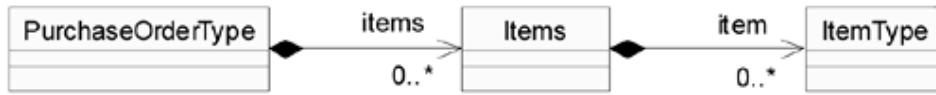


Figure 4.13. Mapping complex XML Schema types to classes in Ecore.

An extra class, **Items**, has been introduced, but all it does is hold the multiplicity-many reference to the **ItemType** class. It is unnecessary because lists are built in to Ecore and implied by multiplicity-many references. Thus, we can make our schema more Ecore-friendly by changing the multiplicity of **items**, itself:

```
<xsd:element name="items" type="Item" minOccurs="0"
  maxOccurs="unbounded">
```

Another problem we will face is that of reversing the mapping, re-creating the XML Schema from the Ecore form. To enable this, we retain additional information about the structure of the schema in the core model using annotations.⁶ We address all of these issues more completely in Chapter 7, but for now, the point is simply that a core model created from an arbitrary XML Schema is going to be more complicated than an equivalent one that came from annotated Java or UML.

For the purposes of this example, we will use a slightly modified, though functionally equivalent, version of the primer's purchase order schema; however, the EMF models that we create will still differ slightly from those we create from the other sources. A listing for our schema, *Pri-merPO.xsd*, can be found in Appendix B.

Again, we begin by launching the **New Project Wizard**, selecting “Eclipse Modeling Framework” on the left, “EMF Project” on the right, and then clicking **Next**. We name the project *com.example.ppo* and advance to the next page, accepting the default directory.

Here, we specify that we want to **Load from an XML Schema**, as shown in Figure 4.14.

⁶Ecore annotations are discussed in Chapter 5.

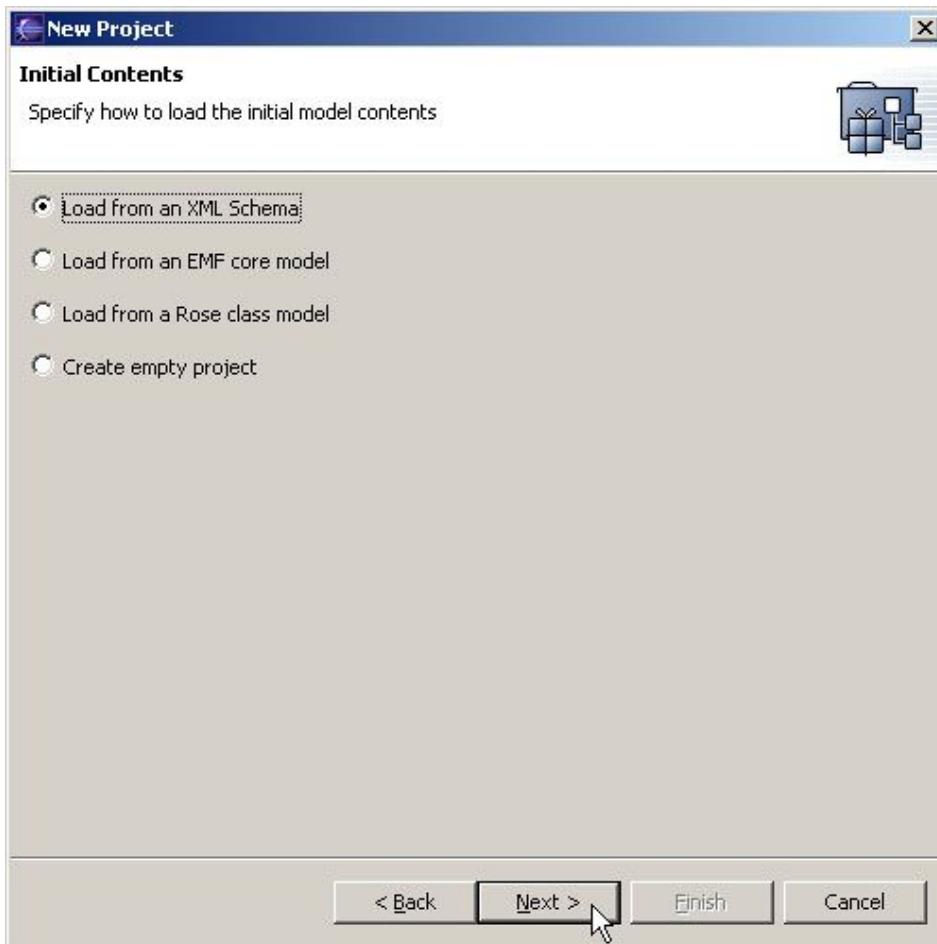


Figure 4.14. Selecting XML Schema as initial contents.

On the following screen, shown in Figure 4.15, we specify the location of the XML Schema using a Universal Resource Identifier (URI).

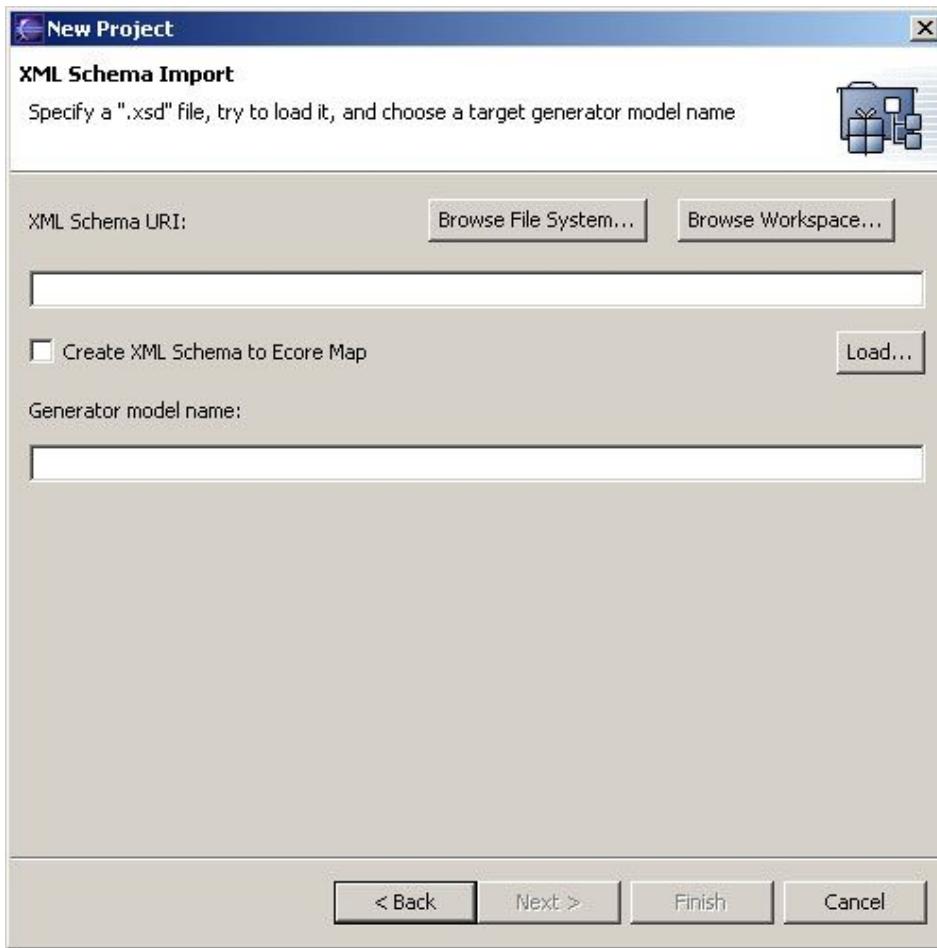


Figure 4.15. Locating the XML Schema.

URIs are the strings used to locate and identify content on the World Wide Web. We discuss them, and EMF's extensive use of them, in Chapter 13. The URI syntax is generic, but we can limit ourselves to a small subset of it for now. A URI's form is specified by its *scheme*, the name given to its first word, which is followed by a colon. At this point, we are only interested in two schemes:

- A *file* scheme URI includes a slash-delimited path to locate a file on a local or networked file system (for example, on Windows, *file:/C:/eclipse/eclipse.exe*).
- A *platform* scheme URI also includes a slash-delimited path; when its first segment is “resource”, the remainder of the path identifies a resource relative to the workspace (for example, *platform:/resource/com.example.ppo/src/model/*). This scheme is internal to Eclipse.

Since we have not put the purchase order schema in the workspace, we click **Browse File System** \85 and locate the file in the **Open** dialog box. The file scheme URI is filled in, and we accept the default **Generator model name**, advancing to the last page of the wizard.

Figure 4.16 shows this page. This time, we are presented with two different Ecore packages that we can model. Why the difference?

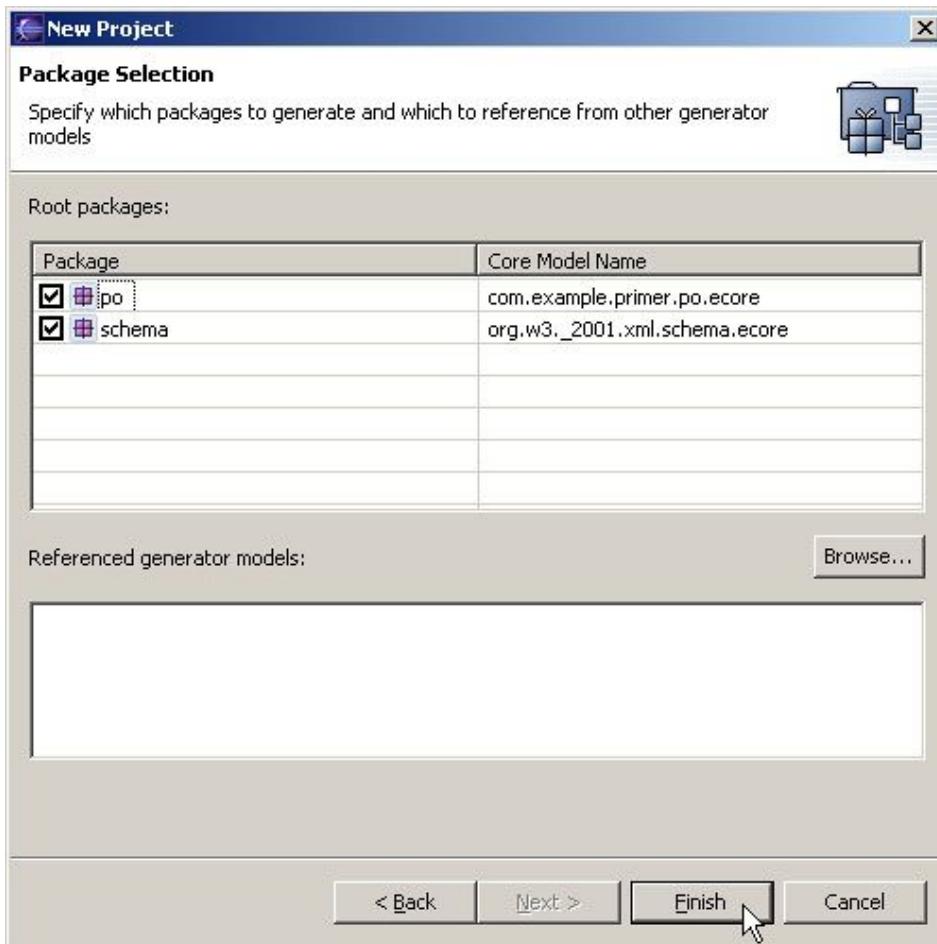


Figure 4.16. Selecting packages from the XML Schema.

The XML Schema recommendation includes a number of built-in simple types that do not map well to built-in Ecore types. Rather than forcing something arbitrary and inappropriate, EMF just creates a package that models as data types any of these that are referenced by our schema. In the case of the purchase order schema, **Decimal** and **Date** are created. By default, all such types are to be represented in Java by `String`. Later, we could select a different Java class and implement our own serialization and de-serialization factory methods, as needed. Or, if we wanted to, we could model the complete set ourselves, and then have all the schema-based core models we create reference that, instead. For now, we simply select both packages, leave the default file names, and click **Finish**.

The wizard creates the new project, and places in it the two new core models and the new generator model.

Creating a Generator Model for a Core Model

The final scenario we will describe is the simplest. In this case, we already have an EMF core model, and we only need to create a generator model that will drive code generation for it. It will seem particularly simple since, by now, we have already seen all the relevant pages in the wizard.

As mentioned in Chapter 2, direct Ecore modeling is currently possible in Eclipse using EMF's tree-based Sample Ecore Editor or EclipseUML from Omondo (www.omondo.com). Other UML tool vendors are also starting to add support for exporting an Ecore XMI model, as we described in Chapter 2. Regardless of the tool used to create it, once you have an Ecore model, you can always use the following approach to import it into EMF.⁷

Figure 4.17 shows the Eclipse platform in the Resource perspective, again with a single *com.example.ppo* project. This time, the project contains the *PrimerPO.ecore* core model in its *src/model* folder. A listing of the file can be found in Appendix B.

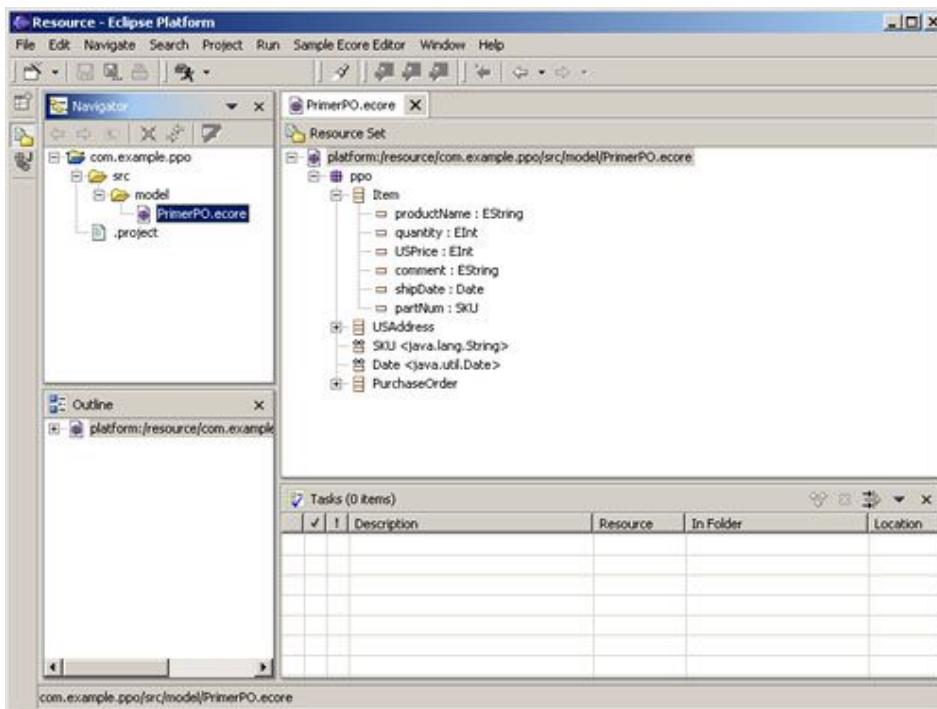


Figure 4.17. Project containing a core model.

To create a generator model, we right-click on the *model* folder and launch the **New** Wizard from the pop-up menu. Once again, we select “Eclipse Modeling Framework” on the left and “EMF Models” on the right, and then advance to the next page. Again, we name the generator model *PrimerPO.genmodel*, leave *com.example.ppo/src/model* as its folder, and advance. Then, we specify that for initial contents we want to **Load from an EMF core model**, as illustrated in Figure 4.18.

⁷Some tools, like EclipseUML, allow you to invoke the EMF wizard and generator directly from within them. If this is the case, you may not need to perform this step at all. You should follow the directions provided with the tool instead.

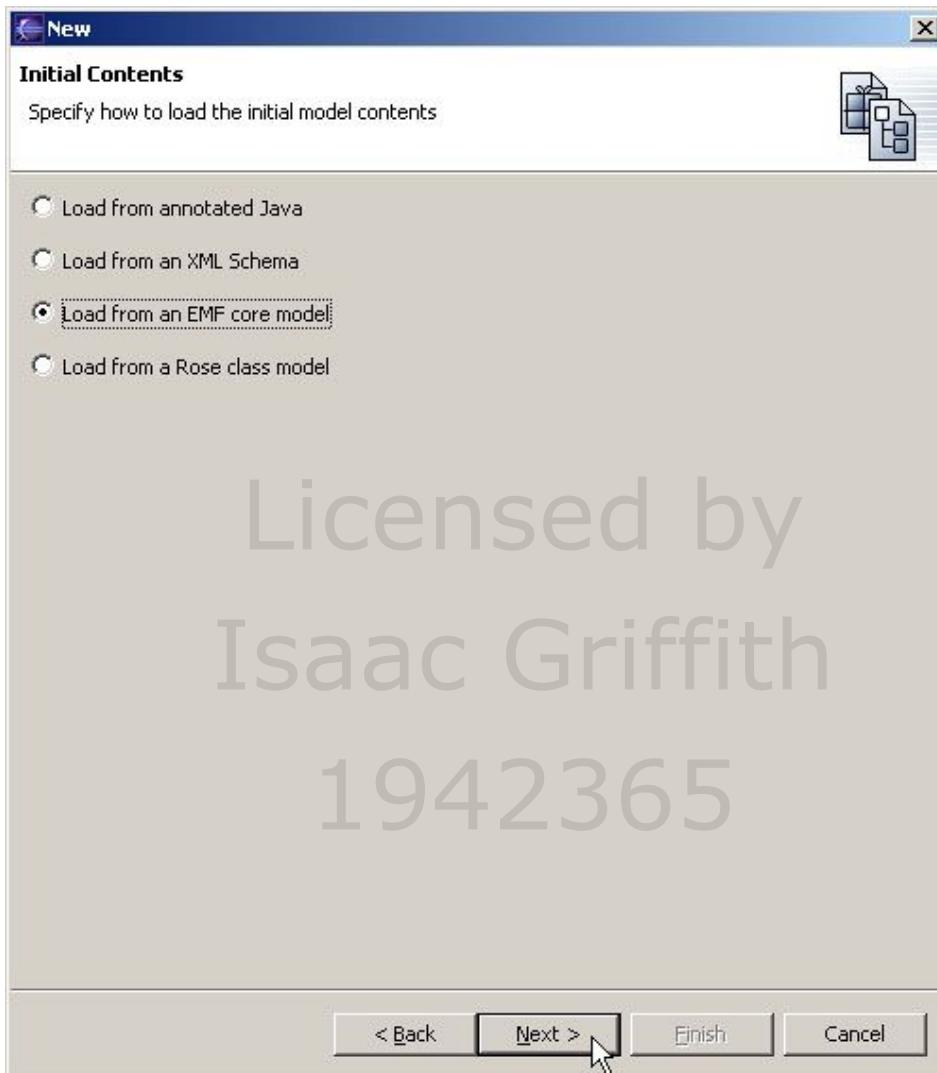


Figure 4.18. Selecting EMF core model as initial contents.

On the next page of the wizard, we must specify the URI for our core model. Since it is located in the workspace, we click the **Browse Workspace\85** button. Figure 4.19 shows the dialog box that appears, with which we select a file. On the left, we click the *model*/folder under *workspace/com.example.ppo/src*; on the right, we select *PrimerPO.ecore*. When we click **OK**, a platform scheme URI will be filled in, and we can advance.

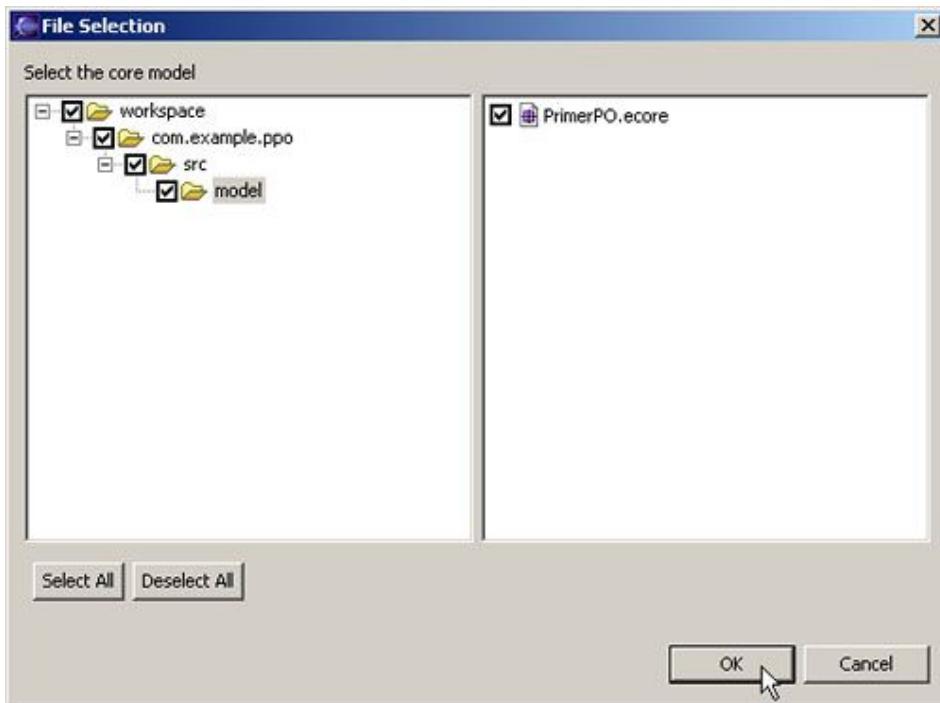


Figure 4.19. Browsing for the core model.

By now, the last page of the wizard should seem very familiar. Once again, we select our single package. Notice that the default name for the core model file is the same as that of the existing file we chose, so we won't really be importing it. Had we specified a different name from the original, the file would be copied. However, we only want to create the generator model, and not make a copy of the core model. We click **Finish** to continue.

Generating Code

At this point, we have taken four different forms of what is essentially the same data model, and from each created two EMF models, a core model and a generator model. These two models will now drive the generation of a complete application for handling our data, purchase order records. Regardless of where our data model came from, once we've converted it into its EMF form, we proceed with code generation in the same way.

We could continue with the models that we obtained in any of the four subsections above. However, we did note that there are some minor differences among them, so for the sake of clarity, we will proceed with the ones that we created from the Rose class model.

Figure 4.20 shows the Eclipse platform in the Resource perspective with that familiar *com.example.ppo* project expanded to show the two models. We have launched the EMF generator by double-clicking on the generator model, *PrimerPO.genmodel*.

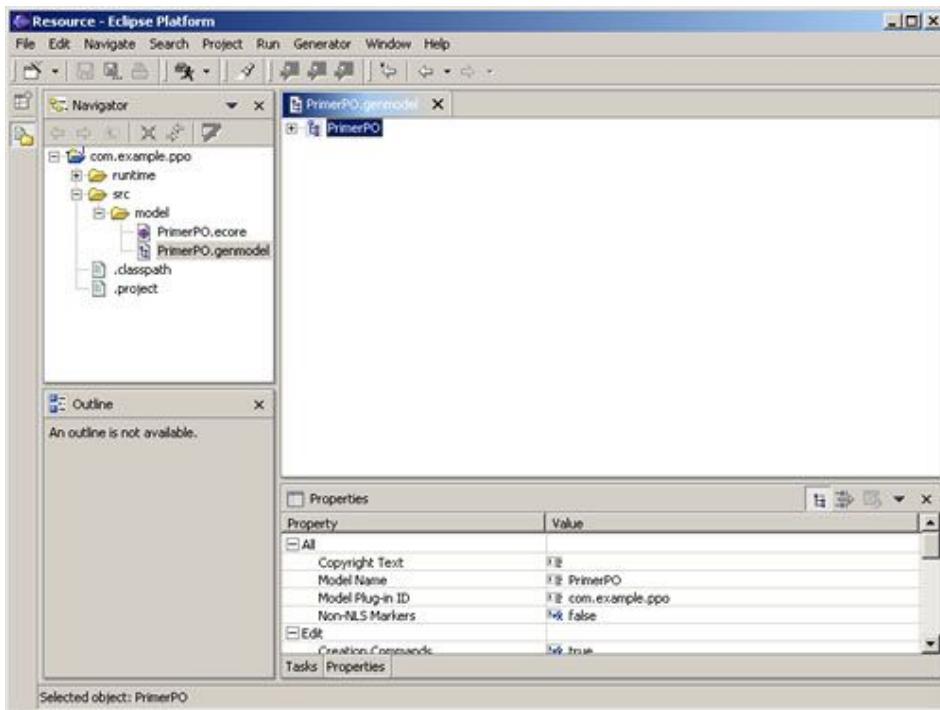


Figure 4.20. Project containing core and generator models with Generator running.

The EMF generator provides a main editor view that contains a single tree of the artifacts to be generated. At the root level, there is an element labeled “PrimerPO” that represents the whole model. On expanding it, we see that it has a single child, a package called “PPO”. Beneath that, we find our classes, “Item”, “USAddress”, and “PurchaseOrder”, and our data types, “SKU” and “Date”. Each class’ children include its attributes, references and operations.

The generator makes another contribution to the workbench: as you select different objects in the tree, their properties are shown in the Properties view. These properties correspond to the attributes of the generator model objects, and they affect the way code is generated. If the Properties view is not visible, you can show it with **Show View**, under the **Window** pull-down menu.

There are a significant number of properties, offering a great deal of control over code generation. For example, we can suppress the generation of switch and adapter factory classes (see Section 2.4.2) for a package, select between the Singleton and Stateful patterns (see Section 3.2.5) for an item provider, and specify for which features change notifications should be passed to the model’s central change notifier (see Section 3.2.4). We will discuss all of the available properties in Chapter 11. For now, the defaults will do just fine.

To generate code, we can right-click on any object in the generator tree. As illustrated in Figure 4.21, the pop-up menu contains four code generation actions: **Generate Model Code**, **Generate Edit Code**, **Generate Editor Code**, and **Generate All**. For any object in the tree, selecting one of the first three of these actions will generate all of the code associated with that object and belonging to the model, edit or editor plug-in, respectively. As you would expect, selecting the last of these actions generates all of the code associated with the selected object across all three plug-ins. As an alternative to using the pop-up menu, we may select an object in the tree and then choose one of the four code generation actions from the **Generator** pull-down menu.

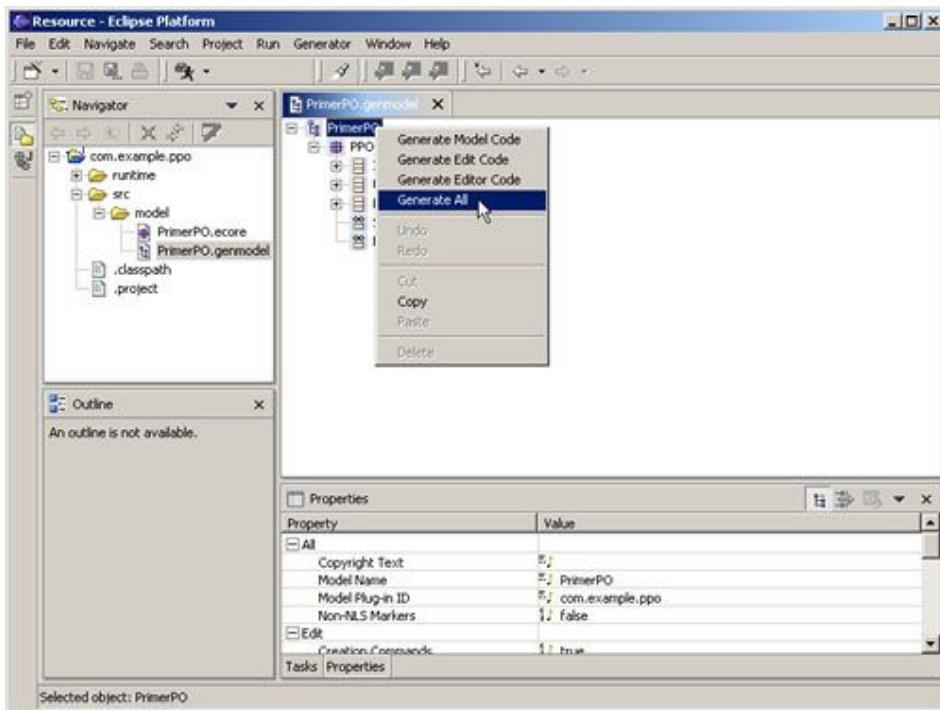


Figure 4.21. Code generation actions.

If any of the code generation actions are grayed out, it simply means that the selected object has no source file associated with it in those plug-ins, or that the current state of the properties dictate that it should not be generated.

Now, we would like to generate code for the whole model, in all three plug-ins, so we simply right-click the “PrimerPO” model object and select **Generate All**.

In a few moments, after code generation is complete, we notice in the Navigator view that a *plugin.xml* manifest file and *src/com* folder, containing all of the generated source, have been added to our existing project, and that new *com.example.ppo.edit* and *com.example.ppo.editor* projects have been created. If automatic building on resource modification is enabled in the Workbench Properties, the application will be compiled as it is generated. As usual, any compilation errors will be marked in the **Tasks** view. There should be none. If automatic building is disabled, we can compile our projects by selecting them in the Navigator and choosing **Rebuild Project** from the **Project** pull-down menu.

Running the Application

In order to test our application, we need to launch a second instance of Eclipse, called a *runtime workbench*, in which our new plug-ins will be accessible. To do so, switch to the Java perspective and, from the **Run** pull-down menu, select **Run As**, then **Runtime Workbench**. If properly configured, the second instance of Eclipse will come up in a few moments. Otherwise, the launch configuration can be modified by selecting **Run\85** from that menu.

In the runtime workbench, we need to create a project to hold an instance of our purchase order model. Once again, launch the New Project Wizard. On the left, select “Simple” and on the right, “Project”. Click **Next**. Give the project a name, say “PPOProject”, and then click **Finish**.

We can now create our purchase order model using the generated PPO Model Wizard. In the Navigator view, right-click the new “PPOProject”, and select **New**, then **Other**’85 from the pop-up menu. The generated wizard was placed under the “Example EMF Model Creation Wizards”, along with the samples that are included with EMF. Select “PPO Model”, as shown in Figure 4.22, and advance to the next page of the wizard. We can accept the suggested file name, *My.ppo*.

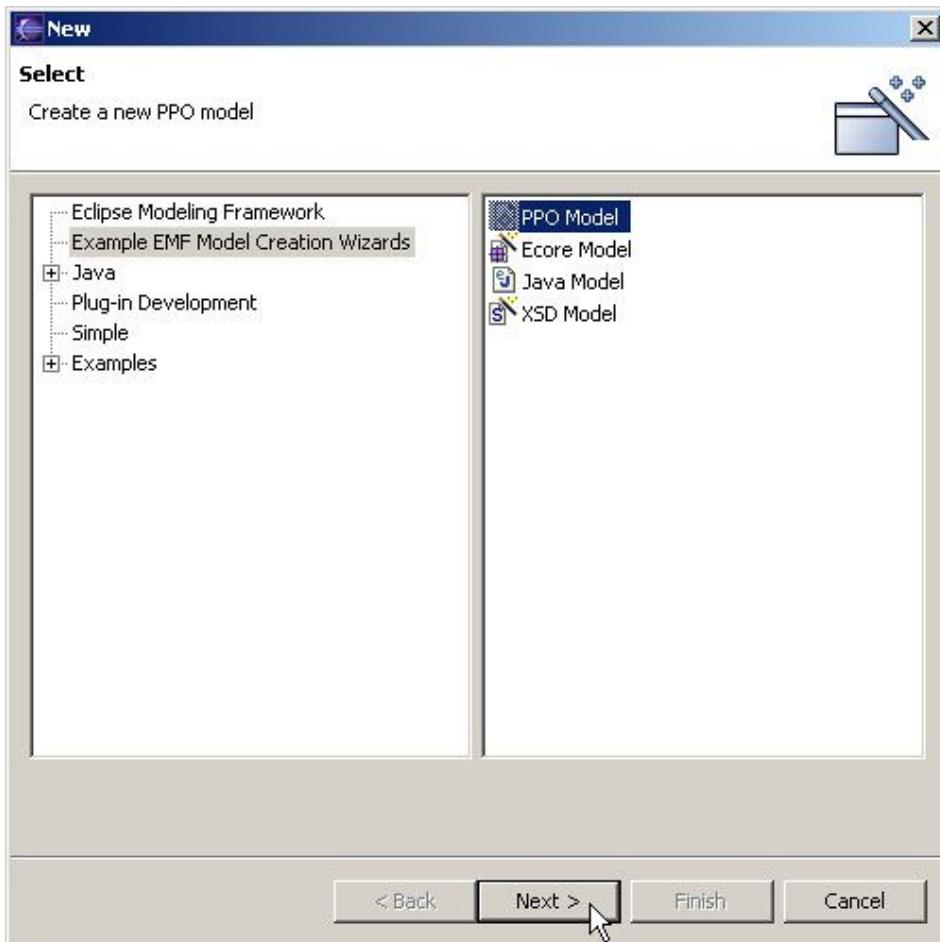


Figure 4.22. The New Wizard.

On the wizard's last page, shown in Figure 4.23, we must select one of the classes defined by the metamodel—the core model from which we generated the application—to use as the type of the model object. This will correspond to the document element of the model's serialized form. The logical choice is **PurchaseOrder**. Select it and click **Finish**.

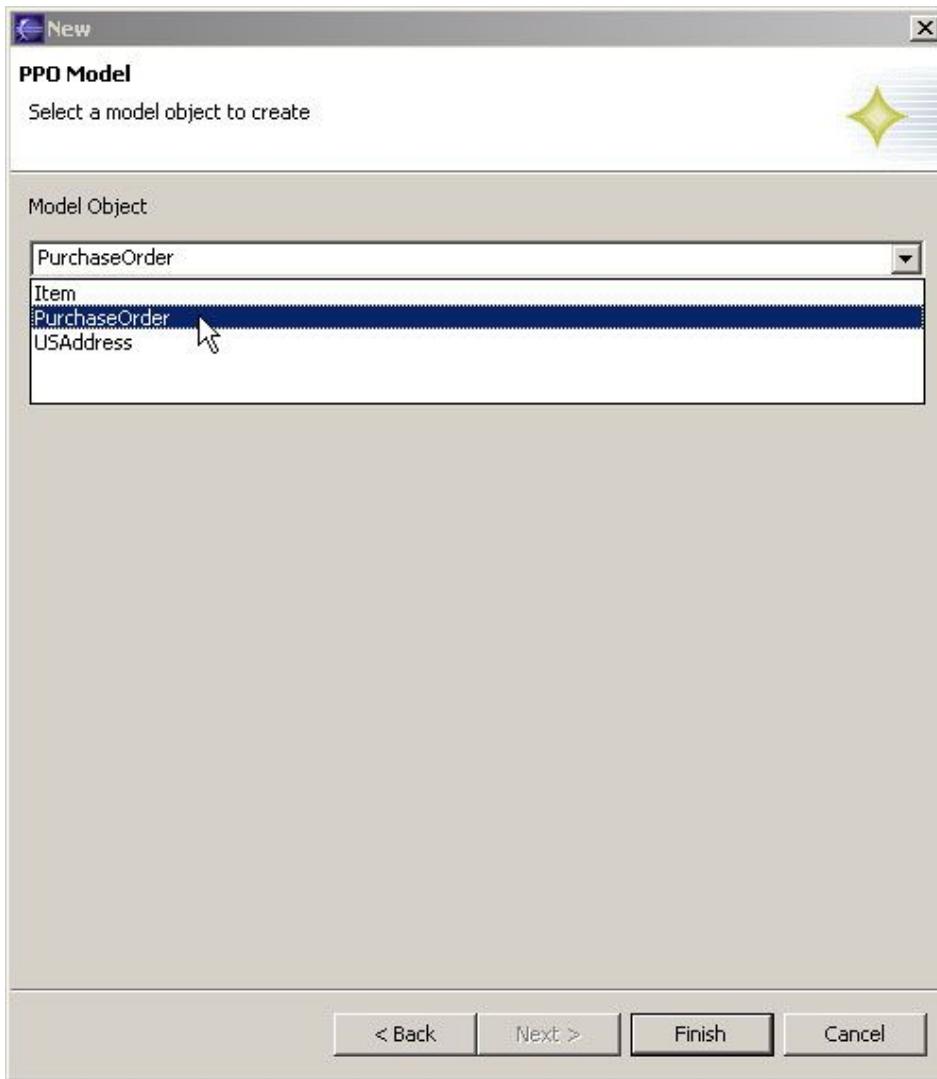


Figure 4.23. Selecting the model object.

The model is created and opened in the PPO Editor. Like the EMF generator, the editor presents us with a tree view of the model.⁸ The root-level element corresponds to the resource. If we expand it, we see its single child, the **PurchaseOrder**-typed model element. We can add children and siblings to model objects via the pop-up menu, shown in Figure 4.24, or the **PPO Editor** pull-down menu. Notice that only valid options, as determined by the class' containment references, are presented. The model objects can also be cut, copied, and pasted, as well as dragged and dropped. Also, their attributes can be edited in the Properties view. Finally, notice the editor's different pages and the content that it provides to the Outline view, all of which was described in Section 3.4.2. Feel free to experiment with the editor to get a feel for all of the functionality it provides.

⁸As we mentioned in Chapter 2, the generator is actually a generated editor itself, though it has been modified.

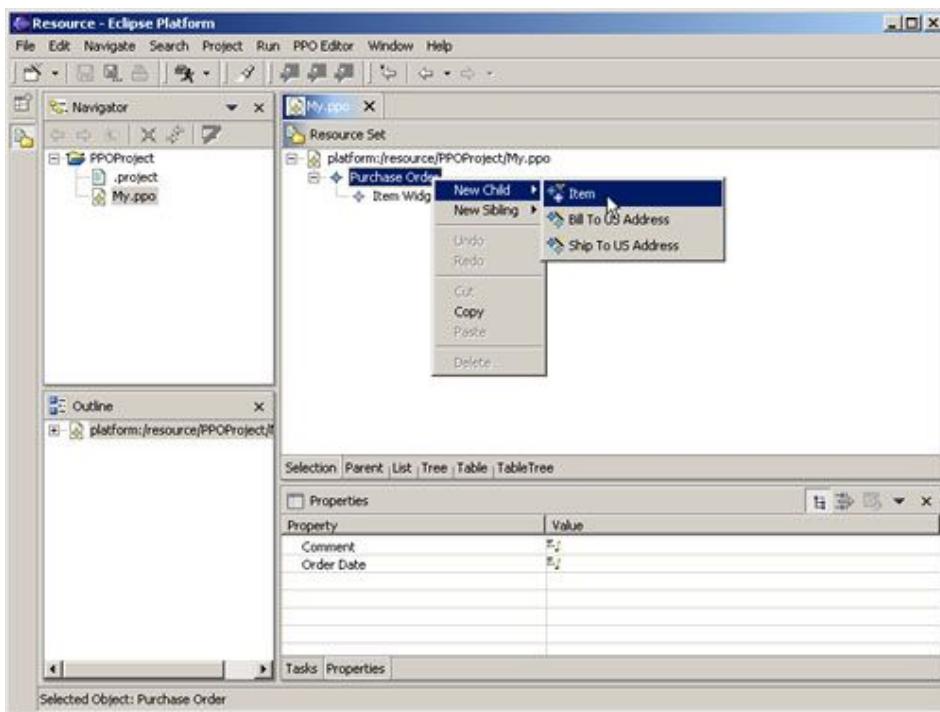


Figure 4.24. Adding objects in the generated editor.

When you are finished, save the model from the **File** menu or the toolbar. Close the editor. If you right-click on the model, *My.ppo*, in the Navigator view, you can select **Open With** and then **Text Editor** from the pop-up menu. This allows you to see the format of the XMI model serialization.

Close the text editor and the runtime workbench.

Continuing Development

We have now seen exactly how EMF lets us take a data model, convert it to Ecore, generate code for model and editor plug-ins, and use those plug-ins to create and edit instances of the model. However, for a real application, that's just the starting point.

From here, we would want to continue to develop our model and editor in two different ways:

1. By writing new Java code and modifying the generated code
2. By updating the original data model, regenerating code based on those changes, and merging the new code into the existing code base

EMF-generated code is meant to be modified. As explained in Section 2.4.3, code merging is done according to @generated Javadoc tags; it is the presence or absence of such tags that determines whether the associated code elements should be updated or left alone during regeneration.

However, we still need to know how to update the models from which we are generating. In general, we edit the data model in whatever form we started with, and then update the Ecore form with those changes. If the changes to the core model are structural, the corresponding changes will be made to the generator model at the same time, while maintaining its existing attribute values.⁹

⁹Note that the generator model automatically adapts to changes in its associated core model when it is opened in the generator. Thus, if we change the core model directly, we do not need to re-import it just to update the generator model.

To perform this update of the models, we right-click the generator model in the Package Explorer or Navigator view and select **Reload** from the pop-up menu, as shown in Figure 4.25.

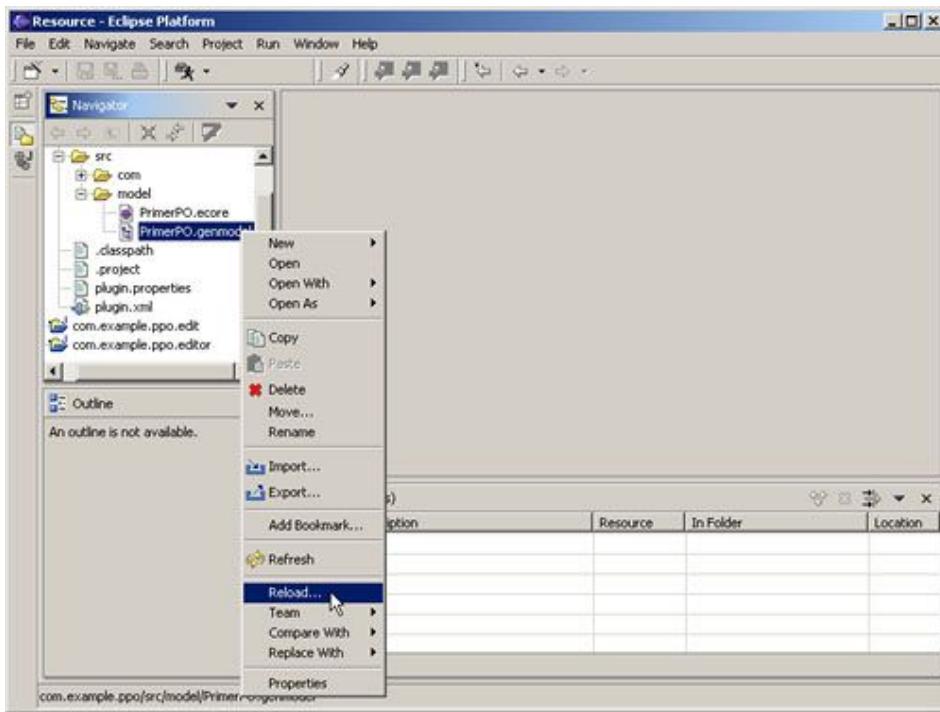


Figure 4.25. Reloading EMF models.

We then step through the wizard, as we did when we created the models originally, reviewing the selections made at that time.

In this chapter, we have discussed how to use the EMF tools to create core and generator models, to generate model and editor implementations from those models, and to continue the development of an application through model and code modifications. Together with the overview of modeling and EMF concepts and the introduction to EMF.Edit presented in the previous two chapters, this equips you well to begin working with these frameworks.

That said, there are still many details that we have yet to discuss. In the next part of the book, we will closely examine the Ecore model at the heart of EMF and its relationships with the other EMF-related technologies: Java, XML, and UML.

Part II. Defining EMF Models

Chapter 5. Ecore Modeling Concepts

When we create a core model—whether directly or from annotated Java, UML, or XML Schema—we are defining the structure for a series of instances of that model. That is, we are specifying the types of objects that make up instance models, the data they contain, and the relationships between them. For example, the PrimerPO model of the previous chapter defined the structure for models that represent purchase orders. Modelers use the term “metamodel” for this kind of model. In a sense, metamodels define a language that their instance models use to describe other things.

As we first discussed in Chapter 2, the metamodel for core models is called Ecore. That is, Ecore defines the structure of core models, which in turn define the structure of the models we use to maintain application data, such as purchase orders. Thus, Ecore is the meta-metamodel for our purchase order instance models.

Most of the modeling concepts that Ecore defines should be quite familiar to avid modelers and object-oriented programmers alike. As we discussed in Chapter 2, Ecore has its roots in MOF and UML, and was designed to map cleanly to Java implementations. Essentially, if you’re comfortable thinking about classes and the relationships between them, you shouldn’t have too much trouble using Ecore to define core models.

Ecore supports a number of higher-level concepts that are not directly included in Java. For example, core models can include containment, bidirectional relationships, and enumeration data types. You are likely familiar with these concepts and have probably implemented them yourself more than once. Part of EMF’s value is its ability to generate correct and efficient Java implementations for these constructs, saving the programmer time and effort.

Ecore is, itself, a core model. That is, Ecore acts as its own metamodel, so it is defined in terms of itself. This presents a few conceptual and implementation-related challenges, but it’s generally considered a good thing. The other alternatives—defining yet another model to act as its metamodel or “hand-waving” through an informal definition—are significantly less attractive. Moreover, we are able to treat Ecore much like any other EMF model and to reap the benefits of the EMF generator in creating and maintaining its implementation.

In this chapter, we will examine Ecore in detail, with the aim of enabling you to use it effectively in defining your own core models. The use of Ecore as its own metamodel inevitably influences such a discussion, as aspects of certain model elements will be described in terms of other model elements. Thus, we begin the discussion with a simplified subset of the model that you may recognize from Chapter 2. Once we understand that subset in its entirety, we can build on that understanding a complete description of the details of Ecore.

Core Model Uses

Before delving into the discussion of Ecore, a word about how EMF uses core models would be appropriate. Like any other model in EMF, a core model can be built programmatically or loaded from a serialized form. It is generally used in two different contexts: during application development and when the application is running.

During development, the core model is the primary source of information for the EMF generator, when it produces code to be used in the application. As we discussed in Chapter 2, this code includes the interfaces and classes that realize the modeled types, a factory for instantiating them, and a package that efficiently builds the core model at runtime and provides convenient access to its members. The EMF generator reads core models from their XMI serializations.

At runtime, the core model is used by generic framework code to determine correct behavior for that particular model, and is likewise available to user-written code that needs to dynamically discover particulars of the model. The framework code that depends upon the core model handles not only such peripheral functionality as serialization, but also includes some of the basic Ecore functionality expressed in the `EObject` API. In fact, in the absence of generated code, the behavior of all dynamic model objects is completely dependent on a core model. At runtime, the core model can either be built programmatically or loaded from a serialization; however, the former, more efficient approach is always taken for generated models.

In discussing the details of Ecore, we will see that most of the concepts that it defines are equally applicable in both the code generation and runtime contexts. Some, however, have special significance in one context or the other, which we will point out as appropriate.

The Ecore Kernel

In Section 2.3.1, we presented an illustration of a simplified subset of the Ecore model. We describe this, with minor additions, as the Ecore kernel, and we will use it to “bootstrap”¹ the discussion of the full model. This model is illustrated in Figure 5.1.

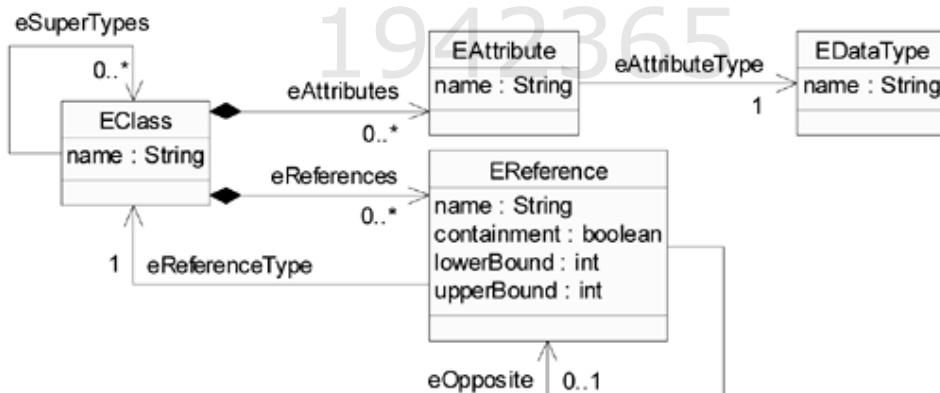


Figure 5.1. The Ecore kernel.

Note that we have created this kernel only for the purposes of this discussion. There is nothing particularly special about how these elements are defined or created in the implementation of Ecore.

Essentially, reviewing from Chapter 2, this model defines four types of objects—that is, four classes:

1. **EClass** models classes themselves. Classes are identified by name and can contain a number of attributes and references. To support inheritance, a class can refer to a number of other classes as its supertypes.
2. **EAttribute** models attributes, the components of an object's data. They are identified by name, and they have a type.

¹The term *bootstrap* is commonly used to describe the process by which a system starts up, which generally involves loading a small portion of the system in order to support the loading and initialization of its remainder. It should bring to mind the awkward image of a person attempting to stand by pulling up on his or her own boots. This is all a bit of a tangent; don't worry about it too much.

3. **EDataType** models the types of attributes, representing primitive and object data types that are defined in Java, but not in EMF. Data types are also identified by name.
4. **EReference** is used in modeling associations between classes; it models one end of such an association. Like attributes, references are identified by name and have a type. However, this type must be the **EClass** at the other end of the association. If the association is navigable in the opposite direction, there will be another corresponding reference. A reference specifies lower and upper bounds on its multiplicity. Finally, a reference can be used to represent a stronger type of association, called containment; the reference specifies whether to enforce containment semantics.

Notice that this model really needs to be understood as a single unit, as it is highly self-referential. In describing **EClass**, we described its attributes, which are modeled using **EAttribute**, and its references, modeled with **EReference**, even referring indirectly to the attributes of **EReference**. Fortunately, the concepts expressed in this model should be quite familiar to modelers and object-oriented programmers, so this probably wasn't too troublesome. Now, with this subset of Ecore in hand, we can tackle the rest of the model.

Structural Features

Looking back at the Ecore kernel, you may notice a number of similarities between **EAttribute** and **EReference**: they both have names and types, and taken together, they define the state of an instance of the **EClass** that contains them. There are many more common aspects of these two classes, including, in fact, the **lowerBound** and **upperBound** attributes, which we previously showed only for **EReference**. To capture these similarities, Ecore includes a common base for these two classes, called **EStructuralFeature**. The situation is illustrated in Figure 5.2.

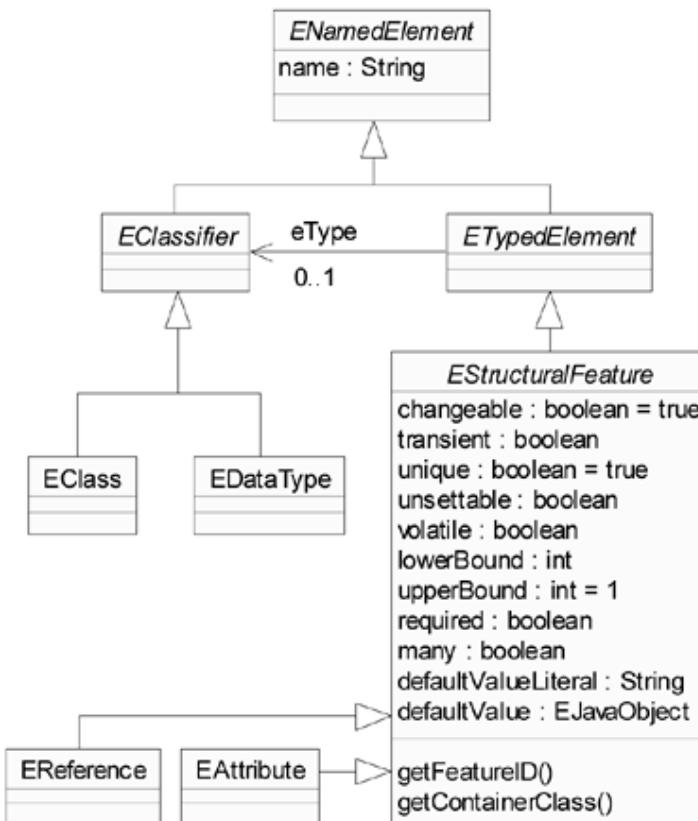


Figure 5.2. Ecore structural features.

As the figure shows, **EStructuralFeature** is, itself, derived from other supertypes. **ENamedElement** defines just one attribute, the **name** that we have seen in every class discussed so far. Most classes in Ecore extend this class in order to inherit this attribute.

Another common aspect of **EAttribute** and **EReference** that we observed is the notion of a type. Because this is also shared with other classes in Ecore, as we will soon see, the **eType** attribute is factored out into **ETypedElement**, the immediate supertype of **EStructuralFeature**. Notice that the type of **eType** is **EClassifier**, a common base class of **EDatatype** and **EClass**, which were the required types for **eAttributeType** and **eFeatureType**, respectively.

EStructuralFeature includes a number of attributes used to characterize both attributes and references. Five boolean attributes define how the structural feature stores and accesses values:

- **changeable** determines whether the value of the feature may be externally set.
- **transient** determines whether the feature is omitted from the serialization of the object to which it belongs.
- **unique**, which is only meaningful for multiplicity-many features, specifies whether a single value is prevented from occurring more than once in the feature.
- **unsettable** specifies whether the feature has an additional possible value, called *unset*, that is unique from any of its type's legal values, including `null` for an object type. The value of this attribute also determines the semantics of `EObject`'s `eUnset()` and `eIsSet()` reflective APIs: these methods change an unsettable feature's value to that unset value and test whether it is set to some other value, respectively. But, for a non-unsettable feature, they reset the feature's value to its default and test that it is set to a non-default value, respectively.

- **volatile** specifies whether the feature has no storage directly associated with it; this is generally the case when the feature's value is derived purely from the values of other features. The bodies of the accessors for such features must be coded by hand in Java.

Before continuing, it is worth enlarging slightly on the notion of structural features whose values are derived from those of other features. Since we declare such features as volatile, they contribute nothing to the state of an object and, accordingly, they have nothing to offer to the serialized form. Hence, we usually declare them to be transient, as well. Moreover, to keep things simple, we generally do not want their values to be externally set, so we also declare them to be non-changeable. In fact, throughout this discussion of Ecore, we will use the term *derived* to imply that a structural feature is volatile, transient, and non-changeable. Keep in mind that you are never responsible for determining appropriate values for derived structural features; we describe them only so you can use them to programmatically inspect a core model.

There are four attributes of **EStructuralFeature** that relate to multiplicity, or the number of values associated with the feature. The minimum and maximum number of allowed values are specified by **lowerBound** and **upperBound**, respectively. Although these are defined in the model as integers, only 0 and the positive integers are legal values. In addition, **upperBound** should be greater than or equal to **lowerBound**, or it can be *unbounded*. This latter condition is indicated by in the model by "*" and corresponds to the value of -1, which is represented in the Java implementation of Ecore by the constant `EstructuralFeature.UNBOUNDED_MULTIPLICITY`. Two more derived boolean attributes, **required** and **many**, are provided as conveniences; they reflect whether the lower and upper bounds are non-zero and greater than one, respectively.

The last two attributes of **EStructuralFeature** have to do with the default value, the value of the feature before it is explicitly set to something else. Regardless of the type of the feature, its default value is always modeled by **defaultValueLiteral** as a string. Another attribute, **defaultValue**, can be used to access the equivalent value as a Java object of the appropriate type for the feature. This attribute is derived from **defaultValueLiteral** using the standard Ecore data type conversion mechanism, which we will discuss in Section 5.6. For a reference, **defaultValue** will always be `null`. Although **defaultValue** is modeled as non-changeable, there is a hand-coded convenience method in the Java implementation, `EstructuralFeature.setDefault()`, which takes an object and uses the inverse of the data type conversion mechanism to set **defaultValueLiteral** to the equivalent string.

If a structural feature does not specify a non-null **defaultValueLiteral**, then the intrinsic default value of its type is used instead. As we will see in Section 5.5, **EClassifier** includes a non-changeable **defaultValue** attribute that corresponds directly to the default value of its underlying Java type. Thus, boolean-typed attributes with no default specified, such as **transient**, **unsettable**, and **volatile**, have a default value of `false`. Likewise, as an integer-typed attribute with no explicit default, **lowerBound** has a default value of 0.

Finally, we should mention the operations defined on **EStructuralFeature**: **getFeatureID** and **getContainerClass**. We will not worry about how operations are represented in Ecore until Section 5.4, so for now, we'll just say what these two operations do.

Each structural feature is identified within a class by a unique integer, called the feature ID, which is used in implementing the reflective **EObject** API. This value is set during the initialization of a static model and during a call to `getAllStructuralFeatures()` on a dynamically defined class. Before that point, it will just be -1.

In generated models, the Java class that realizes the containing class is also significant. Where there is multiple inheritance, the feature ID should be considered relative to a particular class that defines or inherits it, and the Java class is used as a key in converting among these values. The class of the structural feature's container identifies the generated, static base value, from which we can efficiently convert to a value relative to any given subtype. This is used in the fast, switch-based, generated implementations that we discuss in Section 9.6. The feature ID and container class are made available by the `getFeatureID` and `getContainerClass` operations, respectively.

Attributes

Having looked at what attributes have in common with references, we will now examine what sets them apart. Figure 5.3 illustrates the unique aspects of `EAttribute`.

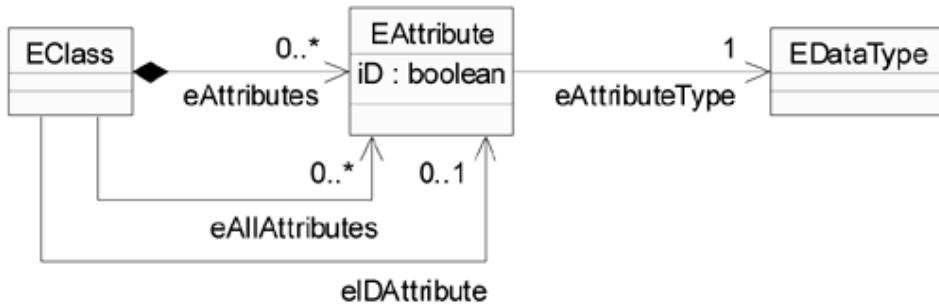


Figure 5.3. Ecore attributes.

In addition to all of the attributes it inherits from `EStructuralFeature`, `EAttribute` defines one more, `iD`,² which determines whether the value of the attribute can be used to uniquely identify the instance of its containing class in a model.

`EAttribute` also defines one derived reference: `eAttributeType`, which actually refers to the same `EClassifier` as `eType`, which we described in the previous subsection. However, the type of an attribute must be a data type and not a class, so this reference casts that object to an `EDataType`.

As we saw in Figure 5.1, `EAttributes` are contained by an `EClass` via its `eAttributes` reference. `EClass` defines two additional related references, `eAllAttributes` and `eIDAtribute`. For an instance of `EClass`, `eAllAttributes` includes not only its own `eAttributes`, but also those collected from the `eAttributes` references of all of its supertypes—that is, the `EClasses` accessible recursively through the `eSuperTypes` reference. As a convenience, `eIDAtribute` refers to the first `EAttribute` in `eAllAttributes` for which `iD` is `true`. Both `eAllAttributes` and `eIDAtribute` are derived references.

References

The unique aspects of `EReference` are illustrated in Figure 5.4.

²The unfortunate capitalization comes from the convention that structural feature names should begin with a lowercase letter. Fortunately, when code is generated for the model, this letter is capitalized in forming the accessor names `isID()` and `setID()`.

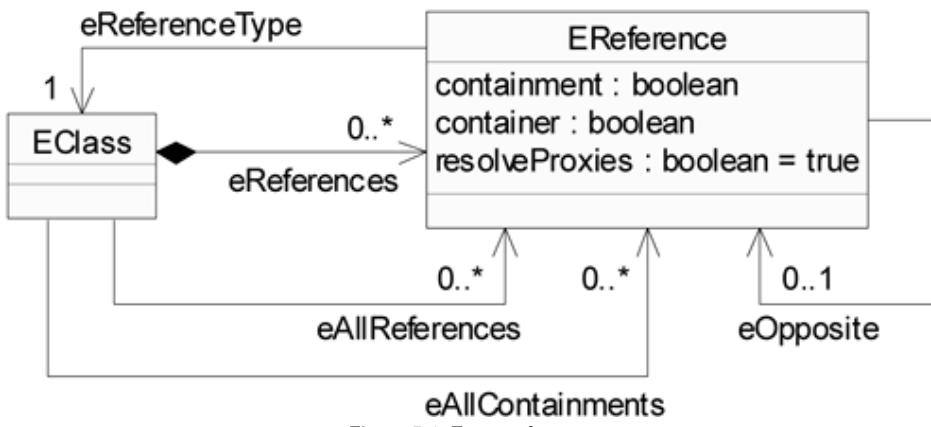


Figure 5.4. Ecore references.

EReference adds two references and three attributes to those defined by **EStructuralFeature**. The first reference, **eReferenceType**, is analogous to **EAttribute**'s **eAttributeType**: it refers to the same **EClassifier** as **EStructuralFeature**'s **eType**, but cast to an **EClass**. Like its attribute analog, it is a derived reference. The other, **eOpposite**, refers to the reference representing the opposite direction of a bidirectional association. Thus, such an association is represented by the two **ERefferences**, each defining the other as its **eOpposite**.

We mentioned the notion of containment in Section 5.2. Known as “by-value aggregation” in UML, containment is a stronger type of association that implies a whole-part relationship: an object cannot, directly or indirectly, contain its own container; it can have no more than one container; and its lifespan ends with that of its container. The **containment** attribute indicates whether an **EReference** is being used in modeling such a relationship; it is set on the reference from the whole to the part, that is, from the container to the contained object. If the containment relationship is explicitly bidirectional, then for the opposite reference, the derived **container** attribute is also `true`.

The last attribute defined by **EReference**, **resolveProxies**, relates to EMF’s resource model for persistence. We described in Section 2.5.2 how, when a resource is loaded, any referenced objects that are persisted in other resources are represented by proxies. We also said that, when such an object is accessed for the first time, its resource is loaded and the real object is returned. In fact, the automatic resolution of proxies is only performed for references that define **resolveProxies** to be `true`. This is the default and should normally be the case; however, sometimes the structure of a model can prevent a reference from ever being cross-document. Generally, such cases stem from the restriction on containment references that they may only exist between objects in the same resource. In these cases, **resolveProxies** can be `false`, permitting the use of a more efficient implementation. We will see an example of such a reference within Ecore in Section 5.5.1.

EReference is also analogous to **EAttribute** in its containment relationship with **EClass**: an **EClass** contains **ERefferences** via its **eReferences** reference, and defines **eAllReferences** to also include the references of its supertypes. Finally, **eAllContainments** refers to all of those **ERefferences** accessible via **eAllReferences** for which **containment** is `true`. Both **eAllReferences** and **eAllContainments** are derived references.

Behavioral Features

In addition to their structural features, Ecore can model the behavioral features of an **EClass** as **EOperations**. Actually, it’s really only modeling the interfaces to those operations; a core model gives no further clue as to the actual behavior that operations exhibit. The bodies of operations must be coded by hand in the generated Java classes. **EOperation** is illustrated, along with the closely related **EParameter**, in Figure 5.5.

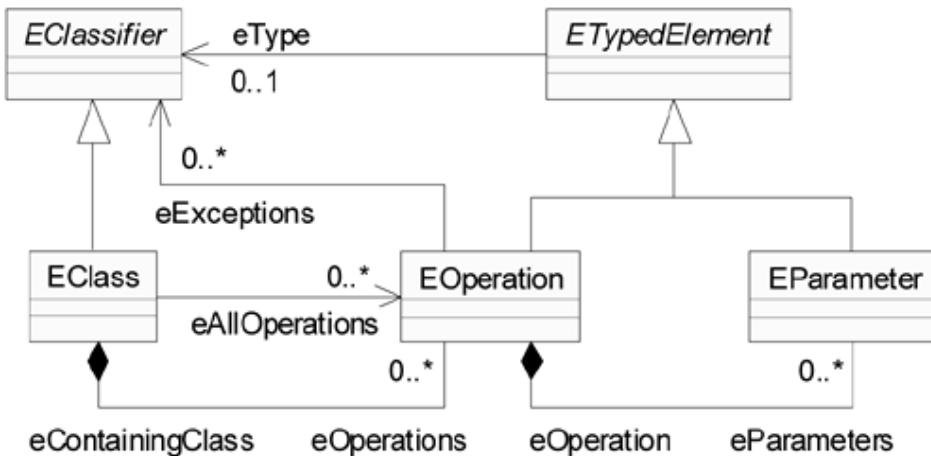


Figure 5.5. Ecore operations and parameters.

Notice that **EOperation**'s relationship with **EClass** is quite similar to those of **EAttribute** and **EReference**: **EOperations** are contained by an **EClass** via the **eOperations** reference, and a derived **eAllOperations** reference is defined to include the operations of a class and its supertypes. Notice, however, that **eOperations** is part of a bidirectional association, which allows an **EOperation** to easily obtain the **EClass** that contains it via the opposite reference, **eContainingClass**. This is the first bidirectional association that we have seen in Ecore, which may lead you to wonder why **eAttributes** and **eReferences** didn't have opposites. In fact, they have an equivalent reference, which we will discuss in the next section.

An **EOperation** contains zero or more **EParameters**, accessible via **eParameters**, which model the operation's input parameters. Again, this reference constitutes half of a bidirectional association; the **EParameters** can access the **EOperation** to which they belong via **eOperation**.

Both **EOperation** and **EParameter** inherit the **name** attribute and **eType** reference from **ETypedElement**. These **eType** references model the operation's return type and the input parameter type, respectively, and can refer to any **EClassifier**, whether **EClass** or **EDataType**.

Finally, **EOperation** defines an additional reference, **eExceptions**, to zero or more **EClassifiers**, in order to model the types of objects that an operation can throw as exceptions.

Classifiers

Having discussed Ecore's representations of structural and behavioral features, let's now back up and take a detailed look at the classes that, taken together, they define. As we have already seen, **EClass** shares a base class with **EDataType**, so we will have to discuss the two together. That base class, **EClassifier**, acts as a common target for **ETypedElement**'s **eType** reference, allowing structural features, operations, and parameters to specify as their types either classes or data types. As illustrated in Figure 5.6, **EClassifier** also contributes a number of attributes and operations to its subtypes, which we will now discuss.

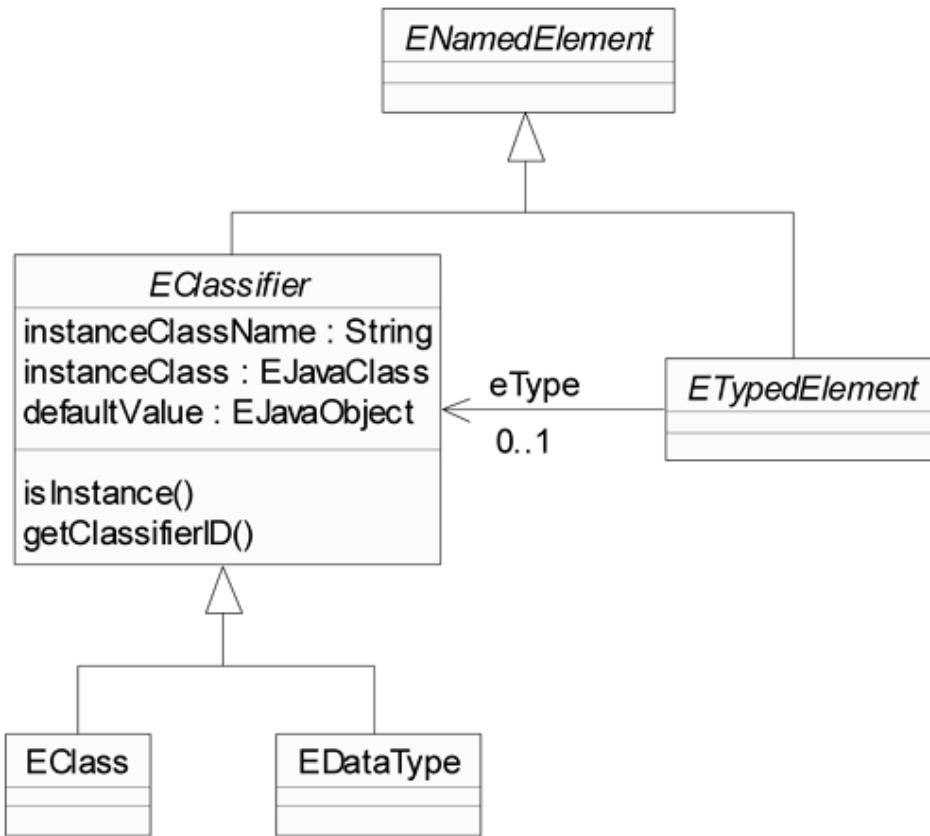


Figure 5.6. Ecore classifiers.

First, we observe that, like everything else we have seen so far, **EClassifier** inherits a **name** attribute from **ENamedElement**.

Two of the attributes defined by **EClassifier** specify a correspondence to an ordinary, non-EMF Java class or interface; we will expand on the particular character of this correspondence for **EClasses** and for **EDataTypes** in the following subsections. Such a class or interface is specified, by name, by the **instanceClassName** attribute. A derived attribute, **instanceClass** can be used to access the Java class with that name. The relationship between these two attributes is similar to the one between **EStructuralFeature**'s **defaultValueLiteral** and **defaultValue** attributes, described in Section 5.3, in that the Java implementation also provides a hand-coded `setInstanceClass()` method that takes a Java class, but actually sets the value of **instanceClassName**.

The last of **EClassifier**'s attributes, **defaultValue**, represents the intrinsic default value of a class or data type—that is, the default value of the Java primitive or class that realizes it. This attribute is derived from the value of **instanceClass**.

EClassifier provides an **isInstance** operation, which tests whether an arbitrary Java object, specified as a parameter, is non-null and an instance of the class or data type represented by the **EClassifier**, or of one of its subtypes. For an **EClass**, this is determined by consulting the **eAllSuperTypes** reference, which we will see in the following subsection. For an **EDataType**, it means a reflective Java `Class.isInstance()` test on the value of **instanceClass**. Finally, the **getClassifierID** operation returns an integer that is used to uniquely identify a classifier. This value is set during the initialization of a static model; it will always be `-1` for dynamically defined classifiers.

Classes

The unique aspects of **EClass** are illustrated in Figure 5.7.

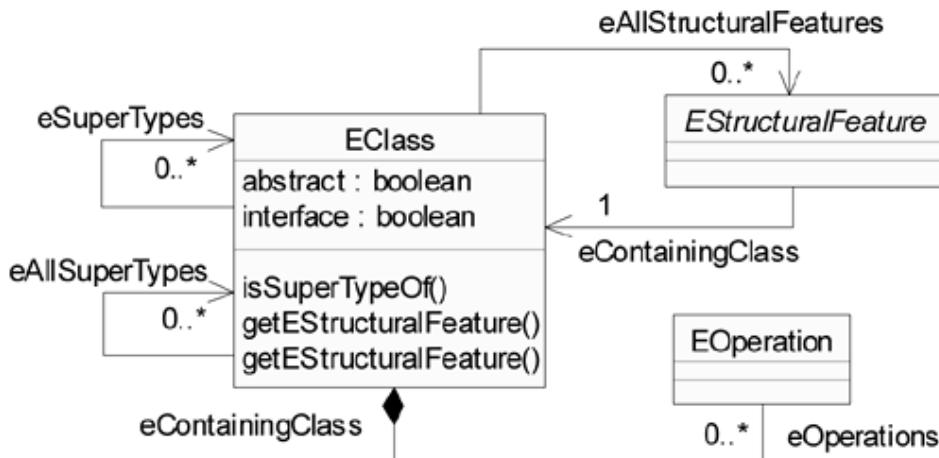


Figure 5.7. Ecore classes.

We have already discussed at some length the relationships between **EClass** and the classes that represent structural and behavioral features; let's complete that discussion now. We have seen the **eOperations** containment reference and its opposite, **eContainingClass**, which connect an **EClass** with its **EOperations**. We have also seen **eAttributes** and **eReferences**, which are not bidirectional but similarly connect an **EClass** to its **EAttributes** and **EReferences**. We now introduce the **eContainingClass** feature of **EStructuralFeature**, which acts like a single opposite for **eAttributes** and **eReferences**; it simply refers to a feature's container, casted as an **EClass**. In fact, casting the object's container is exactly how EMF usually implements the opposite for a containment reference. However, we have modeled this reference separately and explicitly so that it may be implemented once in **EStructuralFeature**, rather than in both **EAttribute** and **EReference**, the actual targets of the containment references. This is a derived reference; since it is just the opposite of two containment relationships, it is guaranteed not to cross document boundaries and can thus also be declared as non-proxy resolving.

EClass also defines a reference called **eAllStructuralFeatures** that conveniently aggregates the objects accessible via **eAllAttributes** and **eAllReferences**—that is, it includes all the structural features defined and inherited by a class.

We saw in Section 5.2 that **EClass** provides support for multiple inheritance via its **eSuperTypes** reference. A class inherits structural and behavioral features from all of its supertypes, all of their supertypes, and so on. This complete set of classes from which features are inherited is provided by the derived **eAllSuperTypes** reference. Notice that, since **eSuperTypes** is not a containment reference, a class can inherit from base classes defined in other Ecore documents.

The two attributes defined by **EClass** itself can be used to specify the particular type of class being modeled. If **interface** is `true`, the **EClass** represents an interface that declares its operations and the accessors for its attributes and references, but can provide no implementation for them. An interface cannot be instantiated. Implementation of an interface is modeled by including the interface in the **eSuperTypes** reference of a non-interface **EClass**; that class will implement the operations and the accessors for the structural features declared by the interface.³ If **abstract** is `true`, the

³Of course, since Ecore includes no way to specify the behavior of operations or volatile structural feature accessors, the implementations of such methods must be completed by hand.

EClass represents an abstract class, from which other classes can inherit features, but which cannot itself be instantiated. Ecore includes a number of abstract classes, most of which we have already seen: **ENamedElement**, **EClassifier**, **ETypedElement**, and **EStructuralFeature**. We will look at the only other one, **EModelElement**, in Section 5.6.

As mentioned previously, the **instanceClassName** and **instanceClass** attributes defined by **EClassifier** establish a connection to a non-EMF Java class or interface. In the case of an **EClass**, they specify the interface that corresponds to the class it defines. This is only applicable in the context of generated models, as there is no such interface for classes defined dynamically.

Finally, **EClass** defines three operations that provide simple conveniences. As a shortcut for consulting the **eAllSuperTypes** reference, **isSuperTypeOf** tests whether one class extends another. The two forms of **getEStructuralFeature** return the **EStructuralFeature** from **eAllStructuralFeatures** with either a specified feature ID or name.

Data Types

While classes define multiple structural and behavioral features, data types represent a single piece of “simple” data. This distinction is somewhat similar to, though not the same as, that between classes and primitive types in Java. In fact, an **EDataType** models a single Java type, whether a primitive type, a Java class or interface, or even an array. This allows conceptually simple classes, like `String`, to be represented as a single unit.

In general, however, it is considered a bad idea to represent a highly complicated Java class as a data type. Instead, it is preferable to directly model the class in EMF with an **EClass** and, thus, to benefit from the framework’s assistance in matters of serialization, notification and reflection. As a general rule, if its values cannot be simply represented as a string without the use of some notation to impose structure, it probably shouldn’t be modeled as a data type.⁴

It is the **instanceClassName** and **instanceClass** attributes inherited from **EClassifier** that are used to specify the Java type that is modeled by a data type. **EDataType** does not add much to the attributes and operations that it inherits. In fact, it defines just one attribute, **serializable**, which indicates whether values of the type may be serialized. Setting it to `false` has the same effect on serialization, at runtime, as making all attributes using the type transient. The difference between these two approaches is seen at development time: a generated factory includes methods to convert data types to and from strings, but only if they are serializable.

Enumerated Types

An enumerated type is a special data type that is defined by an explicit list of values that it may possibly take, called literals. Figure 5.8 illustrates **EDataType**, **EEnum**, and **EEnumLiteral**, which model data types, enumerated types, and literals, respectively.

⁴In fact, this is more than just a rule of thumb: serialization of data type values is implemented at a core model, not resource, level. Thus, in order to be compatible with different resource types that perform different serializations, it should never impose structure.

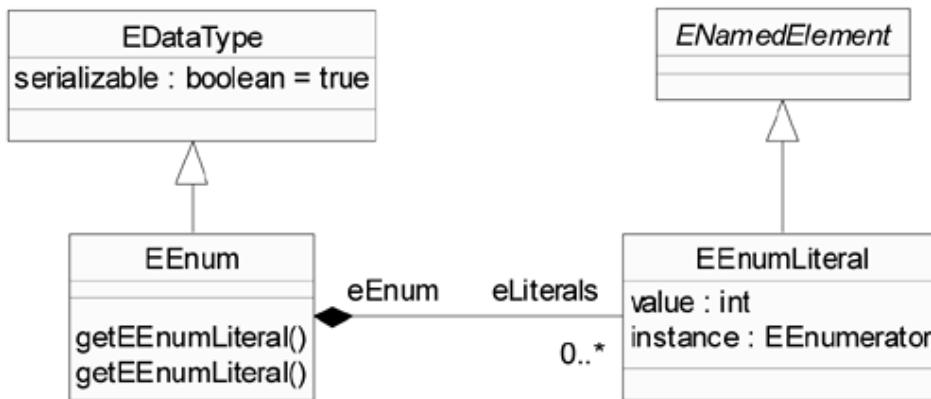


Figure 5.8. Ecore data types, enumerated types, and literals.

An **EEnum** specifies zero or more valid **EEnumLiteral**s via the `eLiterals` containment reference, which is bidirectional, with an opposite named `eEnum`. **EEnumLiteral** inherits a **name** attribute from **ENamedElement** and defines its own integer-typed `value` attribute. In generated models, enumerated types are realized using the Type-safe Enum pattern, described by Joshua Bloch in *Effective Java Programming Language Guide* [6]. In this pattern, a Java class with no public constructor is defined for the enumerated type; publicly available, static instances of it represent its literal values. **EEnumLiteral**'s `instance` attribute is used to specify the static object that represents a literal.

For convenience, **EEnum** defines two `getEEnumLiteral` operations, to return the **EEnumLiteral** from `eLiterals` with either a specified name or value.

Packages and Factories

In Ecore, related classes and data types are grouped into packages, and a factory is used to create instances of the classes and values of the data types that belong to the package. When a core model is serialized, the document-level element represents a package. **EPackage** and **EFactory**, which model packages and factories, are illustrated in Figure 5.9.

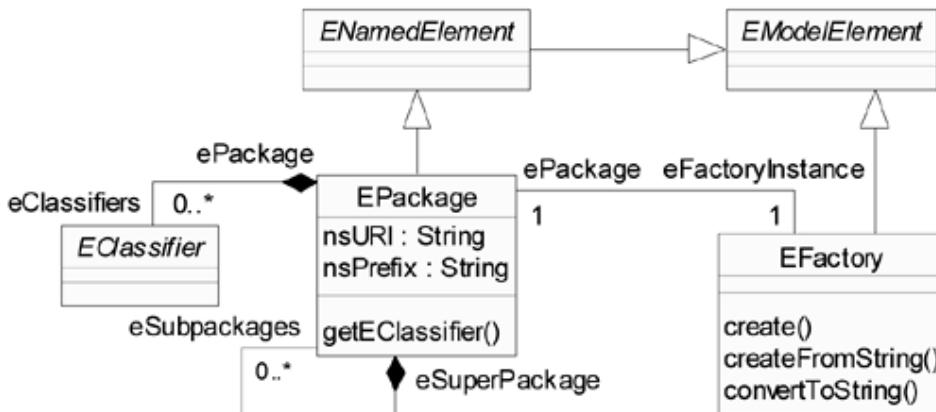


Figure 5.9. Ecore packages and factories.

EPackage inherits a **name** attribute from **ENamedElement**. A package's name need not be universally unique. Instead, a URI is used to uniquely identify the package. Reflecting EMF's close relationship with XML, this URI is also used in the serialization of instance documents to identify an XML namespace⁵ for the document-level element. This URI is given as the `nsURI` attribute of the **EPackage**; the `nsPrefix` attribute is used to specify the corresponding namespace prefix.

The **EClassifiers** grouped by an **EPackage** are contained via the **eClassifiers** reference, which has an opposite reference called **ePackage**. A **getEClassifier** operation is also defined, to conveniently obtain one of these **EClassifiers** by name. **EPackages** support nesting, via the **eSubpackages/eSuperPackage** bidirectional, containment reference. An **EPackage**'s associated **EFactory** is accessed via the **eFactoryInstance** reference.

Ecore packages are closely related to Java packages. Conceptually, they serve the same purpose, and when code is generated from a core model, the Java classes and interfaces that realize its classifiers are placed in packages that correspond to their Ecore counterparts. When Ecore packages are nested, the package names used in generated code are automatically formed, in the usual Java manner, by prepending their dot-separated superpackage names. Note, however, that it is not necessary to model empty, nested packages just to ensure that generated code uses universally unique package names. **GenPackage**, the generator model class that decorates **EPackage** with code generation data, provides an attribute called **basePackage** for this purpose, which we discuss in Chapter 11.

While all of the classes in Ecore that we have seen so far are subtypes of **ENamedElement**, **EFactory** directly extends **EModelElement**, the single base-type for everything in Ecore. As a result, **EFactory** does not have a **name** attribute. In fact, **EFactory** does not declare or inherit any structural features except for **ePackage**, the reverse of the reference from the associated **EPackage**. Since factories do not have any further instance data, they need not be explicitly modeled or serialized; a factory can simply be demand-created by its package. As a result, both of the references that model the association between **EPackage** and **EFactory** are transient.

The factory's role is purely behavioral: it defines **create**, **createFromString**, and **convertToString** operations, which instantiate classes and convert data type values to and from strings. For generated models, each of these operations is implemented using call-out methods, one for each class or data type. The mappings between data type values and their serialized forms are thus specified by changing the implementations of these **createFromString()** and **convertToString()** methods.

Annotations

Annotations constitute a mechanism by which additional information can be attached to any object in a core model. **EAnnotation**, illustrated in Figure 5.10, is used to model annotations.

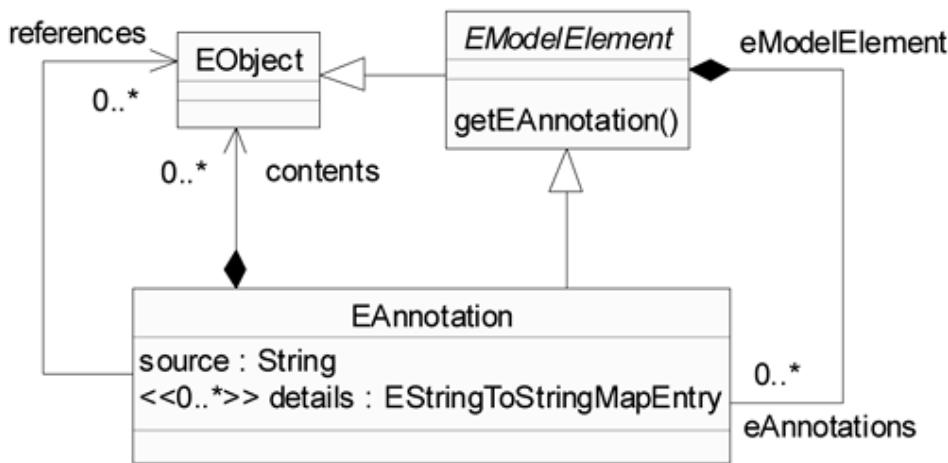


Figure 5.10. Ecore annotations.

⁵The syntax for XML namespaces is defined by the W3C's *Namespaces in XML* recommendation, available at www.w3.org/TR/REC-xml-names.

By subclassing **EModelElement**, all classes defined in Ecore include an **eAnnotations** reference, which can contain zero or more **EAnnotations**. As this association is bidirectional, an **EAnnotation** can access its **EModelElement** via the opposite reference, **eModelElement**.

EAnnotation defines a **source** attribute, which is typically used to store a URI representing the type of the annotation. **EModelElement** also defines a **getEAnnotation** operation, to conveniently obtain one of its **EAnnotations** according to this attribute. Two references, **contents** and **references**, allow an **EAnnotation** to contain or refer to any number of arbitrary EMF objects, which we model using **EObject**.⁶

The last structural feature defined by **EAnnotation** is worth pointing out, as it demonstrates an Ecore modeling technique that we have not yet seen: the use of map-typed features. Although there is no map type explicitly modeled in Ecore, support for such features is enabled through a special behavior of the generator.

Figure 5.11 illustrates **EStringToStringMapEntry**, the type of the **details** feature.

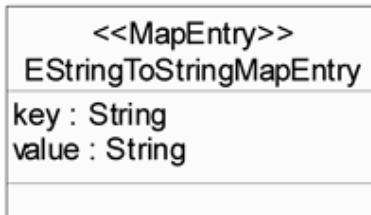


Figure 5.11. Example of a map-entry derived class.

Like any other class, **EStringToStringMapEntry** is modeled using **EClass**. It includes two special attributes, **key** and **value**, the types of which will be used to constrain the types of the map keys and values in the instance model. In general, **key** and **value** can be references, instead, so that their types can be other modeled classes. Because of its **<>MapEntry>** stereotype, the **EClass**'s **instanceClassName** attribute will be set to “java.util.Map\$Entry”, to indicate that the class will implement a `Map.Entry`-based interface.

So, a multivalued containment reference⁷ to a class that implements `Map.Entry` and contains two structural features, **key** and **value**, tips off the generator that the pattern for a map-typed feature should be used. We will discuss this pattern in Chapter 9.

In this case, the end result is a feature called **details** that stores mappings from one string value to another. We will see one important use for annotations in Chapter 7: allowing core models based on XML Schemas to retain information that does not map cleanly to Ecore.

Modeled Data Types

Ecore itself includes a number of **EDataType** instances representing all of the most common Java types, as specified by Table 5.1. Any core models can refer to any of these data types and take advantage of the conversions to and from strings that are provided by Ecore's factory implementation.

Table 5.1. Java Language Types in Ecore

Ecore Data Type	Java Primitive Type or Class	Serializable
EBoolean	boolean	true

⁶Notice from Figure 5.10 that **EModelElement**, the base type for Ecore, is derived from **EObject**. Since an **EObject**'s meta-type is **EClass**, this relationship is consistent with our earlier statement that Ecore is its own metamodel.

⁷Although **details** looks like an attribute in UML, because its type is a class and not a data type, it is actually represented as a reference in Ecore. Refer to Chapter 8 for a more detailed explanation.

<i>Ecore Data Type</i>	<i>Java Primitive Type or Class</i>	<i>Serializable</i>
EByte	byte	true
EChar	char	true
EDouble	double	true
EFloat	float	true
EInt	int	true
ELong	long	true
EShort	short	true
EBooleanObject	java.lang.Boolean	true
EByteObject	java.lang.Byte	true
ECharacterObject	java.lang.Character	true
EDoubleObject	java.lang.Double	true
EFloatObject	java.lang.Float	true
EIntegerObject	java.lang.Integer	true
ELongObject	java.lang.Long	true
EShortObject	java.lang.Short	true
EString	java.lang.String	true
EJavaObject	java.lang.Object	false
EJavaClass	java.lang.Class	true

Ecore also includes a few more **EDataTypes** that allow it to completely model its own API, including **EResource**, **EList**, **ETreeIterator**, and **EEnumerator**. These should not be used in other models.

Chapter 6. Java Source Code

If you are more familiar with Java programming than you are with modeling tools, you may find that the easiest way to describe your model is to use Java code. By providing a few simple Java specifications you can give EMF enough information to derive an entire core model. In this chapter we will explore the Java source code constructs and the special annotations that are needed by EMF in order to derive a core model.

As you have seen in Chapter 5, a core model is made up of one or more **EPackages**, each of which contains a number of **EClasses**, where each **EClass** has some number of **EAttributes**, **EReferences**, and **EOperations**. An **EPackage** may also contain **EDataTypes** and **EEnums**. If you specify your model using Java source code, the **EClasses** are given as Java interfaces. The **EAttributes**, **EReferences**, and **EOperations** will be inferred from the methods that are specified for each interface. **EEnums** are specified as Java classes. **EPackages** may also be specified as Java interfaces, although in most cases they do not need to be specified explicitly because they can be inferred from the package statements that are used to define the **EClasses**. Similarly, **EDataTypes** may be specified explicitly, although they can usually be inferred by references to the corresponding Java types.

EMF requires you to annotate your interfaces with specially formatted comments that are used to identify the model elements, and to optionally provide additional information that is not directly expressed in the Java interfaces. These take the form of Javadoc comments that precede the class, interface, or method definitions to which they apply. They are identified by an `@model` tag and have the following syntax:

```
/**  
 * @model [ property="value" | property='value' ] ...  
 */
```

Each `@model` tag can be followed by several property-value pairs. The allowed specifications for property and value depend on which model construct is being defined. These are discussed in the following sections. Note that the value must be enclosed in either double quotes or single quotes.

The Java source code that you specify does not need to be complete. The code only needs to include enough information to describe your model. So, for example, you need to provide a Java interface for each **EClass**, but you do not need to provide an implementation class. Similarly, you need to provide a `get()` method for each attribute and reference, but you do not need to specify any other accessor methods. (If other accessor methods are needed, they will be generated by EMF as appropriate.)

Java Specification for Packages

Normally, you do not need to specify anything to get an **EPackage** because the **EPackage** can be inferred from the classes and enumerations that appear in the Java package. The only time when you may need to explicitly specify a package is when you wish to override some of the package properties.

The Java specification for an **EPackage** is an interface that is not preceded by an `@model` statement and that has a name that ends with the suffix “Package”.¹

There are various attributes of the **EPackage** that may be specified by including those properties as final static fields in the Java interface. These fields should have type `String` and should have an initializer.

- If a field named `eNAME` is specified, the `name` attribute of the **EPackage** is set to the initializer of the field.
- If a field named `eNS_URI` is specified, the `nsURI` attribute of the **EPackage** is set to the initializer of the field.
- If a field named `eNS_PREFIX` is specified, the `nsPrefix` attribute of the **EPackage** object is set to the initializer of the field.

For example, you may define a package that contains the following fields:

```
public interface EPO3Package
{
    String eNAME = "epo3";
    String eNS_URI = "http://com/example/epo3.ecore";
    String eNS_PREFIX = "com.example.epo3";
}
```

Another situation where it may be necessary to provide a Java specification for an **EPackage** is if you need to define an **EDataType** explicitly. This is discussed in Section 6.4. Similarly, you may wish to provide an explicit definition of a map entry, in which case you also need to define your **EPackage**. This is discussed in Section 6.5.

To be recognized as the specification for an **EPackage**, the package interface must include an `eNAME`, `eNS_URI`, or `eNS_PREFIX` field or explicitly define an **EDataType** or a map type. If it does not do at least one of these things, it is simply considered to be an unmodeled interface and is ignored.

Java Specification for Classes

The Java specification for an **EClass** is simply any Java interface that is preceded by an `@model` tag. The **EAttributes** and **EReferences** that belong to the **EClass** are represented by `get()` methods that are defined in the Java interface. The **EOperations** that belong to the **EClass** are represented by other methods in the interface.

The `name` attribute of the **EClass** is set to the name of the Java interface.

The `eSuperTypes` reference of the **EClass** is set to include the **EClasses** corresponding to the interfaces listed in the `extends` clause of the interface. (Note that **EObject** is the implicit base class of all modeled objects in EMF, and therefore should not be listed in the `extends` clause.)

Other properties of the **EClass** can be set using the `@model` properties shown in Table 6.1.

Table 6.1. The `@model` Properties for a Class

Property	Value	Usage
<code>abstract</code>	<code>true false</code>	The <code>abstract</code> attribute of this EClass is set to the specified value.
<code>interface</code>	<code>true false</code>	The <code>interface</code> attribute of this EClass is set to the specified value.

¹When a package is explicitly specified, EMF uses the prefix of the package interface name (that is, everything before “Package”) as the value of the `prefix` attribute of the corresponding **GenPackage** in the generator model, which is created with the core model. If code is generated for the model, this ensures that the generated package interface will replace the existing one.

For example, you could define the following interface, which would map to an **abstract EClass** named “Address”:

```
/**  
 * @model abstract="true"  
 */  
public interface Address  
{  
    ...  
}
```

Attributes

An **EAttribute** is specified as an accessor method in the interface corresponding to the **EClass** that contains the attribute. The method must meet the following criteria:

- It is preceded by an `@model` tag that does not include a property called `parameters`.
- The name of the method begins with “get”, or the type of the method is `boolean` and its name begins with “is”. The character immediately following the “get” or “is” prefix must be uppercase.
- The return type of the method does not correspond to an **EClass**. (Note: if the method is declared to return either `List` or `EList`, then this refers to the type specified with the `@model` tag, as described below.)

The **name** attribute of the **EAttribute** is derived from the name of the method. Specifically, the prefix (that is, “get” or “is”) is removed from the name of the method and the first character of the resulting name is folded to lower case.

If the return type of the method is `List` or `EList`, the **upperBound** of the **EAttribute** is set to `-1` (unbounded) and the **eType** must be specified using the `type` property of the `@model` tag (see below). Otherwise, the **eType** of the **EAttribute** is set to an **EDataType** corresponding to the return type.

Other properties of the **EAttribute** can be set using the `@model` tag. Table 6.2 identifies the `@model` properties and values that are used to set each of the possible **EAttribute** features.

Table 6.2. The `@model` Properties for an Attribute

Property	Value	Usage
<code>changeable</code>	<code>true false</code>	The changeable attribute of this EAttribute is set to the specified value.
<code>dataType</code>	<code>data-type</code>	The specific <i>data-type</i> is used as the EDataType for this EAttribute . See Section 6.4.
<code>default</code> or <code>defaultValue</code>	<code>default-value</code>	The defaultValueLiteral attribute of this EAttribute is set to the string value identified by <i>default-value</i> .
<code>id</code>	<code>true false</code>	The id attribute of this EAttribute is set to the specified value.
<code>lower</code> or <code>lowerBound</code>	<code>integer-value</code>	The lowerBound attribute of this EAttribute is set to <i>integer-value</i> . The specified value must be ≥ 0 .
<code>many</code>	<code>true false</code>	If <code>true</code> , the upperBound attribute of this EAttribute is set to <code>-1</code> (unbounded).
<code>required</code>	<code>true false</code>	If <code>true</code> , the lowerBound attribute of this EAttribute is set to <code>1</code> . Otherwise, the lowerBound attribute is set to <code>0</code> .
<code>transient</code>	<code>true false</code>	The transient attribute of this EAttribute is set to the specified value.
<code>type</code>	<code>type-name</code>	The eType reference of this EAttribute is set to an EDataType corresponding to <i>type-name</i> .
<code>unique</code>	<code>true false</code>	The unique attribute of this EAttribute is set to the specified value.
<code>unsettable</code>	<code>true false</code>	The unsettable attribute of this EAttribute is set to the specified value.

<i>Property</i>	<i>Value</i>	<i>Usage</i>
upper	<i>integer-value</i>	The upperBound attribute of this EAttribute is set to <i>integer-value</i> . The specified value must be >0 or -1 (unbounded).
or		
upperBound		
volatile	true false	The volatile attribute of this EAttribute is set to the specified value.

For example, in order to define the **comment**, **orderDate**, **status**, and **totalAmount** attributes of the **PurchaseOrder** class from the ExtendedPO3 model, you could provide the following Java declarations:

```
/*
 * @model
 */
public interface PurchaseOrder
{
    /**
     * @model
     */
    String getComment();

    /**
     * @model dataType="com.example.epo3.Date"
     */
    Date getOrderDate();

    /**
     * @model
     */
    OrderStatus getStatus();

    /**
     * @model transient="true" changeable="false" volatile="true"
     */
    int getTotalAmount();
}
```

References

An **EReference** is specified as a method in the interface corresponding to the **EClass** that contains the attribute. The method must meet the following criteria:

- It is preceded by an `@model` tag that does not include a property called `parameters`.
- The name of the method begins with “get” followed immediately by an uppercase character.
- The return type of the method corresponds to an **EClass**. (Note: if the method is declared to return a **List** or an **EList**, then this refers to the type specified with the `@model` tag, as described below.)

The **name** attribute of the **EReference** is derived from the name of the method. Specifically, the “get” prefix is removed from the name of the method and the first character of the resulting name is folded to lower case.

If the return type of the method corresponds to an **EClass**, the **eType** of the **EReference** is set to that **EClass**. Otherwise, the return type of the method must be **List** or **EList**, in which case the **eType** of the **EReference** must be specified as part of the `@model` tag (see below) and the **upperBound** of the **EReference** is set to -1.

Other properties of the **EReference** can be set using the `@model` tag. Table 6.3 identifies the `@model` properties and values that are used to set each of the possible **EReference** features.

Table 6.3. The @model Properties for a Reference

Property	Value	Usage
changeable	true false	The changeable attribute of this EReference is set to the specified value.
containment	true false	The containment attribute of this EReference is set to the specified value.
lower	integer-value	The lowerBound attribute of this EReference is set to <i>integer-value</i> . The specified value must be ≥ 0 .
or		
lowerBound		
many	true false	If true, the upperBound attribute of this EReference is set to -1 (unbounded).
opposite	reference-name	The opposite reference of this EReference is set to the EReference corresponding to the specified <i>reference-name</i> . The opposite EReference must belong to the EClass that is identified by the eType of this EReference .
resolveProxies	true false	The resolveProxies attribute of this EReference is set to the specified value.
required	true false	If true, the lowerBound attribute of this EReference is set to 1. Otherwise, the lowerBound attribute is set to 0.
transient	true false	The transient attribute of this EReference is set to the specified value.
type	class-name	The eType reference of this EReference is set to the EClass identified by <i>class-name</i> .
unique	true false	The unique attribute of this EReference is set to the specified value.
unsettable	true false	The unsettable attribute of this EReference is set to the specified value.
upper	integer-value	The upperBound attribute of this EReference is set to <i>integer-value</i> . The specified value must be > 0 or -1 (unbounded).
or		
upperBound		
volatile	true false	The volatile attribute of this EReference is set to the specified value.

For example, in order to define the **items** and **previousOrders** references of the **PurchaseOrder** class from the ExtendedPO3 model, you could provide the following Java declarations:

```
/**
 * @model
 */
public interface PurchaseOrder
{
    /**
     * @model type="Item" containment="true"
     */
    EList getItems();

    /**
     * @model
     */
    PurchaseOrder getPreviousOrder();
}
```

Operations

An **EOperation** is specified as a method in the Java interface corresponding to the **EClass** that contains the operation. The method must meet the following criteria:

- It is preceded by an **@model** tag.
- Either the **@model** tag has a property called **parameters** or the method name is such that it does not conform to one of the **EAttribute** or **EReference** patterns, as described in the previous sections. For example, a method named **island()** would be considered an **EOperation**, but **isLand()** would be considered an **EAttribute** (assuming its return type is **boolean**), unless it included **parameters** with its **@model** tag.

The **name** attribute of the **EOperation** is the same as the name of the method.

The **eType** reference of the **EOperation** is the **EClass** or **EDataType** that corresponds to the return type of the Java method.

The **eParameters** reference of the **EOperation** is constructed from the arguments of the Java method. Specifically, **EParameters** are constructed for the method where the **name** and **eType** features of each **EParameter** are derived from the name and type of each parameter that appears in the Java method specification.

If an **EOperation**'s return type and parameters are standard data types, you normally do not need to specify any properties on the `@model` tag of the method. However, you will need to specify the **parameters** property if the name of the method could be confused with the name of a method that is used to define an **EReference** or **EAttribute**. The `@model` properties that can be set for an **EOperation** are shown in Table 6.4.

Table 6.4. The `@model` Properties for an Operation

<i>Property</i>	<i>Value</i>	<i>Usage</i>
<code>dataType</code>	<code>data-type</code>	The specific <code>data-type</code> is used as the EDataType for this operation's return type. See Section 6.4.
<code>parameters</code>	<code>list-of-types</code>	The <code>list-of-types</code> is a list of space-separated entries, one per parameter. Each entry is either the single character “-”, or a specific data-type as described in Section 6.4. This keyword is usually optional, but it must be specified if the method name begins with “get” or “is”, or if the type of any parameter needs to use a specific EDataType .

For example, in order to specify an operation on the **PurchaseOrder** class called `removeItem` that takes a `String` as an argument and returns a `boolean` result, we could provide the following method in the interface **PurchaseOrder**:

```
/** 
 * @model parameters="com.example.epo3.SKU"
 */
boolean removeItem(String item);
```

Java Specification for Enumerations

Any final static class declaration that appears in your Java package and is preceded by a `@model` tag is mapped to an **EEnum**.

The **name** attribute of the **EEnum** is derived from the name of the Java class. The **eLiterals** reference for the **EEnum** is constructed from the list of **EEnumLiterals** that are derived from the fields that belong to the Java class, as described below.

Java Specification for Enumeration Literals

Each `int` valued field appearing in a Java class that represents an **EEnum** is mapped to an **EEnumLiteral** if it is preceded by an `@model` tag.

If any of the fields has an initial value, that value is used as the **value** of the **EEnumLiteral**.

Normally, the **name** of the **EEnumLiteral** is derived directly from the name of the corresponding field. However, you can add a name specification to the `@model` tag preceding the definition of the field, as described in Table 6.5, in order to control the capitalization of the name.

Table 6.5. The `@model` Properties for an Enum Literal

<i>Property</i>	<i>Value</i>	<i>Usage</i>
<code>name</code>	<code>literal-name</code>	The name of the EEnumLiteral is set to the specified <code>literal-name</code> .

For example, in order to define the **Pending**, **BackOrder**, and **Complete** literals of the **OrderStatus** enumeration from the ExtendedPO3 model, you could provide the following Java declarations:

```
/** 
 * @model
 */
public final class OrderStatus extends AbstractEnumerator
{
    /**
     * @model name="Pending"
     */
    public static final int PENDING = 0;

    /**
     * @model name="BackOrder"
     */
    public static final int BACK_ORDER = 1;

    /**
     * @model name="Complete"
     */
    public static final int COMPLETE = 2;
}
```

Java Specification for Data Types

EDataTypes can be defined explicitly in the Java specification of your model. However, you usually don't need to do this because EMF will automatically create an **EDataType** whenever any attribute or operation in your Java specification makes use of a data type that has not been explicitly defined in your model or in Ecore.

There may be some special cases where you do need to define an **EDataType** explicitly. You might do this if, for example, there was already an existing **EDataType** corresponding to a Java type for which you wished to provide an alternate serialization format. Then, you would need to create a Java interface that represents your **EPackage** (see Section 6.1) and include a `get()` method that accesses the **EDataType**.

To be recognized as explicitly defining an **EDataType**, such a method must meet the following criteria:

- It is preceded by an `@model` tag.
- The return type of the method is `EDataType`.
- The name of the method begins with “get”.

The `@model` properties that can be set for an **EDataType** are shown in Table 6.6.

Table 6.6. The `@model` Properties for a `DataType`

Property	Value	Usage
instanceClass	<code>java-type</code>	The <code>instanceClassName</code> attribute of this EDataType is set to the specified <code>java-type</code> . The <code>java-type</code> should be a fully qualified name of a Java interface or class.
serializable	<code>true</code> <code>false</code>	The <code>serializable</code> attribute of this EDataType is set to the specified value.

For example, in order to define the **Date** data type from the ExtendedPO3 model, you could provide the following Java declarations:

```

public interface EPO3Package
{
    /**
     * @model instanceClass="java.util.Date"
     */
    EDataType getDate();
}

```

If you look back at Section 6.2.1 where **Date** is referenced as the type of the **orderDate** attribute, you may notice that the data type's name is prefixed by "com.example.epo3", the name of the Java package that corresponds to the **EPackage** in which **Date** is defined. Such qualification is necessary in any reference to an explicitly defined **EDataType**.

Java Specification for Maps

EMF provides special support for defining maps that are strongly typed and order preserving. This support requires the creation of a special map entry type. As described in Chapter 5, a map entry is an **EClass** that implements the **Map.Entry** interface and has two features that are named "key" and "value". A containment reference whose target is such an **EClass** will be treated specially by the EMF code generator. It will be implemented using an **EMap**, instead of just an **EList**.

You can define a map entry in your Java source code either explicitly or implicitly. An explicit definition is made by creating an accessor method for the map entry class in the interface that corresponds to the **EPackage**. An implicit definition is made by defining a containment-type **EReference** in an **EClass** that uses a map entry that has not been defined elsewhere. An implicit definition is somewhat simpler but has some limitations.

Explicit Definition of Map Entries in a Package

To define a map entry explicitly, you provide an accessor method for that entry in the interface that represents the **EPackage** that contains it.

To be recognized as such, the method must meet the following criteria:

- It is preceded by an **@model** tag. (This **@model** tag must not have an **instanceClass** defined.)
- The return type of the method is **EClass**.
- The name of the method begins with "get" followed immediately by an uppercase character.

The **@model** tag can define any of the properties described in Table 6.7.

*Table 6.7. The **@model** Properties of a Map Entry EClass*

Property	Value	Usage
features	feature-list	The <i>feature-list</i> is a list of the names of the EAttributes and EReferences that belong to the map entry. If this list is omitted, features named "key" and "value" are assumed.
feature-property	value	A <i>feature-property</i> specifies a property for one of the map entry's EAttributes or EReferences . The property name is the concatenation of one of the feature names defined in the <i>feature-list</i> , above, with any valid property for that feature, as described in Sections 6.2.1 and 6.2.2. Note that the first character of the property should be capitalized. The <i>value</i> is simply the desired value for the feature's property.

For example, the following Java code will define an **EPackage** named "POPackage" that contains two map types:

```

public interface POPackage
{
    /**
     * @model features="key value info"
     * keyType="int" keyRequired="true"
     * valueType="PurchaseOrder"

```

```

        * valueContainment="true" infoType="String"
        */
EClass getIntToOrderMapEntry();

/**
 * @model keyType="int"
 * valueType="PurchaseOrder"
 * valueContainment="true"
 * valueMany="true"
 */
EClass getIntToOrdersMapEntry();
}

```

The first map entry is named “IntToOrderMapEntry”. It’s used to map a single `int` key to a single value, a `PurchaseOrder` object. It also has an additional `String` attribute called “info”. The second map entry, called “IntToOrdersMapEntry”, maps an `int` key to a list of `PurchaseOrder` objects.

Defining References to Map Entries

A map is defined like an ordinary **EReference**, as described Section 6.2.2, with the following additional criteria:

- The return type of the `get()` method must be either `Map` or `EMap`.
- The `@model` tag includes either a `mapType` or both the `keyType` and `valueType` properties.

An **EReference** defined this way will automatically have its `containment` and `many` attributes set to true. The `@model` tag that precedes the definition of the map can have the properties shown in Table 6.8, in addition to the normal **EReference** properties described in Section 6.2.2.

Table 6.8. Additional `@model` Properties of a Map **EReference**

Properties	Value	Usage
<code>keyType</code>	<code>key-type</code>	The <code>key-type</code> is used as the <code>eType</code> of the <code>key</code> attribute of the EClass that is created to represent the map entry.
<code>mapType</code>	<code>map-type</code>	The <code>map-type</code> either identifies a map entry EClass that has already been defined or specifies a name that should be given to a new, implicitly defined map entry EClass .
<code>valueType</code>	<code>value-type</code>	The <code>value-type</code> is used as the <code>eType</code> of the <code>value</code> attribute of the EClass that is created to represent the map entry.

If the `mapType` property is included and its `map-type` identifies a map entry **EClass** that has been explicitly defined in an **EPackage** or implicitly defined as the type of another **EReference**, then the type of this **EReference** is that map entry. The `keyType` and `valueType` properties should not be included in this case.

If the `mapType` property is not included or its `map-type` is not the name of an existing **EClass**, then this **EReference** implicitly defines a map entry, and the `keyType` and `valueType` properties must be included. In this case, any `map-type` is used to name the implicitly defined map entry. The default name is “`key-typeTovalue-typeMapEntry`”.

The following code defines an **EClass** called “Index” that includes two **ERefences** to map entries:

```

/**
 * @model
 */
public interface Index
{
    /**
     * @model keyType="int" valueType="PurchaseOrder"
     */
EMap getOrderMap();
}

```

```
/**  
 * @model mapType="IntToOrdersMapEntry"  
 */  
EMap getCustomerOrderMap();  
}
```

The first method defines an **EReference** called “orderMap” using an implicitly defined map entry **EClass**, which will be given the default name “EIntToPurchaseOrderMapEntry”. The second **EReference**, named “customerOrderMap”, uses the existing “IntToOrdersMapEntry” map entry, which was described in the previous section.

Chapter 7. XML Schema

If you wish to create an object model for manipulating an XML data structure of some type, EMF provides a particularly desirable approach based on XML Schema. EMF can create a core model that corresponds to a schema, allowing you to leverage the code generator, or dynamic EMF, to provide a Java API for manipulating instances of that schema.

From a modeling perspective, XML Schema is not as expressive as Ecore. For example, it can't be used to define bidirectional references, or to provide the type of a reference target.¹ Nevertheless, for the kinds of models that can be expressed using XML Schema, for example, typical XML message structures, the mapping to Ecore is simple and direct:

- A schema maps to an **EPackage**.
- A complex type definition or a top-level element declaration maps to an **EClass**.
- A simple type definition maps to an **EDatatype**, except for the types `anyURI`, `QName`, `IDREF`, or `IDREFS`, which instead map to the **EClass** for **EObject**.
- An attribute declaration or a nested element declaration maps to an **EAttribute** if its type maps to an **EDatatype**, or to an **EReference** if its type maps to an **EClass**.

Unlike core models created from other representations, a model that is produced from XML Schema retains extra information about its source. Each Ecore element includes an **EAnnotation**, which provides additional information from the schema, related to serialization format. Because XML Schema's primary purpose is to define the structure of XML instance documents, instances of the core model should conform to the corresponding schema. The annotations are used to customize the default EMF serialization for exactly this purpose.

Recall from Chapter 5 that an **EAnnotation** has a **source** attribute that identifies its type, and a **details** map, the contents of which depends on the type of annotation. The annotations added to schema-defined Ecore elements will all have their **source** attribute set to the value "`http://org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore`". The **details** map will always include at least one entry, whose key is "representation". The corresponding value identifies the type of XML Schema construct that the Ecore element was created from. Additional entries may be included in the **details** map, depending on the type of the Ecore element.

In the following sections, we'll look at the details of how the various schema elements map to Ecore, and how their associated **EAnnotations** are initialized. The use of XSD2Ecore annotations to control the XML serialization is described in Chapter 13.

¹Future versions of EMF may support annotating an XML Schema with the missing information, similar to the way `@model` annotations are used when defining an EMF model using Java interfaces.

Schema Definition of Packages

A complete schema, itself, maps to an **Epackage**. Note, however, that a schema may import from other schemas—either explicitly, using an `xsd:import` element, or implicitly, from the schema for schemas. A separate **Epackage** will be created corresponding to each imported schema that declares a different target namespace from the original schema.

Recall from Chapter 5 that an **Epackage** has three attributes: `name`, `nsURI`, and `nsPrefix`. The value of these attributes depends on whether the **Epackage**'s corresponding schema has a target namespace. If the schema has a target namespace:

- The `nsURI` is set to the target namespace of the schema.
- The `nsPrefix` is set to a qualified (that is, “.” separated) package name, derived from the target namespace.
- The `name` attribute is set to the last component of the qualified package name that is used for the `nsPrefix`.

If a schema has no target namespace, then the attributes are initialized based on the URI (file path) of the schema document itself:

- The `nsURI` is set to the URI of the schema document.
- The `nsPrefix` and `name` are both set to the last segment of the URI (the short file name), excluding the file extension.

The **EAnnotation** that is added to an **Epackage** created from a schema has two entries in it, “representation” and “targetNamespace”. The “representation” will be set to the value “schema” and the “targetNamespace” will be set to the target namespace of the schema, if there is one.

Consider the following schema definition:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.com/simplepo"
    xmlns:po="http://www.example.com/simplepo">
    ...
</xsd:schema>
```

Because the schema includes a `targetNamespace` attribute, this will produce an **Epackage** with its `nsURI` set to the target namespace “<http://www.example.com/simplepo>”. The `nsPrefix` will be set to “`com.example.simplepo`”, and the `name` will be “`simplepo`”. Here's what the corresponding **Epackage** looks like, serialized to XML:

```
<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="simplepo"
    nsURI="http://www.example.com/simplepo"
    nsPrefix="com.example.simplepo">
    <eAnnotations
        source="http://org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
        <details key="representation" value="schema"/>
        <details key="targetNamespace"
            value="http://www.example.com/simplepo"/>
    </eAnnotations>
    ...
</ecore:EPackage>
```

Notice that when EMF processes the target namespace to produce a qualified package name, it ignores the “www” in the URI authority, and reverses the components: “example.com” became “com.example”. If necessary, it would also convert each name in the qualified package name to a valid Java identifier, and make all of the characters of the package lowercase to conform to Java naming conventions. If names are mixed case, it will also break them into separate package components.

For example, the simple purchase order schema that we introduced in Chapter 2 has the target namespace “<http://www.example.com/SimplePO>”. Because it has the mixed case name component “SimplePO” instead of the lowercase name, “simplepo”, the derived package name will be slightly different. In this case, the **EPackage**’s attribute **nsPrefix** would be “com.example.simple.po” and the **name** would be just “po”.

Now suppose we remove the target namespace, like this:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.com/simplepo"
    xmlns:PO="http://www.example.com/simplepo">
    ...
</xsd:schema>
```

Let’s assume that this schema is in a resource named *file:/c:/myexample/simplepo.xsd*. In this case, the **nsURI** would be the complete file URI, “*file:/c:/myexample/simplepo.xsd*”, while the **name** and **nsPrefix** would be the file’s short name, “simplepo”.

Schema Definition of Classes

Each complex type definition or top-level element declaration maps to an **EClass** in the **eClassifiers** reference of the **EPackage** corresponding to the schema that contains it.

Complex Type Definitions

An **EClass** is created for each complex type definition in a schema.

- The **name** of the **EClass** is based on the name of the complex type or, if it is anonymous, on that of the containing element. The name is converted to a valid Java identifier and capitalized if necessary.
- The **abstract** attribute of the **EClass** is set to the value of the **abstract** attribute of the complex type definition, if present.
- If the complex type is derived from another complex type, the other’s corresponding **EClass** is added to the **eSuperTypes** of the **EClass**.

The **EAnnotation** that is added to an **EClass** created from a complex type has at least three entries in it, “representation”, “name”, and “targetNamespace”. The “representation” will be set to “type” for a top-level complex type definition, or to “element” if the definition is anonymous. The “name” will be set to the name of the complex type, as it appeared before mangling, and the “targetNamespace” will be set to the target namespace of the schema that contains the complex type definition.

Consider the following top-level complex type definition:

```
<xsd:complexType name="price">
    ...
</xsd:complexType>
```

This will create an **EClass** with its **name** set to “Price”, and add it to the **eClassifiers** of the **EPackage** corresponding to the complex type’s schema.

```

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

```

In some cases, the **EAnnotation** of an **EClass** may include a fourth entry, “feature-order”. Whenever the schema specifies that an element that maps to an **EReference** should occur ahead of one that maps to an **EAttribute**, this entry will be added to the annotation details of the containing **EClass**. Its value will be a space-separated list of the class' feature names in the order in which they should be serialized.

Top-Level Element Declarations

An **EClass** is also created for each top-level element declaration in a schema.

- The **name** of the **EClass** is based on the name of the element. It is converted to a valid Java identifier and capitalized if necessary.
- The **abstract** attribute of the **EClass** is set to the value of the **abstract** attribute of the element declaration, if present.
- If the type of the element has a corresponding **EClass**, it is added to the **eSuperTypes** of the **EClass**.
- If the element declaration includes a substitution group, the **EClass** corresponding to the substitution group's head element is added to the front of the **eSuperTypes** of the **EClass**.

The **EAnnotation** for the **EClass** will have its “representation” set to the value “element”, its “name” set to the name of the element declaration, and its “targetNamespace” set to the target namespace of the schema that contains the element declaration.

Consider the following top-level element declaration:

```
<xsd:element name="comment" type="xsd:string"/>
```

This will create an **EClass** with its **name** set to “Comment”, like this:

```

<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>
  ...
</eClassifiers>

```

Because the element is of a simple type, `xsd:string`, the corresponding **EClass** has no **eSuperTypes**. Assume, instead, that the element declaration had the complex type from the previous section as its type:

```
<xsd:element name="comment" type="po:price"/>
```

In this case, the **EClass** corresponding to the complex type `po:price` would be added to the **eSuperTypes** of the class:

```

<eClassifiers xsi:type="ecore:EClass" name="Comment"
               eSuperTypes="#//Price">
  <eAnnotations
    source="http://org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
    <details key="representation" value="element"/>
    <details key="name" value="comment"/>
    <details key="targetNamespace"/>
  </eAnnotations>
  ...
</eClassifiers>

```

If a top-level element declaration includes an anonymous complex type definition, then a single **EClass** is created to represent both. Its **eSuperTypes** are determined only from the complex type, as described in the previous section. Moreover, the **EAttributes**, **EReferences** and **EOperations** that it contains will be determined just as if it represented an ordinary complex type definition.

Schema Definition of Attributes

An **EAttribute** is created in an **EClass** corresponding to a complex type definition (that doesn't have mixed content) for:

- Each contained attribute declaration
- Each contained nested element declaration of simple type

In addition, either an element declaration with a simple type or a complex type definition with simple content will result in the creation of a single **EAttribute** named "value" in its corresponding EClass. We'll discuss each of these cases in the next three sections.

Attribute Declarations

Each schema attribute declaration maps to an **EAttribute** in the **EClass** corresponding to the complex type definition containing the attribute declaration.

- The **name** of the **EAttribute** is set to the name of the attribute, converted to a valid (non-capitalized) Java identifier, if necessary.
- The **eType** of the **EAttribute** is an **EDataType** corresponding to the attribute's simple type, as described in Section 7.5.
- The **iD** attribute of the **EAttribute** is set to `true` if the type of the attribute is, or is derived from, `xsd:ID`.
- If the schema attribute is `required`, then the **lowerBound** of the **EAttribute** is set to 1. Otherwise, it's 0. The **upperBound** is always 1.

The **EAnnotation** for the **EAttribute** will have its "representation" set to the value "attribute", its "name" set to the name of the attribute, and its "targetNamespace" set to the target namespace of the schema that contains the attribute declaration.

Consider the following example:

```
<xsd:attribute name="totalAmount" type="xsd:float" use="required"/>
```

This will produce an **EAttribute** with name set to "totalAmount", **eType** set to the built-in Ecore **EDataType** corresponding to `xsd:float` (that is, **EFloat**), and **lowerBound** set to 1.

```

<eAttributes name="totalAmount" eType=
             "ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EFloat"
             lowerBound="1">
  <eAnnotations
    source="http://org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
    <details key="representation" value="attribute"/>

```

```

<details key="name" value="totalAmount"/>
<details key="targetNamespace"/>
</eAnnotations>
</eAttributes>

```

Nested Element Declarations

Each nested element declaration whose type maps to an **EDataType** (as described in Section 7.5) is mapped to an **EAttribute** in the **EClass** corresponding to the complex type definition containing the nested element declaration.

- The **name** of the **EAttribute** is set to the name of the element, converted to a valid (non-capitalized) Java identifier, if necessary.
- The **eType** of the **EAttribute** is an **EDataType** corresponding to the simple type of the element.
- The **lowerBound** and **upperBound** of the **EAttribute** are set to the values of the **minOccurs** and **maxOccurs** attributes of the element declaration, respectively.

The **EAnnotation** for the **EAttribute** will have its “representation” set to the value “element”, its “name” set to the name of the nested element, and its “targetNamespace” set to the target namespace of the schema that contains the element declaration.

Imagine the following as a nested element declaration:

```
<xsd:element name="shipTo" type="xsd:string"/>
```

This will produce an **EAttribute** with name set to “shipTo”, **eType** set to the built-in Ecore **EDataType** corresponding to **xsd:string** (that is, **EString**), and **lowerBound** and **upperBound** attributes both set to 1, the default value of **minOccurs** and **maxOccurs** in an element declaration.²

```

<eAttributes name="shipTo" eType=
  "ecore:EDataType http://www.eclipse.org/emf/2002/Ecore//EString"
  lowerBound="1">
<eAnnotations
  source="http://org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
  <details key="representation" value="element"/>
  <details key="name" value="shipTo"/>
  <details key="targetNamespace"/>
</eAnnotations>
</eAttributes>

```

Simple Content

A complex type definition with simple content or a top-level element declaration with a simple type will produce a single **EAttribute** whose **name** is “value” in its corresponding **EClass**. The **eType** of this **EAttribute** is an **EDataType** corresponding to the simple base of the complex type or to the type of the element. The **EAnnotation** for the **EAttribute** has only a “representation” entry, with value “simple-content”.

The following complex type definition has simple (string) content:

```

<xsd:complexType name="comment">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string"/>
  </xsd:simpleContent>
</xsd:complexType>

```

²Note that **upperBound** is also 1 by default in Ecore and therefore will not be serialized in the XMI.

An **EClass** named “Comment” will be created for the complex type, in the manner described in Section 7.2.1. It will contain an **EAttribute** whose name is “value” and whose **eType** is set to the built-in Ecore **EDataType** corresponding to `xsd:string` (that is, **EString**).

```
<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>
```

In section 7.2.2, we first saw the following top-level element declaration:

```
<xsd:element name="comment" type="xsd:string"/>
```

We described the EClass that would be created, but said nothing about its contents. In fact, it would contain exactly the same **EAttribute** as the one created for the above complex type definition.

Schema Definition of References

An **EReference** is created in an **EClass** corresponding to a complex type definition (that doesn't have mixed content) for each nested element declaration with a complex type or with any of the following simple types: `anyURI`, `QName`, `IDREF`, or `IDREFS`. A single **EReference** will also be created for a complex type definition with mixed content. We'll look at all of these cases in the following sections.

Nested Element Declarations

Each nested element declaration whose type maps to an **EClass** or whose type is one of `anyURI`, `QName`, `IDREF`, or `IDREFS` is mapped to an **EReference** in the **EClass** corresponding to the complex type definition containing the nested element declaration.

- The **name** of the **EReference** is set to the name of the nested element, converted to a valid (non-capitalized) Java identifier, if necessary.
- The **eType** of the **EReference** is set to the **EClass** corresponding to the element's complex type or to **EObject** if the type is `anyURI`, `QName`, `IDREF`, or `IDREFS`.
- If the element type is complex, then **containment** is set to `true`. That is, for the types `anyURI`, `QName`, `IDREF`, or `IDREFS`, it will be `false`.
- The **lowerBound** and **upperBound** of the **EReference** are set to the values of the `minOccurs` and `maxOccurs` attributes of the element declaration, respectively.

The **EAnnotation** for the **EReference** will be exactly the same as that of an **EAttribute** corresponding to a nested element declaration, as described in Section 7.3.2.

Imagine the following as a nested element declaration:

```
<xsd:element name="items" type="PO:Item" minOccurs="0"
  maxOccurs="unbounded"/>
```

This will produce an **EReference** with **name** set to “items”, **eType** set to the **EClass** corresponding to the complex type `PO:Item`, **containment** set to `true`, and **upperBound** set to `-1` (unlimited).

```
<eReferences name="items"
  eType="#//Item" upperBound="-1" containment="true">
  <eAnnotations
    source="http://org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
```

```

<details key="representation" value="element"/>
<details key="name" value="items"/>
<details key="targetNamespace"/>
</eAnnotations>
</eReferences>

```

Mixed Content

A complex type with mixed content will produce a single **EReference** named “contents”, whose **eType** is the **EClass** for **EObject**, **upperBound** is **-1** (unbounded), and **containment** is **true**. Nested element declarations are ignored in this situation, but attribute declarations are processed as described in Section 7.3.1.

The **EAnnotation** in this case has a single entry, “representation”, with value “general-content”.

Consider the following complex type definition:

```

<xsd:complexType name="MixedBag" mixed="true">
  ...
</xsd:complexType>

```

As we saw in Section 7.2.1, an **EClass** named “MixedBag” will be created for the complex type definition. Because of the mixed content, the **EClass** will contain a single **EReference** named “contents” whose **eType** is the **EClass** for **EObject**.

```

<eReferences name="contents" eType=
  "ecore:EClass http://www.eclipse.org/emf/2002/Ecore#//EObject"
  upperBound="-1" containment="true">
<eAnnotations
  source="http://org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
  <details key="representation" value="general-content"/>
</eAnnotations>
</eReferences>

```

Schema Simple Types

A simple type in a schema maps to an Ecore **EDataType** whether it is user-defined or one of the schema predefined types. The only exceptions are the types **anyURI**, **QName**, **IDREF**, and **IDREFS**, which as we've seen in the previous section, map to an **EClass** (that is, the **EClass** for **EObject**) instead.

Some of the predefined simple types map directly to built-in **EDataTypes**, as shown in Table 7.1.

Table 7.1. Mapping from XML Schema Types to Ecore EDatas

Schema Simple Type	Ecore EDatas
boolean	EBoolean
byte	EByte
double	EDouble
float	EFloat
int	EInt
long	ELong
short	EShort
string	EString
unsignedByte	EShort
unsignedInt	ELong
unsignedShort	EInt

A user-defined simple type that derives either directly or indirectly from one of these predefined simple types maps to the same **EDataType** as that base type. For example, consider the following simple type definition:

```
<xsd:simpleType name="myinteger">
  <xsd:restriction base="xsd:int">
    <xsd:minInclusive value="1"/>
    <xsd:maxInclusive value="10"/>
  </xsd:restriction>
</xsd:simpleType>
```

This simple type will be mapped to the Ecore data type **EInt**, because it restricts the schema type `xsd:int`.

Other simple types that do not derive from any type in Table 7.1 will map to a manufactured **EDataType** with **name** equal to the name of the type, and an **instanceClassName** of “`java.lang.String`” (that is, the type will be represented as a Java `String`). For example, the simple type `xsd:decimal` will map to the following **EDataType**:

```
<eClassifiers xsi:type="ecore:EDataType" name="Decimal"
  instanceClassName="java.lang.String">
  <eAnnotations
    source="http://org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
    <details key="representation" value="type"/>
    <details key="name" value="decimal"/>
    <details key="targetNamespace"
      value="http://www.w3.org/2001/XMLSchema"/>
  </eAnnotations>
</eClassifiers>
```

One exception to the above is a user-defined type that is derived from `NCName` and restricted by enumeration. In this case, the type will map to an **EEnum**, and each enumeration will map to an **EEnumLiteral** of the **EEnum**. For example, consider the following simple type:

```
<xsd:simpleType name="OrderStatus">
  <xsd:restriction base="xsd:NCName">
    <xsd:enumeration value="Pending"/>
    <xsd:enumeration value="BackOrder"/>
    <xsd:enumeration value="Complete"/>
  </xsd:restriction>
</xsd:simpleType>
```

Instead of an ordinary **EDataType**, this type will map to an **EEnum** whose **name** is “`OrderStatus`”. The **EEnum** contains three **EEnumLiterals** with **name** attributes set to “`Pending`”, “`BackOrder`”, and “`Complete`”. The literal values are sequentially ordered, starting from 0.

```
<eClassifiers xsi:type="ecore:EEnum" name="OrderStatus">
  <eAnnotations
    source="http://org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore">
    <details key="representation" value="type"/>
    <details key="name" value="OrderStatus"/>
    <details key="targetNamespace"
      value="http://www.example.com/Library"/>
  </eAnnotations>
  <eLiterals name="Pending"/>
  <eLiterals name="BackOrder" value="1"/>
  <eLiterals name="Complete" value="2"/>
</eClassifiers>
```

Chapter 8. UML

In becoming the standard language for modeling software systems, UML has garnered wide acceptance among modeling enthusiasts and extensive support from modeling tools. If you're already comfortable with UML, this will be a convenient representation for specifying your models. But, since EMF only concerns itself with a small subset of UML, you really don't need to be a UML expert to use this feature. The UML constructs that are relevant to EMF are essentially the ones that are defined in a UML class diagram: packages, classes, attributes, associations, and operations. There are a few special conventions for some properties of the UML constructs, but otherwise, the mapping is really quite straightforward.

Some of the attributes of certain Ecore elements cannot be expressed directly in UML.¹ For example, the **resolveProxies** attribute of an **EReference** cannot be defined in a UML class diagram. Often, you don't need to worry about these non-UML attributes because EMF chooses default values that are appropriate in the majority of cases. However, if you do need to override the values of such attributes, the way you do it will depend on the UML modeling tool that you use. Rational Rose,² for example, allows you to add extended properties to the elements in a model. Exploiting this feature, EMF provides a Rose property file called *ecore.pty*, which defines all the non-UML properties needed to completely express an EMF model in Rose. If you use it with your model, then you can use Rose to set any of the extended Ecore attributes to non-default values directly. For other UML modeling tools, you'll need to refer to the documentation provided with the tool. If the tool supports EMF model creation, then you should be able to use it to modify the non-UML Ecore attributes as well. Alternatively, you can always edit the *.ecore* file that is exported from the UML tool to change any of the attributes after the model has been created.

In the following sections, we will explore how UML constructs map to Ecore elements and we'll identify all of the non-UML Ecore properties, along with the specific UML constructs to which they apply.

UML Packages

As you may recall from Chapter 5, the top-level element of a core model is an **EPackage**. Each UML package that you specify in your model will map to an **EPackage**.

- The **name** of each **EPackage** is the same as the name of the corresponding UML package.
- The **eClassifiers** that belong to an **EPackage** are determined by the UML classes that are contained in the UML package. The type of each **EClassifier** depends on the corresponding UML class' stereotype. This is discussed in the following section.
- If the UML package is a subpackage of another one, then its **EPackage** will be contained in the **eSubpackages** of the other's **EPackage**.

¹We're now referring specifically to UML version 1.4.

²We're mentioning Rational Rose in particular because EMF has built-in import support for models created with this tool.

- The **nsURI** and **nsPrefix** attributes of the **EPackage** cannot be expressed directly in UML. Default values will be chosen automatically based on the name of the package. Specifying different values requires the use of a tool-specific technique. In Rose, these attributes are represented by package properties defined in the *ecore.pty* property file.³

For example, the ExtendedPO3 model contains the following packages (Figure 8.1):



Figure 8.1. UML packages.

This will map to two **EPackages** whose **name** attributes are “epo3” and “supplier”, respectively. The **epo3** package’s **nsURI** will be set to “<http://com/example/epo3.ecore>” and its **nsPrefix** will be “`com.example.epo3`”. The **nsURI** and **nsPrefix** of the supplier package follow similar patterns. Neither of these packages has any **eSubpackages**.

UML Specification for Classifiers

A UML class will map to an **EClass**, **EEnum**, or an **EDataType**, depending on the class’ stereotype. We’ll look at each of these three cases in the following sections.

Classes

Every UML class that has either a stereotype of `<<interface>>` or no stereotype at all, will map to an **EClass** in the **EPackage** corresponding to UML package of the class.

- The **name** of each **EClass** is the same as the name of the corresponding UML class.
- The **eSuperTypes** are the **EClasses** that correspond to the UML classes that are the targets of generalization relationships. If you specify multiple generalizations, you can attach the `<<extend>>` stereotype to one of them, to make it first in the **eSuperTypes** list. This is an important distinction because the EMF code generator uses the first of the **eSuperTypes** for the implementation base class of the generated Java class.
- The **eAttributes** are a collection of **EAttributes** that are derived from the UML attributes for the class.
- The **eReferences** are a collection of **EReferences** that are derived from the UML associations for the class.
- The **eOperations** are a collection of **EOperations** that are derived from the UML operations for the class.
- The **abstract** attribute is set to true if the UML class is abstract, as indicated in a class diagram by the italicization of its name. In Rose, whether a class is abstract is controlled by a checkbox that appears on the **Detail** page of its **Class Specification** dialog.
- The **interface** attribute of the **EClass** is not directly represented in UML, so we use a stereotype for it; the **interface** attribute is set to `true` if the UML class has the stereotype `<<interface>>`.

Class **Address** and its subclasses, from the ExtendedPO2 model, are represented in UML as follows (Figure 8.2):

³The property file also defines two additional properties, **basePackage** and **prefix**, which map to attributes of the corresponding **GenPackage** in the generator model. The value of **basePackage** is also used in computing the default values for **nsURI** and **nsPrefix**.

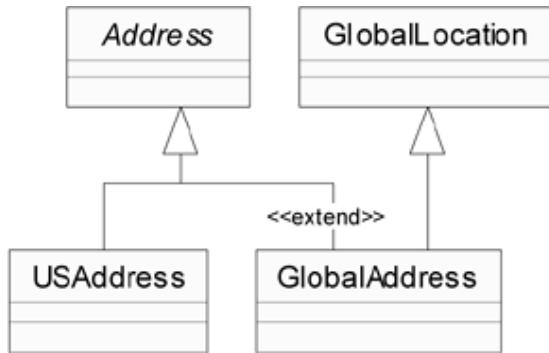


Figure 8.2. UML classes with generalization relationships.

These four UML classes map to the following **EClasses**:

- The **EClass** named “Address” has no **eSuperTypes**. Its **abstract** attribute is `true` and its **interface** attribute is `false`.
- The **EClass** named “USAddress” has class **Address** in its **eSuperTypes** reference. Its **abstract** and **interface** attributes are `false`.
- The **EClass** named “GlobalLocation” has no **eSuperTypes**, and both the **abstract** and **interface** attributes are `false`.
- The **EClass** named “GlobalAddress” has **Address** and **GlobalLocation** as its **eSuperTypes**. Note that the generalization relation has been marked with the stereotype `<<extend>>` in the UML diagram. This means that **Address** will be the first **eSuperTypes** entry, which in turn will mean that the generated implementation class for **GlobalAddress** will extend from **Address**. The **abstract** and **interface** attributes are both `false` for this class.

The various attributes, references, and operations for these classes are not shown in this diagram. These elements will be discussed in later sections.

Enumerations

A UML class where the stereotype is specified as `<<enumeration>>` is mapped to an **EEnum**.

- The **name** of each **EEnum** is the same as the name of the corresponding UML class.
- The **eLiterals** are a collection of **EEnumLiterals** that are derived from the attributes that belong to the UML `<<enumeration>>` class. The **name** of each **EEnumLiteral** is set to the name of the UML attribute. The **value** of a literal, an integer, can be provided explicitly as the UML attribute's initial value. Otherwise, it will be assigned a sequential value by default. Note that the attributes that are given in the UML class have no type. Only the name of the attribute is needed in this case.

For example, the ExtendedPO3 model includes the following UML class (Figure 8.3):

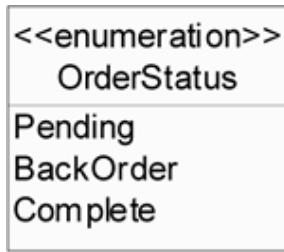


Figure 8.3. A UML class representing an enumerated type.

This maps to an **EEnum** named “OrderStatus” that contains three **EEnumLiterals**. The literals are automatically assigned sequential `int` values, starting with 0. Thus, **Pending** is assigned the value 0, **BackOrder** is assigned 1, and **Complete** is assigned 2. If you want to specify any literal values yourself, you can do so by including initial values for the UML attributes that represent those literals (Figure 8.4):

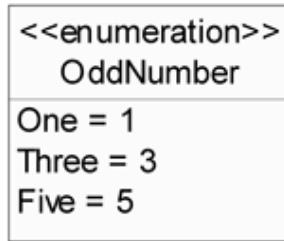


Figure 8.4. Specifying literal values.

Data Types

A UML class where the stereotype is specified as `<<datatype>>` is mapped to an **EDataType**.

- The **name** of each **EDataType** is the same as the name of the corresponding UML class.
- The **instanceClassName** is set to the name of the UML class' single attribute, which must have a stereotype of `<<javaclass>>`. This value is the fully qualified name of the Java class or interface that the **EDataType** will be used to model.

There are two data types defined in the ExtendedPO3 model (Figure 8.5):



Figure 8.5. UML classes representing data types.

The **EDataType** named “SKU” is implemented by `java.lang.String`. The **EDataType** named “Date” is implemented by `java.util.Date`.

UML Specification for Attributes

Each attribute of a UML class is mapped to an **EAttribute** of the corresponding **EClass**.

- The **name** of each **EAttribute** is the same as the name of the corresponding UML attribute.
- The **eType** of each **EAttribute** is derived from the type of the UML attribute. The type must be either a basic Java type,⁴ or an **EEnum** or **EDataType** that has been defined in the model.
- By default, the **lowerBound** of each **EAttribute** is 0 and the **upperBound** is 1. The lower and upper bounds can also be specified explicitly, by attaching a stereotype to the UML attribute. The syntax for this is the same as you would use to specify the upper and lower bounds of an association.
- The **defaultValueLiteral** for an **EAttribute** is derived from the initial value of the UML attribute, if specified.

⁴Ecore has built-in **EDataTypes** for all the Java primitive types (`boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`) their corresponding `java.lang` wrapper classes (for example, `java.lang.Character`), and `java.lang.String`.

- By default, the **transient**, **volatile**, **unique**, **unsettable**, and **id** attributes are all `false` and the **changeable** attribute is `true`. There is no way to specify these attributes in standard UML.

Single-Valued Attributes

The **name** and **eType** of an **EAttribute** are derived directly from the name and type of the attribute in UML. Unless explicitly indicated, all attributes are single-valued. For example, consider this attribute in class **Supplier** (Figure 8.6):

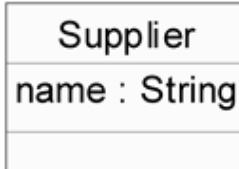


Figure 8.6. A single-valued attribute.

The corresponding **EAttribute** has its **name** set to “name”. Its **eType** is the Ecore built-in data type **EString**, which represents `java.lang.String`. The **lowerBound** and **upperBound** are 0 and 1 respectively, and **defaultValueLiteral** is `null`.

Multi-Valued Attributes

In order to indicate that an attribute is multi-valued, we need to specify a UML stereotype for the attribute. The content of the stereotype has the same syntax as the multiplicity specification for a role in an association. For example, `<<0..*>>` would indicate a lower bound of 0 and no upper bound. Consider the **location** attribute in class **GlobalAddress** (Figure 8.7):

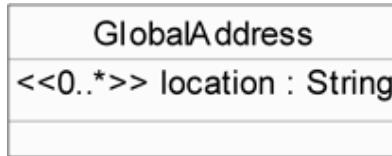


Figure 8.7. A multi-valued attribute.

The **EAttribute** in this case is named “location”. As in the previous example, its **eType** is the **EString** data type, but in this case, the **lowerBound** and **upperBound** attributes are set explicitly to 0 and `-1` (unbounded), respectively.

Attributes with a Default Value

We can set the default value of an attribute by specifying an initial value in the UML. The initial value must conform to the proper syntax for a Java literal of the attribute’s type. The value is used to set the **defaultValueLiteral** attribute (that is, the string form) of the corresponding **EAttribute**. For example, consider class **Customer** (Figure 8.8):

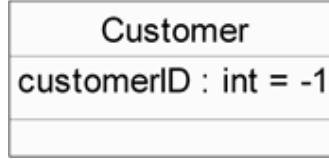


Figure 8.8. Specifying a default value.

The **EAttribute** named “customerID” is of primitive type int (that is, the **EInt** data type), and its **defaultValueLiteral** attribute will be set to the string “-1”.

Attributes with Ecore Properties

As mentioned previously, the **transient**, **volatile**, **unique**, **unsettable**, **changeable**, and **id** attributes of an **EAttribute** cannot be expressed in UML. For example, consider the **totalAmount** attribute in class **PurchaseOrder** (Figure 8.9):



Figure 8.9. The computable **totalAmount** attribute.

In the UML diagram, it looks like any other simple attribute; it represents an **EAttribute** named “**totalAmount**”, with **eType** primitive **int**, **lowerBound** of **0**, **upperBound** of **1**, and no specified **defaultValueLiteral**.

In the model, we would also like to specify that the attribute is **transient**, **volatile**, and not **changeable**. Since these properties are not part of UML, we need to set them using a tool-specific technique. In Rose, we would use the **Ecore** page of the **Attribute Specification** dialog, which is available when using the *ecore.pty* property file with your model. Properties corresponding to the non-UML attributes can be set on this page, as shown in Figure 8.10.

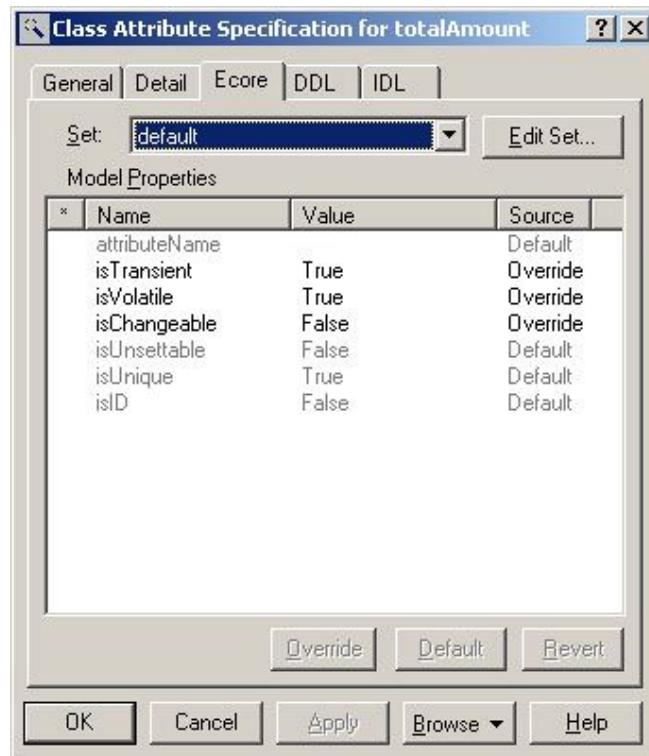


Figure 8.10. Setting non-UML attribute properties with Rational Rose.

Now, the “totalAmount” **EAttribute** will have its **transient** and **volatile** attributes set to `true`, and its **changeable** attribute set to `false`.

UML Specification for References

Each navigable association end of a UML class is mapped to an **EReference** of the corresponding **EClass**.

- The **name** of each **EReference** is the same as the role name of the corresponding UML association end.
- The **eType** of the **EReference** is the **EClass** that is at the end of the UML Association.
- The **lowerBound** and the **upperBound** of each **EReference** are derived from the multiplicity of the corresponding UML association end. For example, if you specify the multiplicity as `0..1`, the **lowerBound** is set to `0` and the **upperBound** is set to `1`. If the multiplicity is `0..*`, the **lowerBound** will be `0` and the **upperBound** will be `-1` (unbounded).
- If a UML association is navigable in both directions, the **eOpposite** of the **EReference** for one end of an association will be the **EReference** that corresponds to the other end. If a UML association can only be navigated in one direction, the **eOpposite** for the corresponding **EReference** will be `null`.
- The **containment** attribute for an **EReference** is set to `true` whenever the corresponding UML association end is an aggregate and the containment of the target class is “by value”. This is illustrated in a class diagram by a solid diamond on the container side of the association.

- By default, the **transient**, **volatile**, **unique**, and **unsettable** attributes are all `false` and the **resolvableProxies** and **changeable** attributes are `true`. There is no way to specify these attributes in standard UML.

Bidirectional, Non-Containment References

The name and multiplicity of an **EReference** are derived from the role name and multiplicity of its corresponding UML association end. An **EReference** is owned by the class that is on the opposite end of the line from where its role name and multiplicity appear, while the **eType** is the class at the same end as the name and multiplicity. If an association is navigable in both directions, the **eOpposite** reference will be set to the **EReference** associated with the other association end. For example, consider the following association from the ExtendedPO2 model (Figure 8.11):



Figure 8.11. An association that is navigable in both directions.

This UML association is bidirectional and therefore maps to two **EReferences** in Ecore:

- The **EReference** named “customer” that belongs to the **EClass** for “PurchaseOrder”. Its **eType** is the **EClass** for “Customer”, its **lowerBound** and **upperBound** attributes are both `1`, and its **containment** attribute is `false`. The **eOpposite** reference is set to the **EReference** named “orders” that belongs to the “Customer” **EClass**.
- The **EReference** named “orders” that belongs to the “Customer” **EClass**. Its **eType** is the “PurchaseOrder” **EClass**. It has a **lowerBound** of `0`, an **upperBound** of `-1` (unbounded), and the **containment** attribute is `false`. Its **eOpposite** reference is set to the **EReference** named “customer” that belongs to the “PurchaseOrder” **EClass**.

As you can see, each of the two references has the other as its **eOpposite**.

Containment References

If an association is marked with a solid diamond in the UML (that is, if it is a by-value aggregation association), then the **containment** attribute of the corresponding **EReference** will be set to `true` (Figure 8.12).



Figure 8.12. A by-value aggregation.

Here we have a single **EReference** named “orders”, which belongs to the “Supplier” **EClass**. Its **eType** is the **EClass** for “PurchaseOrder”. The **lowerBound** is `0`, the **upperBound** is `-1` (unbounded), and the **eOpposite** is `null`. The **containment** attribute is `true`.

Map References

A containment reference whose target UML class has a stereotype of <<MapEntry>> will be treated specially by the EMF code generator. It will be implemented using an `EMap` instead of just an `EList`. `EMap` is an `EList`-derived interface for a map whose keys and/or values are `EObjects`. Entries in an `EMap` are type-safe and ordered.

The way to specify a map in UML is to create a class that represents a map entry type. Specifically, you create a UML class that has a stereotype of <<MapEntry>> and contains two features named “key” and “value”, which can be either attributes or references. Any EMF class can then have a containment reference to the map entry type. For example, consider the class `Index` in the following diagram (Figure 8.13):

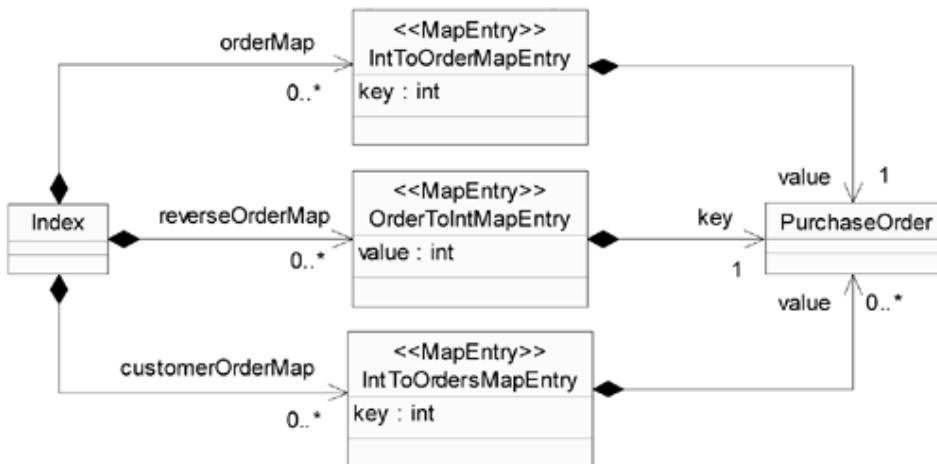


Figure 8.13. UML classes representing map types.

The `EClass` named “`Index`” has three map-type `EReferences` named “`orderMap`”, “`reverseOrderMap`”, and “`customerOrderMap`”. The `orderMap` reference is a map from integer identifiers to purchase order objects. Its map entry class, `IntToOrderMapEntry`, has an attribute `key` of type `int`, and a `value` that is a reference to the class of the desired type, `PurchaseOrder`. Conversely, `reverseOrderMap` is a map of objects back to their identifiers. It uses a map entry class, `OrderToIntMapEntry`, whose `key` is a reference to the `PurchaseOrder` class and whose `value` is an `int` attribute.

The `key` and `value` features can also be multi-valued. The `customerOrderMap` reference is an example. Its map entry class, `IntToOrdersMapEntry`, uses a `int` attribute `key`, but its `value` is a multiplicity-many reference to a group of purchase orders.

Because either or both of the `key` and `value` features of an EMF map entry can be attributes, it's often more natural to think of the map itself as an attribute of the class that it's contained in. For this reason, a map can alternatively be declared in UML as an attribute whose type is a UML map entry class. The `details` attribute in the `EAnnotation` class, in Ecore itself, is an example (Figure 8.14):



Figure 8.14. A map shown as an attribute.

In this case, even though **details** looks to be defining an **EAttribute**, it's really just an alternative syntax in UML. In Ecore, this will map to a containment-type **EReference** to the map entry class, **EStringToStringMapEntry**, just as in the previous examples.

References with Ecore Properties

As mentioned previously, the **transient**, **volatile**, **unique**, **unsettable**, **resolveProxies**, and **changeable** attributes of an **EReference** cannot be expressed in UML. For example, consider the **shippedOrders** reference in class **Supplier** (Figure 8.15):



Figure 8.15. The computable **shippedOrders** association.

In the UML diagram, it looks like a simple one-way association, the **EReference** named “**shippedOrders**”. Its **eType** is the **EClass** “**PurchaseOrder**”, its **lowerBound** is **0**, **upperBound** is **-1** (unbounded), **eOpposite** is **null**, and its **containment** attribute is **false**.

In the model, we would also like to specify that the reference is **transient**, **volatile**, and not **changeable** and that the **resolveProxies** attribute should be **false**. Since these properties are not part of UML we need to set them using a tool-specific technique. In Rose, we would use the **Ecore A** or **Ecore B** page (depending on which end of the association we're changing) of the **Association Specification** dialog, which is available when using the **ecore.pty** property file with your model. Properties corresponding to the non-UML attributes can be set on this page, as shown in Figure 8.16.

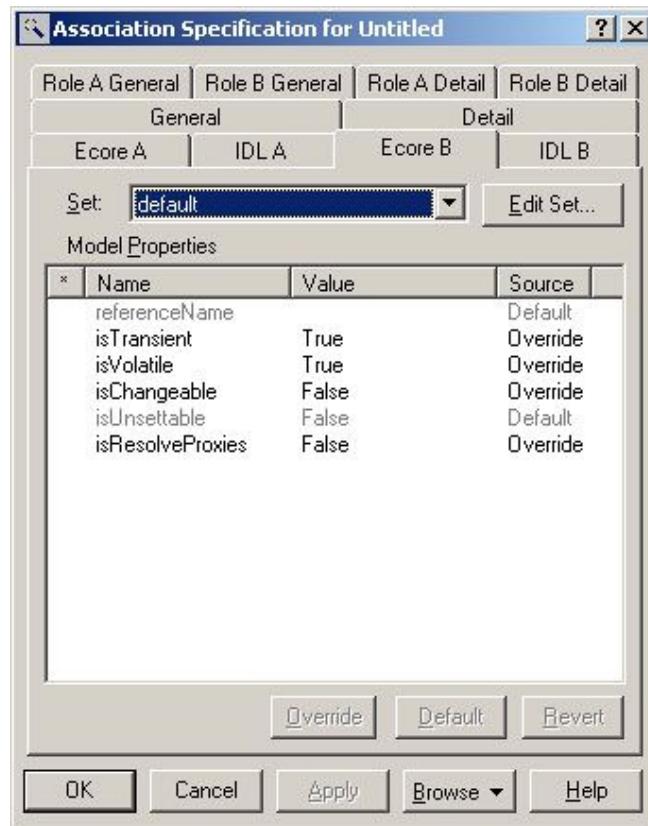


Figure 8.16. Setting non-UML reference properties with Rational Rose.

Now, the **EReference** for “shippedOrders” will have the **transient** and **volatile** attributes set to **true**, and the **changeable** and **resolveProxies** attributes will be **false**.

UML Specification for Operations

Each operation in a UML class is mapped to an **EOperation** of the corresponding **EClass**.

- The **name** of each **EOperation** is the same as the name of the UML operation.
- The **eType** of each **EOperation** is the return type of the UML operation. The return type can be either a basic Java type, or a UML class corresponding to an **EEnum**, **EDataType**, or **EClass**.
- The **eParameters** of each **EOperation** is a collection of **EParameters** where each **EParameter** is derived from the corresponding parameter specified in the UML operation. The **name** of each parameter is set to the UML parameter name. The **eType** of each parameter is set to the **EClassifier** corresponding to the UML parameter type, which can be either a basic Java type, or a UML class corresponding to any **EEnum**, **EDataType**, or **EClass**.

For example, consider the following UML operation definition (Figure 8.17):

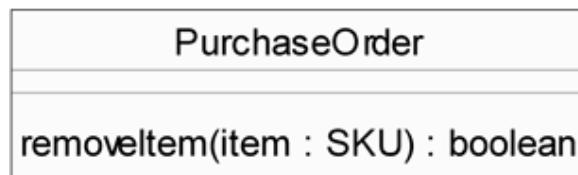


Figure 8.17. An operation.

This will result in an **EOperation** named “removeItem”. The operation's **eType** is the built-in data type **EBoolean** (that is, the predefined Java type `boolean`), and it has a single **EParameter** whose **name** is “item” and whose **eType** is the **EDataType** named “SKU”.

Part III. Using the EMF Generator

Chapter 9. EMF Generator Patterns

Arguably, the most valuable feature of EMF is its ability to generate high-quality implementation code. As a carefully written and well-tested base for application development, generated EMF code both speeds the development process and helps improve software quality. This model code includes type-safe interfaces and efficient implementations for creating, modifying, saving, and loading instances of modeled classes.

This chapter explores the various patterns EMF uses in generating Java code from the elements of a model. We will discuss how modeling decisions affect the generated code, and give insights into how to design a model most effectively. To illustrate the code generation patterns, we will begin by drawing on features from the primer purchase order model, PrimerPO. However, as we discuss more advanced features, which are not included in this model, we will borrow from two extended versions of it, ExtendedPO1 and ExtendedPO2. A summary of all the example models can be found in Appendix B.

As we have seen in Chapters 5, 6, 7, and 8, a model can be specified in several ways. The examples in this chapter will be discussed in terms of Ecore and accompanied by UML diagrams that illustrate the model constructs. The same code would be generated if we used annotated Java source or XML Schema to specify an equivalent model.

The code generated to implement an Ecore package is organized into two Java packages: one contains the set of Java interfaces that together represent the client interface to the model, while the other contains corresponding implementation classes. A third optional utility package can be created containing a switch class and an adapter factory, which can be used to attach adapters to the model classes.

In general, an interface and an implementation class are generated for each class in the model; enumerations are implemented by a single class. In addition, an interface and an implementation are created for the package itself, and for a factory that creates instances of the classes defined in it.

In addition to Java code, the EMF generator also produces two other files that support the use of the model code as an Eclipse plug-in. As we described in Chapter 1, a plug-in manifest file, called *plugin.xml*, identifies the model plug-in to the platform and declares its interconnections to other plug-ins. A property file, *plugin.properties*, contains translatable strings for use by the manifest file.

Modeled Classes

Classes are the essential elements of any EMF model. As we saw in Chapter 5, we model a class in Ecore with an instance of **EClass**. The generated code for a class defines a number of methods that provide access to the attributes and references belonging to the class, as well as methods corresponding to its operations. Details of the generated code depend not only on these structural and behavioral features, but also on the attributes of the **EClass** itself.

Interfaces and Implementation Classes

For each class in a model, the EMF code generator usually produces both a Java interface and an implementation class. Consider the **PurchaseOrder** class in the PrimerPO model. For the moment, let's ignore the attributes and references that belong to it, which leaves us with just the following class (Figure 9.1):

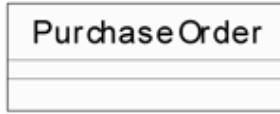


Figure 9.1. A simple model class.

The generated interface for **PurchaseOrder** looks like this:

```

public interface PurchaseOrder extends EObject
{
    ...
}
  
```

Because the **PurchaseOrder** model class has no explicit base class, the generated **PurchaseOrder** interface extends the **EObject** interface. **EObject** is the EMF equivalent of `java.lang.Object`, that is, the base of every EMF class. **EObject** and its corresponding implementation class **EObjectImpl** provide a lightweight base for **PurchaseOrder**'s participation in EMF, supporting reflection, notification, and persistence.

The implementation class for **PurchaseOrder** is called **PurchaseOrderImpl**, and it is declared as follows:

```

public class PurchaseOrderImpl extends EObjectImpl
    implements PurchaseOrder
{
    ...
}
  
```

We will discuss how the use of inheritance affects the interface and class declarations in Section 9.5.

Accessor Methods

The code that is generated for a class includes accessor methods that allow us to get and set values for each of the various attributes and references that belong to the class.

Recall that the **PurchaseOrder** class includes two simple attributes: **comment** of type **String**, and **orderDate** of type **Date**. Among others, it also includes a reference, **items**, to class **Item** (Figure 9.2):

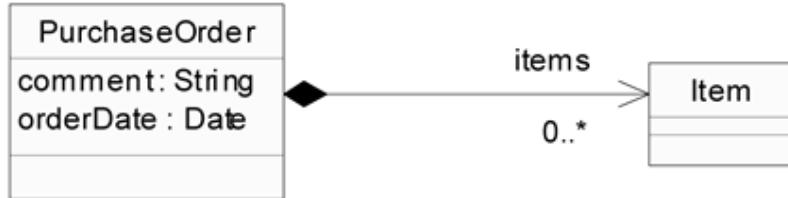


Figure 9.2. A class with attributes and references.

The generated interface for **PurchaseOrder** looks like this:

```

public interface PurchaseOrder extends EObject
{
    String getComment();
    void setComment(String value);
  
```

```

Date getOrderDate();
void setOrderDate(Date value);

EList getItems();
}

```

This interface contains accessor methods that permit access to and modification of the value of each modeled attribute and reference. One or more methods are generated, depending on the specifics of the attribute or reference involved. At a minimum, this includes a method that gets the value. Depending on whether the structural feature is changeable and unsettable, you may also see methods that set the value, unset the value, and check whether the value has been set.

Each generated implementation class includes implementations of the accessor methods defined in the corresponding interface, plus some other methods required by the EMF framework. Class `PurchaseOrderImpl` implements the accessors for `comment`, `orderDate`, and `items`:

```

public class PurchaseOrderImpl extends EObjectImpl
    implements PurchaseOrder
{
    public String getComment() {
        ...
    }

    public void setComment(String newComment) {
        ...
    }

    public Date getOrderDate() {
        ...
    }

    public void setOrderDate(Date newOrderDate) {
        ...
    }

    public void EList getItems() {
        ...
    }

    // etc...
}

```

Again, the selection of method implementation patterns depends on the definitions of the modeled features. Sections 9.2 and 9.3 describe all the important accessor patterns and how Ecore modeling constructs map to and control them.

Abstract Classes

As in Java, an Ecore class is made abstract to forbid its own instantiation, while allowing other classes to extend it. An abstract class is modeled by an `EClass` with its `abstract` attribute set to `true`. This is reflected in the generated code in two ways. The first and most obvious impact is that the `class` statement for the implementation class includes the `abstract` keyword. The other is that the generated factory interface, which we will discuss in Section 9.7, does not include a `create()` method for the class.

Interfaces

Rather than generating an implementation for every class in a model, it may be desirable to only create Java interfaces for some classes and let any classes that inherit from them implement the inherited features, themselves. Such an interface-only class is modeled by an **EClass** with its **interface** attribute set to `true` and no **instanceClass** specified.

If an **instanceClass** is specified, then a Java interface will not be generated, either. Instead, the class will be considered a proxy for an external interface that already exists, in the same way that data types, which we discuss in Section 9.2.2, are used as proxies for external Java classes. This technique is useful for adding external Java interfaces to a generated interface's `extends` clause, as we will see in Section 9.5.3.

Like the abstract classes described in the previous section, interfaces cannot be instantiated, so the generator does not include a `create()` method in the factory interface.

Attributes

As described above, the generated interface and implementation class that correspond to an **EClass** include one or more accessor methods for each **EAttribute** that the **EClass** contains. The number and nature of these accessors depends upon the type of the attribute, as defined by the **EAttribute's eType**, as well as the values of several of the **EAttribute's** own attributes. They specify whether an attribute can contain multiple values, what the default value of an attribute will be, whether to provide storage for the attribute's value, whether it can be changed and whether it can be unset. The following subsections describe how these settings affect the generated code patterns.

Simple Attributes

We describe an attribute as *simple* when it is single-valued and its type is a Java language data type, like `int`, `boolean`, or `String`, that is defined by Ecore.¹ Our purchase order example includes several simple attributes. Let's look at the **comment** attribute in class **PurchaseOrder** (Figure 9.3):



Figure 9.3. A simple attribute.

When we generate code for this class, `get()` and `set()` methods for **comment** are included in the **PurchaseOrder** interface:

```

String getComment();
void setComment(String value);
  
```

Note that the generator departs from this naming pattern only for attributes with `boolean` type; to follow the convention for Java bean properties, the name of the first accessor for such an attribute begins with "is" rather than "get".

We now turn to the implementation of simple attributes. The **comment** attribute is implemented by the following code that is generated in **PurchaseOrderImpl**:

¹These types are modeled in Ecore by **EInt**, **EBoolean**, and **EString**, respectively.

```

protected static final String COMMENT_EDEFAULT = null;
protected String comment = COMMENT_EDEFAULT;

public String getComment() {
    return comment;
}

public void setComment(String newComment) {
    String oldComment = comment;
    comment = newComment;
    if (eNotificationRequired())
        eNotify(new ENotificationImpl(this, Notification.SET,
                                      PPOPackage.PURCHASE_ORDER_COMMENT,
                                      oldComment, comment));
}

```

The generated `get()` method is optimally efficient. It simply returns an appropriately typed instance variable that represents the attribute.

The `set()` method is slightly more complicated. In addition to setting the `comment` instance variable, it needs to send change notification to any observers that may be listening to the object by calling the `eNotify()` method. The notification object, an instance of `ENotificationImpl`, specifies the notifier (`this`), the type of notification (set), the structural feature involved,² and the old and new values.

To optimize the case where there are no observers (for example, in a batch application), the construction of the notification object and the call to `eNotify()` are guarded by an `eNotificationRequired()` test. The default implementation of `eNotificationRequired()` simply checks if there are any adapters³ attached to the object. Therefore, when EMF objects are used without observers, the call to `eNotificationRequired()` amounts to nothing more than an efficient `null` pointer check. Considering that this can be inlined by a JIT compiler, there is very little overhead for the benefit of being able to participate in the EMF notification framework.

Data Type Attributes

As mentioned previously, all the classes generated from an EMF model, such as `PurchaseOrder` and `Item`, derive from the EMF base class `EObject`. However, models may also refer to classes that are not `EObjects`. For example, assume we want to add an attribute of type `java.util.Date` to a class in our purchase order model. Before we can do that, we need to define an Ecore data type, modeled by `EDataType`, to represent the external type. In UML, we use a class with the `<<datatype>>` stereotype for this purpose (Figure 9.4):

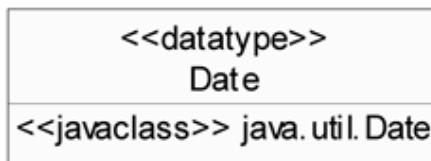


Figure 9.4. A data type definition.

²`PPOPackage.PURCHASE_ORDER_COMMENT` is the feature ID for the `comment` attribute. We describe feature IDs, which are defined by the generated package interface, in Section 9.6.1.

³Recall from Chapter 2 that EMF refers to all observers as adapters because the same `Adapter` interface is used to listen for notifications and to extend the behavior of a type without subclassing it.

A data type is simply a named element in the model that acts as a proxy for the actual Java class. The Java class is specified by the data type's **instanceClass** attribute; it is represented in UML by an attribute with the `<<javaclass>>` stereotype, whose name is the fully qualified Java class name. Attributes of type **Date**, like the **orderDate** attribute in class **PurchaseOrder**, can then be defined like this (Figure 9.5):

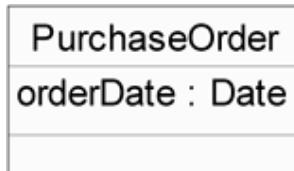


Figure 9.5. An attribute use of a data type.

When we generate code for this class, the accessors for **orderDate** appear like this in the **PurchaseOrder** interface:

```

Date getOrderDate();
void setOrderDate(Date value);
  
```

As you can see, **orderDate** is treated like any other attribute; the generated implementations of these methods are also just like those of simple attributes. As we saw in Chapter 5, all data types, including the Java language types like `int`, `boolean`, and `String`, are modeled in Ecore as instances of **EDataType**. The only special thing about the language types is that their corresponding data types are predefined in the Ecore model, so they don't need to be redefined in every model that uses them.

A data type definition has one other effect on the generated code. Since data types represent some arbitrary class, a generic serializer and parser, like the default XMI serializer, has no way of knowing how to save the state of an attribute of that type. Should it call `toString()`? That's a good default, but the EMF framework requires more flexibility, so it generates two more methods for every data type that is defined in a modeled package:

```

/**
 * @generated
 */
public String convertDateToString(EDataType eDataType,
                                  Object instanceValue) {
    return super.convertToString(eDataType, instanceValue);
}

/**
 * @generated
 */
public Date createDateFromString(EDataType eDataType,
                                  String initialValue) {
    return (Date)super.createFromString(eDataType, initialValue);
}
  
```

These methods convert an instance of the data type into a string and back. They are included in the implementation of the factory that is generated for the package to support EMF's standard factory mechanism for the serialization of data type values, which we will discuss in Section 9.7.

The generated implementations of these methods simply delegate to their superclass for default, but not particularly efficient, implementations: `convertToString()` just calls `toString()` on the instance, but `createFromString()` tries, using Java reflection, to call a string constructor or, failing that, a static `valueOf()` method. Typically, you should replace these implementations with something more appropriate:

```
/**
 * @generated
 */
public String convertDateToString(EDataType eDataType,
                                  Object instanceValue) {
    return instanceValue.toString();
}
```

Note that the `@generated` Javadoc comment preceding the method has been removed. As you will see in Section 9.9, this preserves the changes you make to this method should you ever regenerate the code. We will look more closely at how to properly implement data types, including this Date example, in Chapter 12.

Enumeration Attributes

An enumerated type is defined by an explicit list of values that it may possibly take, called literals. Enumerated types are commonly used as the types of attributes that represent status or kind and are implemented in EMF using the Type-safe Enum pattern, described by Joshua Bloch in *Effective Java Programming Language Guide* [6].⁴

ExtendedPO1 introduces an enumeration attribute called `status` to `PurchaseOrder` as follows (Figure 9.6):

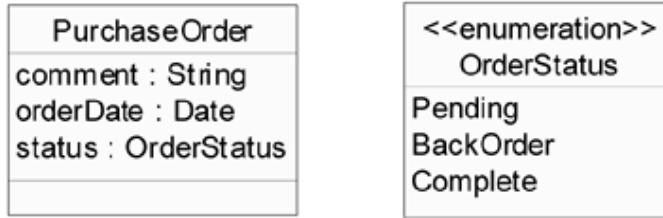


Figure 9.6. The `status` enumeration attribute.

The following accessors are produced for `status` in the `PurchaseOrder` interface:

```
OrderStatus getStatus();
void setStatus(OrderStatus value);
```

As you can see, the generated methods use a type-safe enumeration class called `OrderStatus`. This class defines static constants for the enumeration's values and other convenience methods like this:

```
public final class OrderStatus extends AbstractEnumerator
{
    public static final int PENDING = 0;
    public static final int BACK_ORDER = 1;
    public static final int COMPLETE = 2;

    public static final OrderStatus PENDING_LITERAL =
        new OrderStatus(PENDING, "Pending");
    public static final OrderStatus BACK_ORDER_LITERAL =
        new OrderStatus(BACK_ORDER, "BackOrder");
    public static final OrderStatus COMPLETE_LITERAL =
        new OrderStatus(COMPLETE, "Complete");

    public static OrderStatus get(String name) {
        ...
    }
}
```

⁴This pattern is also discussed in the August 7, 2001, edition, of Sun's *JDC Tech Tips*, available at <http://developer.java.sun.com/developer/JDCTechTips/2001/tt0807.html#tip2>.

```

public static OrderStatus get(int value) {
    ...
}

private OrderStatus (int value, String name) {
    super(value, name);
}
}

```

As shown, the enumeration class includes static `int` constants for the enumeration's values as well as static constants for the enumeration's literal objects. The `int` constants have the same names as the model's literal names.⁵ The literal constants have the same names with `_LITERAL` appended.

These constants provide convenient access to the literals when, for example, setting the status of a purchase order:

```
po.setStatus(OrderStatus.PENDING_LITERAL);
```

Notice that the constructor for `OrderStatus` is private. Therefore, the only instances of the enumeration class that will ever exist are the ones used for the static fields `PENDING_LITERAL`, `BACK_ORDER_LITERAL`, and `COMPLETE_LITERAL`. As a result, equality comparisons using `equals()` are never needed. Literals can always be reliably compared using the simpler and more efficient `==` operator, like this:

```
po.getStatus() == OrderStatus.PENDING_LITERAL
```

When comparing against many values, a `switch` statement using the `int` values is better yet:

```

switch (po.getStatus().value()) {
    case OrderStatus.PENDING:
        // do something...
        break;
    case OrderStatus.BACK_ORDER:
        ...
}

```

For situations where only the literal name or value is available, the enumeration class also includes convenient `get()` methods that can be used to retrieve the corresponding literal object.

Multi-Valued Attributes

In some cases, it is appropriate for an attribute to hold several values rather than a single one. For example, the ExtendedPO2 model introduces a class to model addresses outside of the United States, called **GlobalAddress**. In order to accommodate the variety of address formats used globally, this class includes an attribute, called **location**, whose value is an arbitrary number of strings. In Ecore terms, the **EAttribute** has a **lowerBound** of zero and unbounded **upperBound**. In UML, we indicate the bounds in a stereotype on the attribute (Figure 9.7).

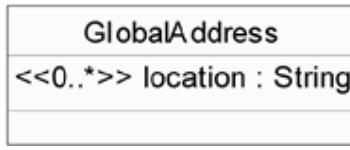


Figure 9.7. A multi-valued attribute.

The generated `GlobalAddress` interface includes a single `get()` accessor that returns a list:

⁵To conform to proper Java programming style, the static constant names are converted to uppercase if the modeled enumeration's literal names are not already uppercase.

```
EList getLocation();
```

Notice that this method returns an `EList` as opposed to a `java.util.List`. Actually, the two are almost the same: `EList` is an EMF subclass of `List`, only adding a `move()` method to the API. From a client perspective, you can consider it a standard Java list.

The generated implementation in `GlobalAddressImpl` looks like this:

```
protected EList location = null;

public EList getLocation() {
    if (location == null) {
        location = new EDataTypeUniqueList(String.class, this,
                                         EPO2Package.GLOBAL_ADDRESS__LOCATION);
    }
    return location;
}
```

The `EDataTypeUniqueList` framework utility class implements a list that may contain only elements of the specified type—in this case, `String.class`—and prevents a single value from occurring in it more than once. It also automatically participates in EMF's notification framework, sending change notifications using the supplied feature ID, `EPO2Package.GLOBAL_ADDRESS__LOCATION`. We will discuss feature IDs further in Section 9.6.1.

If we did not want to enforce the restriction that each object in the attribute must be unique, we would set the `unique` attribute of the `EAttribute` to `false`, and `EDataTypeList` would be used as the `EList` implementation.

An alternate method for defining a multi-valued attribute uses an array data type. In this approach, we would define the `location` attribute to be single-valued, but of type `StringArray`. We would also need to define this data type as representing `String[]` (Figure 9.8):



Figure 9.8. An array type attribute.

Using this approach, the `location` accessors in the generated `GlobalAddress` interface would look like this:

```
String[] getLocation();
void setLocation(String[] value);
```

Their implementation in `GlobalAddressImpl` would use the standard simple attribute pattern.

Having defined a new data type, we would also need to implement the generated factory's `createFromString()` and `convertToString()` methods, in order to serialize its instances. Note that a string form of this data type would have to be a concatenation of the items in the array, with some sort of delimitation—essentially imposing structure on the serialized form, something that is usually handled by the resource. This approach also thwarts the notification framework, sending notifications only when the reference to the whole array is changed—when growing the array, for example—and not when items in it are mutated. For these reasons, the array approach is not often used for object types like `String`. In general, EMF's normal list-based mechanism for multi-valued attributes would be preferable. Array data types are, however, more appropriate for primitive Java types, such as `char[]` or `byte[]`.

Default Values

All single-valued attributes have well-defined default values. The default value for each attribute appears in the generated code as an appropriately typed, static, final field whose name is constructed by capitalizing the attribute name and appending to it the “_EDEFAULT” suffix.

The default value can always be specified in the model, but if one is not explicitly chosen, EMF automatically picks a value appropriate to the type of the attribute. For Java language types, this is the default defined by Java: `false` for `boolean`, zero for numeric primitive types, and `null` for object types. For an enumerated type, it is the first literal value that it defines.

We have already seen an example of this in Section 9.2.1, where the default for the `comment` attribute on the `PurchaseOrder` class was automatically chosen to be `null`. We saw that the `PurchaseOrderImpl` implementation class includes the following:

```
protected static final String COMMENT_EDEFAULT = null;
```

If we want to override this choice, we can update the model to specify an explicit default, as follows (Figure 9.9):

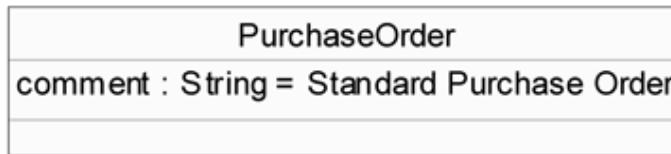


Figure 9.9. An attribute with default value.

Then, the declaration of `COMMENT_EDEFAULT` would look like this:

```
protected static final String COMMENT_EDEFAULT =
    "Standard Purchase Order";
```

For non-string simple attributes, this generated initialization statement uses the same mechanism as the Ecore factory to convert the specified string into an appropriate primitive value or object. For attributes whose type is a user-modeled data type, this initialization statement actually calls the generated factory, so as to use whatever mechanism it eventually defines.

Volatile Attributes

As we have seen, the default implementation for attributes uses a field to hold its value at runtime. This is because, by default, attributes are non-volatile; a volatile attribute is one for which no storage is set aside at runtime.

In ExtendedPO2, the `PurchaseOrder` class includes an attribute called `totalAmount`, which represents the total value of all the items in the order (Figure 9.10):



Figure 9.10. The computable `totalAmount` attribute.

This amount can easily be calculated by taking the sum of the costs of all the items associated with it. Therefore, we declare the **totalAmount** attribute to be volatile, indicating that we will compute it whenever it is needed. This saves storage because we don't need to store the value anywhere, and it also avoids the complication of having to update the value every time we add to or remove from the **items** list or modify an **Item** in it.

Because the **EAttribute**'s **volatile** attribute is set to `true`, the generated implementation class does not include a field for **totalAmount**. Also, its accessor methods are stubs that need to be implemented by hand to provide the desired behavior. In order to indicate that the implementation is missing, the generated stubs throw an `UnsupportedOperationException`. The `getTotalAmount()` method looks like this:

```
/**  
 * @generated  
 */  
public int getTotalAmount() {  
    // TODO: implement this method to return the  
    // 'Total Amount' attribute  
    throw new UnsupportedOperationException();  
}
```

We are now expected to remove the `@generated` tag and replace this implementation with one that does the appropriate calculation. We will look at how this can be done in Chapter 12.

Non-Changeable Attributes

By default, every attribute is changeable, which means that its value can be set externally. For single-valued attributes, this means that a `set()` method is generated. We can suppress the generation of this method by setting the **changeable** attribute of an **EAttribute** to `false`.

Often, volatile attributes should be non-changeable as well. A good example is the **totalAmount** attribute discussed in the previous section. Since its value is computed from the values of several other structural features, it doesn't make much sense to be able to set it explicitly. We make it non-changeable, and no `setTotalAmount()` method is generated in either the interface or the implementation class.

Note that the values of multi-valued attributes can always be changed using their list interface. Since `set()` methods are never generated for multi-valued attributes, making such an attribute non-changeable has no effect on code generation.

Unsettable Attributes

The set of valid values for an attribute is determined by its type. This set may be limited to a large but finite number of primitive values or a relatively small number of enumerated type literals, or it may include `null` and a conceptually unbounded number of object references.⁶

In any case, it is sometimes desirable to allow attributes to be unset—that is, to have no value set. This state is distinguishable from that of being set to any other value, including the attribute's default value. Such attributes are modeled in Ecore by an **EAttribute** with **unsettable** set to `true`.

For example, consider the **Item** class (Figure 9.11):

⁶In reality, the number of possible objects is limited by the finite amount of memory available.

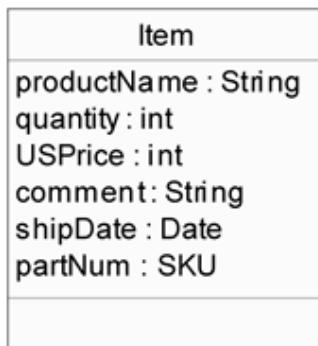


Figure 9.11. The **Item** class and its attributes.

Suppose that we wish to distinguish between an item that has not yet been shipped and one that could not be shipped because it has been discontinued. We can do this by making **shipDate** unsettable. Before an item has been shipped, its **shipDate** will be unset. If it cannot be shipped, we will set its value to `null`.

The following additional methods are included in the `Item` interface to provide the unsettable functionality of the attribute:

```

void unsetShipDate();
boolean isSetShipDate();
  
```

The `unset()` method changes the attribute's value back to its default and marks the attribute as being in the unset state. The `isSet()` method tests whether the attribute has been set since it was created or last unset.

These methods are implemented in `ItemImpl` like this:

```

protected boolean shipDateESet = false;

public void unsetShipDate() {
    Date oldShipDate = shipDate;
    boolean oldShipDateESet = shipDateESet;
    shipDate = SHIP_DATE_EDEFAULT;
    shipDateESet = false;
    if (eNotificationRequired())
        eNotify(new ENotificationImpl(this, Notification.UNSET, ... ,
                                      oldShipDateESet));
}

public boolean isSetShipDate() {
    return shipDateESet;
}
  
```

A boolean field is used to keep track of whether the attribute is set. It is tested by the `isSet()` method and set to `false` by `unset()`. The `set()` pattern is also changed slightly to set this field to `true`:

```

public void setShipDate(Date newShipDate) {
    Date oldShipDate = shipDate;
    shipDate = newShipDate;
    boolean oldShipDateESet = shipDateESet;
    shipDateESet = true;
    if (eNotificationRequired())
        eNotify(new ENotificationImpl(this, Notification.SET, ... ,
                                      !oldShipDateESet));
}
  
```

Notice that unsetting an attribute sends a change notification to any registered listeners in exactly the same manner as setting it. In both of these cases, the `ENotificationImpl` instance is created with an additional constructor argument, indicating whether the “unset state” has changed. The notification’s `isTouch()` method is affected by this value.

Multi-valued, unsettable attributes are implemented using subclasses of their usual `EList` implementations that provide `unset()` and `isSet()` methods. Unsetting a multi-valued attribute clears its values and marks it as unset. Adding back any values marks it as set.

Making an attribute unsettable necessarily incurs a storage cost: the size of an instance of the generated class is increased by the extra field that tracks whether the attribute is set.

References

As described in Section 9.1.2, the interface and implementation class that are generated for an `EClass` include at least one accessor method for each of its `EReferences`. The number and nature of these accessors is determined by the `eType` of the reference—that is, the `EClass` being referenced—as well as the presence or absence of an opposite reference and the values of several of the `EReference`’s own attributes. These attributes specify the multiplicity of the reference, whether the reference represents a containment relationship, and whether it should automatically resolve object proxies. In addition, several additional attributes affect the choice of code patterns for a reference in exactly the same way as for an attribute, including those that determine whether to provide storage for its value, whether it can be changed, and whether it can be unset. The following subsections detail the effects of these settings on the generated code patterns.

One-Way References

In ExtendedPO1, a reference is introduced to link a purchase order to a related previous order. This one-way reference, named `previousOrder`, belongs to the `PurchaseOrder` class, and its type is also `PurchaseOrder` (Figure 9.12):

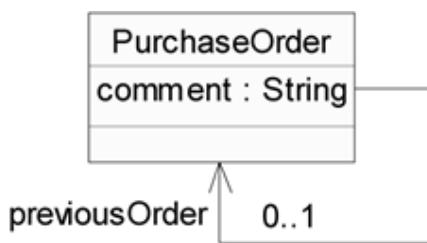


Figure 9.12. A simple one-way reference.

The `PurchaseOrder` interface that is generated for this class includes the following `get()` and `set()` methods:

```

PurchaseOrder getPreviousOrder();
void setPreviousOrder(PurchaseOrder value);
  
```

Because the `previousOrder` reference is one-way, the implementation of the `setPreviousOrder()` method looks much like the simple `set()` methods for attributes that we saw earlier:

```

protected PurchaseOrder previousOrder = null;

public void setPreviousOrder(PurchaseOrder newPreviousOrder) {
    PurchaseOrder oldPreviousOrder = previousOrder;
    ...
}
  
```

```

previousOrder = newPreviousOrder;
if (eNotificationRequired())
    eNotify(new ENotificationImpl(this, ...));
}

```

First, notice that there is no static final field specifying a default value—the default value of every reference is `null`.

Beyond that, the only difference from, say, `getComment()` is that the type of the reference is also a modeled class, so we are passing a pointer to another `EObject`. The implementation of `getPreviousOrder()` is a little more complicated, however, because the previous order may be serialized in a different resource from the referencing order.

Because the EMF persistence framework uses a lazy loading scheme, the resource containing the referenced object may not have been loaded when the accessor is first invoked. In such a case, an object pointer like `previousOrder` would refer to a proxy for the object, instead of the object itself. In order to handle this possibility, `getPreviousOrder()` looks like this:

```

public PurchaseOrder getPreviousOrder() {
    if (previousOrder != null && previousOrder.eIsProxy()) {
        PurchaseOrder oldPreviousOrder = previousOrder;
        previousOrder =
            (PurchaseOrder)EcoreUtil.resolve(previousOrder, this);
        if (previousOrder != oldPreviousOrder) {
            if (eNotificationRequired()) eNotify(
                new ENotificationImpl(this, Notification.RESOLVE, ...));
        }
    }
    return previousOrder;
}

```

Instead of simply returning the `previousOrder` instance variable, we first call the `EObject`'s `eIsProxy()` method to check if the reference is a proxy, and then call the static framework method `EcoreUtil.resolve()` if it is. This method attempts to load first the referenced object's document and then the object itself, using the URI specified by the proxy. If successful, it returns the resolved object. If, however, the document fails to load, it just returns the proxy again.⁷

In some circumstances, you may be able to optimize the `get()` method by omitting the code that resolves proxies. This would result in a simpler `get()` method, just like `getComment()`. We will discuss this optimization in Section 9.3.5.

Bidirectional References

In the previous section, we saw how proxy resolution affects the `get()` pattern for one-way references. Now we can look at how the `set()` pattern changes when an association is bidirectional. Consider the bidirectional association between ExtendedPO1's `PurchaseOrder` and `Customer` classes (Figure 9.13):



Figure 9.13. A bidirectional association.

⁷Document loading and proxy resolution are discussed in detail in Chapter 13.

The absence of an arrowhead on either end of the association line indicates its bidirectionality. In Ecore, this is modeled as a reference from **PurchaseOrder** to **Customer** named **customer** and another reference from **Customer** to **PurchaseOrder** named **orders**. These two **EReferences** have each other as their **eOpposites**.

In the interfaces generated for the two classes, we find the same accessors that we would if these were two independent, one-way references. In **PurchaseOrder**, for example, we find the following:

```
Customer getCustomer();
void setCustomer(Customer value);
```

The implementation of `getCustomer()` in `PurchaseOrderImpl` follows exactly the same pattern as the one-way `getPreviousOrder()` method described in the previous section, but `setCustomer()` looks quite different:

```
public void setCustomer(Customer newCustomer) {
    if (newCustomer != customer) {
        NotificationChain msgs = null;
        if (customer != null) msgs =
            ((InternalEObject)customer).eInverseRemove(this, ..., msgs);
        if (newCustomer != null) msgs =
            ((InternalEObject)newCustomer).eInverseAdd(this, ..., msgs);
        msgs = basicSetCustomer(newCustomer, msgs);
        if (msgs != null) msgs.dispatch();
    }
    else if (eNotificationRequired()) // send "touch" notification
        eNotify(new ENotificationImpl(this, Notification.SET, ...));
}
```

As you can see, when setting a bidirectional reference like **customer**, the other end of the reference needs to be set as well, by calling `eInverseAdd()`. Notice that the **customer** reference is modeled with multiplicity-1, meaning that a purchase order can only have one customer. As a result, this purchase order cannot be in more than one customer's **orders** reference, so we also need to remove the inverse of any previous customer by calling `eInverseRemove()`. Finally, we set the **customer** reference itself by calling another generated method, `basicSetCustomer()`, which looks like this:

```
public NotificationChain basicSetCustomer(Customer newCustomer,
                                         NotificationChain msgs) {
    Customer oldCustomer = customer;
    customer = newCustomer;
    if (eNotificationRequired()) {
        ENotificationImpl notification =
            new ENotificationImpl(this, Notification.SET, ... );
        if (msgs == null) msgs = notification;
        else msgs.add(notification);
    }
    return msgs;
}
```

This method looks very similar to the one-way reference `set()` method, except that the notification gets added to the `msgs` argument, instead of being fired directly. Between the `eInverseAdd()`, the `eInverseRemove()` and this `basicSet()`, `setCustomer()` can generate three different notifications. In other situations, this could be as many as four. A `NotificationChain` is used to collect all these individual notifications so their firing can be deferred until after all the state changes have been made. The queued-up notifications are sent by calling `msgs.dispatch()`, as shown in the `setCustomer()` method, above.

You may be wondering why we bother to delegate to a `basicSet()` method at all, when we could just inline the code in the `set()` method. The reason is that it's also needed by the `eInverseAdd()` and `eInverseRemove()` methods, which we will look at in Sections 9.6.6 and 9.6.7, respectively.

Finally, notice that if ever we set the reference to its existing value, this implementation will avoid the overhead of performing the reverse remove and add, and of creating a `NotificationChain`. It simply notifies any observers of the “touch,” as they would expect.

Multiplicity-Many References

You may have noticed in the previous section's example that the `orders` reference is multiplicity-many, with a lower bound of 0 and no upper bound. In other words, one customer may have many purchase orders. In EMF, a multiplicity-many reference—that is, any reference for which the upper bound is greater than 1—is manipulated using a list API, so only a `get()` method is generated in the interface:

```
EList getOrders();
```

Like the multi-valued attributes that we described in Section 9.2.4, multi-valued references are represented by `EList`, the EMF extension of `java.util.List`. From a client perspective, you can consider it a standard Java list. For example, the following code adds a purchase order to a customer's `orders` reference:

```
aCustomer.getOrders().add(aPurchaseOrder);
```

To iterate over a customer's orders, you would do something like this:

```
for (Iterator iter =
      aCustomer.getOrders().iterator(); iter.hasNext(); ) {
    PurchaseOrder po = (PurchaseOrder)iter.next();
    ...
}
```

As you can see, from a client perspective, the API for manipulating multiplicity-many references is nothing special. However, because the `orders` reference is part of a bidirectional association, we still need to do all the fancy inverse handshaking that we showed for the `setCustomer()` method in the previous section. Looking at the implementation of the `getOrders()` method in `CustomerImpl` shows us how the multiplicity-many case gets handled:

```
protected EList orders = null;

public EList getOrders() {
    if (orders == null) {
        orders =
            new EObjectWithInverseResolvingEList(PurchaseOrder.class,
                                                this, EPO1Package.CUSTOMER__ORDERS,
                                                EPO1Package.PURCHASE_ORDER__CUSTOMER);
    }
    return orders;
}
```

The `getOrders()` method returns a specialized implementation class, `EObjectWithInverseResolvingEList`, which is constructed with all the information it needs to do the inverse handshaking during add and remove calls. EMF actually provides a number of different `EList` implementation classes,⁸ which are used to efficiently implement all the different types of multiplicity-

many structural features. In Section 9.2.4, we saw that `EDataTypeEList` and `EDataTypeUniqueEList` are used for multi-valued attributes. For one-way references, we use `EObjectResolvingEList`. If the reference doesn't need proxy resolution we use `EObjectEList` or `EObjectWithInverseEList`, and so on.

Returning to our example, the list used to implement the `orders` reference is created with two feature ID arguments. `EPO1Package.CUSTOMER__ORDERS` identifies the reference itself and is used in change notifications. The opposite reference is identified by `EPO1Package.PURCHASE_ORDER__CUSTOMER`. It is used in the implementation of `add()`, which calls to `eInverseAdd()` on the `PurchaseOrder`, similar to the way `eInverseAdd()` was called on the `Customer` by `setCustomer()`. We will discuss feature IDs and the reverse handshaking methods further in Section 9.6.

Containment References

The `items` reference from `PurchaseOrder` to `Item` is a containment reference, which makes purchase orders a container for items (Figure 9.14).



Figure 9.14. The `items` containment reference.

A containment reference is indicated by a black diamond on the container end of the association, in this case `PurchaseOrder`. In full, the illustrated association specifies that a `PurchaseOrder` aggregates, by value, 0 or more `Items`. A by-value aggregation association, or containment reference, is modeled in Ecore by an `EReference`, from the container class to the contained class, with its `containment` attribute set to `true`. Such references are particularly important because they identify the parent or owner of a target instance, which implies the location of the object when persisted. That is to say that objects are always persisted in the same document as their container.⁹

Containment affects the generated code in several ways. First of all, because a contained object is guaranteed to be in the same resource as its container, proxy resolution is not needed. Therefore, the generated `get()` method for a multiplicity-many containment reference uses a non-resolving `EList` implementation class:¹⁰

```

protected EList items = null;

public EList getItems() {
    if (items == null) {
        items = new EObjectContainmentEList(Item.class, this, ... );
    }
    return items;
}
  
```

⁸Actually, all the concrete `EList` implementation classes are simple subclasses of one very functional and efficient base implementation class, `EcoreEList`.

⁹A specialized resource implementation could override this, but from the EMF perspective, the objects would still be considered to be in one resource.

¹⁰The naming convention used for `EList` implementation classes is that all resolving list names end with "ResolvingEList".

In addition to not needing to resolve proxies, an `EObjectContainmentEList` also implements the `contains()` method quite efficiently: in constant time, as compared with the linear-time implementation offered by non-containment `ELists`. This is particularly important because duplicate entries are not allowed in EMF reference lists, so `contains()` is called during `add()` operations as well.

Because an object can only have one container, adding an object to a containment reference also means removing the object from any container that it is currently in, even if that containment is through a different reference. For example, adding an item to a purchase order's `items` reference may involve removing it from the `items` of some other purchase order. That's no different from any other two-way association with a multiplicity-1 inverse. Imagine, however, that we have another containment reference from `Customer` to `Item` called `consideredItems`. If a given item is in the `consideredItems` reference of some customer when we add it to a purchase order's `items`, then we'd need to remove it from the customer's `consideredItems` association first.

To implement this behavior efficiently, the `EObjectImpl` base class has a field called `eContainer`, of type `EObject`, that it uses to store its container generically. As a result, containment references are always implicitly bidirectional. The following code accesses an `Item`'s containing `PurchaseOrder`:

```
EObject container = item.eContainer();
if (container instanceof PurchaseOrder)
    purchaseOrder = (PurchaseOrder)container;
```

If you want to avoid the downcast, you can change the association to be explicitly bidirectional instead (Figure 9.15):

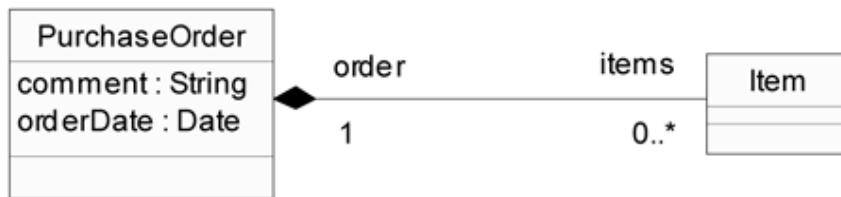


Figure 9.15. An explicitly bidirectional containment association.

Then, EMF will generate a type-safe `get()` method for you in `ItemImpl`:

```
public PurchaseOrder getOrder() {
    if (eContainerFeatureID != POPackage.ITEM__ORDER) return null;
    return (PurchaseOrder)eContainer;
}
```

This explicit `get()` method uses the `eContainer` field from `EObjectImpl` instead of a generated instance variable, which we have seen is used to implement non-containment references such as `getPreviousOrder()`. Also, notice that it verifies that the current containment is, in fact, provided through the `order` feature by testing the `eContainerFeatureID` field, which is also defined and maintained by `EObjectImpl`.

Non-Proxy-Resolving References

In Section 9.3.1, we saw that the implementation of the `get()` method for a reference is slightly more complicated than the implementation of the same method for an attribute. This is true even for a single-valued, non-containment, one-way reference, such as the `previousOrder` reference of the `PurchaseOrder` class. The reason for this added complexity is that we need to allow for the possibility that the target of a reference is a proxy for an object in a resource that has not been loaded. Extra code is needed to check whether the target object is a proxy, and if so, to resolve it.

If you are sure that the source and target objects of a reference will always reside in the same resource, it is possible to optimize the generated `get()` method by omitting the proxy resolution code. This optimization is made when the `resolveProxies` attribute of an `EReference` is `false`.

For example, in the ExtendedPO2 model, the `previousOrder` reference is non-proxy-resolving, resulting in the following simple implementation of `getPreviousOrder()`:

```
public PurchaseOrder getPreviousOrder() {
    return previousOrder;
}
```

A multi-valued, non-proxy resolving reference uses an `EObjectEList` or `EObjectWithInverseEList` as its `EList` implementation, rather than a `EObjectResolvingEList` or `EObjectWithInverseResolvingEList`. This leads to a performance improvement because the implementations of the non-resolving lists, just like the non-resolving `set()` methods, have less overhead.

As mentioned in Section 9.3.4, containment references, which are not permitted to cross resource boundaries, never perform proxy resolution. The decision to suppress proxy resolution on any other references should be made carefully, based on knowledge of how instance models will be placed in resources for serialization.

Volatile References

As we saw in Section 9.2.6, volatile attributes have no storage or implementation code generated. You are required to implement a generated stub implementation by hand. The same is true for volatile references. The `pendingOrders` and `shippedOrders` references defined in the ExtendedPO2 model are good examples (Figure 9.16):

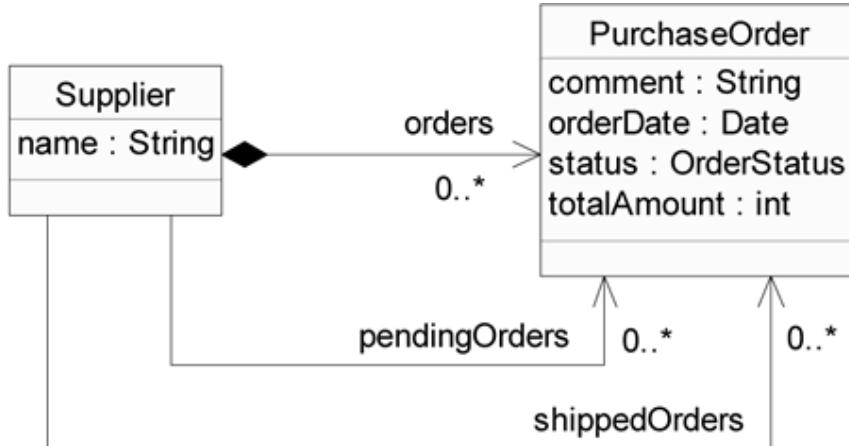


Figure 9.16. The `pendingOrders` and `shippedOrders` volatile references.

These two references are meant to provide filtered views of the purchase orders according to their `status`, the enumeration attribute we introduced in Section 9.2.3. The `shippedOrders` reference should include the subset of `orders` whose status is `Complete`, and the `pendingOrders` reference should include those whose status is `Pending`. In fact, we do not need to store these references because the lists can be computed from `orders` when they are accessed.

The generated implementation for a volatile reference is exactly the same as that for a volatile attribute. It simply throws an `UnsupportedOperationException`:

```

public EList getPendingOrders() {
    // TODO: implement this method to return the
    //       'Pending Orders' reference list
    throw new UnsupportedOperationException();
}

```

We'll follow through with this example in Chapter 12, showing how you can implement this method.

Non-Changeable References

We can suppress the generation of the `set()` method for a reference by setting the `changeable` attribute of the `EReference` to be `false`. Just as with attributes, it often makes sense to make volatile references non-modifiable as well. Note that `set()` methods are never generated for multi-valued references, so making such a reference non-changeable has no effect on code generation.

Unsettable References

Unsettable references, like their attribute counterparts, support an additional unset state that is distinguishable from that in which any value is set, including the default `null` value. Unsettable multiplicity-1 references are also implemented by `unset()` and `isSet()` methods that are generated in addition to the usual `get()` and `set()` accessors.

For one-way references, the generated `isSet()` and `unset()` implementations use the same patterns that we saw in the discussion of unsettable attributes. For bidirectional references, including implicitly bidirectional containment references, the `unset()` method also needs to do the same kind of inverse handshaking as the `set()` method.

For example, consider the `shipTo` containment reference¹¹ defined by `PurchaseOrder` (Figure 9.17):

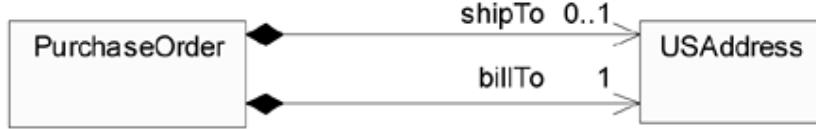


Figure 9.17. `PurchaseOrder` and its two `USAddress` references.

Suppose that we would like to distinguish between the case where `shipTo` is unset and the case where it is set to `null`. We could interpret the former case to mean that the order should be shipped to the billing address and the latter to mean that it should be held for pickup.

If `shipTo` is made unsettable, the following implementations are generated:

```

protected boolean shipToESet = false;

public void unsetShipTo() {
    if (shipTo != null) {
        NotificationChain msgs = null;
        msgs =
            ((InternalEObject)shipTo).eInverseRemove(this, ... , msgs);
        msgs = basicUnsetShipTo(msgs);
        if (msgs != null) msgs.dispatch();
    }
}

else {
    boolean oldShipToESet = shipToESet;
    shipToESet = true;
    if (eNotificationRequired())
        eNotify(new ENotificationImpl(this, ... , oldShipToESet));
}

```

¹¹Recall that containment associations are implicitly bidirectional and therefore require inverse handshaking.

```

}

public boolean isSetShipTo() {
    return shipToESet;
}
}

```

As you can see, the `unsetShipTo()` method uses a pattern similar to bidirectional `set()` method pattern described in Section 9.3.2. It is simplified slightly by the fact that `unset()`, by definition, always sets the value to `null`. Notice how `unset()` even delegates to a `basicUnsetShipTo()` method to do the actual unset, just like a bidirectional `set()` method delegates to a `basicSet()` method.

We must also point out that the `set()` and `basicSet()` methods are also slightly different for unsettable references. In addition to actually setting the value of the reference, they must also set to `true` the field that tracks whether the reference has been set.

Finally, another word on multiplicity-many references: like their attribute counterparts, unsettable multiplicity-many references are implemented using subclasses of their usual `EList` implementations that provide `unset()` and `isSet()` methods.

Map References

EMF provides special support for an indexed collection of type-safe key-value pairs via an interface called `EMap`. Note that `EMap` is not itself derived from `java.util.Map`, although it can provide a map view, which is. Instead, it is a special kind of `EList`—one whose items implement `Map.Entry`.

The class that actually implements `Map.Entry` for a particular `EMap` is an almost typical implementation class that is generated for an `EClass`. Such an `EClass` is distinguished by the fact that its `instanceClassName` is set to “`java.util.Map$Entry`” and by the presence of two structural features: `key` and `value`.

Assume that we want to add a collection of purchase orders to our model. We expect that the collection is going to become large, and we would like to be able to access orders quickly. An approach is captured by the following model, one of the three map examples discussed in Chapter 8 (Figure 9.18):

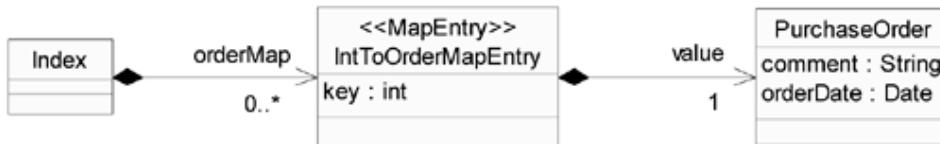


Figure 9.18. A map-type reference.

This models a map from numeric keys to `PurchaseOrder` values. The keys are new—they do not appear elsewhere in the model—and we will assume that they are non-contiguous.¹²

The following implementation class is generated for `IntToOrderMapEntry`:

```

public class IntToOrderMapEntryImpl extends EObjectImpl
    implements BasicEMap.Entry
{
    protected static final int KEY_EDEFAULT = 0;
    protected int key = KEY_EDEFAULT;
    protected PurchaseOrder value = null;

    ...
}

```

¹²If they were contiguous, we could just use an ordinary multiplicity-many feature and access them by index.

The class extends `EObjectImpl` as usual, but notice that it implements `BasicEMap.Entry`, a simple extension of `Map.Entry` that is used internally for this purpose. No interface is generated.

The implementation mostly follows the usual patterns, with a few important exceptions. Most notably, the usual accessors for `key` and `value` are named `getTypedKey()`, `setTypedKey()`, `getTypedValue()`, and `setTypedValue()`, rather than `getKey()`, `setKey()`, `getValue()`, and `setValue()`. Different methods are generated with the usual names that do any casting, wrapping, and unwrapping necessary to connect the generic, `Object-based` `Map.Entry` interface to the type-safe implementation:

```
public Object getKey() {
    return new Integer(getTypedKey());
}

public void setKey(Object key) {
    setTypedKey(((Integer)key).intValue());
}

public Object getValue() {
    return getTypedValue();
}

public Object setValue(Object value) {
    Object oldValue = getValue();
    setTypedValue((PurchaseOrder)value);
    return oldValue;
}
```

Also, the following simple methods are generated to complete the implementation of `BasicEMap.Entry`:

```
protected int hash = -1;

public int getHash() {
    if (hash == -1) {
        Object theKey = getKey();
        hash = (theKey == null ? 0 : theKey.hashCode());
    }
    return hash;
}

public void setHash(int hash) {
    this.hash = hash;
}
```

Finally, `orderEMap` shows up in `Index` as an `EMap`-typed multiplicity-many containment reference. The following declaration is generated in the `Index` interface:

```
EMap getOrderMap();
```

In the `IndexImpl` implementation class, we find the following:

```
protected EMap orderMap = null;

public EMap getOrderMap() {
    if (orderMap == null) {
        orderMap =
            new EcoreEMap(POMapPackage.eINSTANCE.getIntToOrderMapEntry(),
                          IntToOrderMapEntryImpl.class, this,
                          POMapPackage.INDEX__ORDER_MAP);
    }
    return orderMap;
}
```

It is implemented using `EcoreEMap`, which maintains a containment-like relationship with the `Map.Entry` class and keeps an index, for fast access, based on the `key` structural feature.

By fitting the `Map.Entry` interface on top of a nearly ordinary class implementation and maintaining it through a containment-like relationship, we are able to leverage all the usual EMF mechanisms so that everything “just works,” including reflection, notification, and persistence. Also, since `key` and `value` are just structural features, their types can be data types or classes, and they can be multiplicity-1 or multiplicity-many. All of the usual patterns, though using some slightly unusual names, do all the usual things under the covers.

Operations

Any `EOperations` contained by an `EClass` correspond to methods in the generated interface and implementation class. Because EMF provides no way of specifying operation behaviors, it simply generates empty method bodies, leaving you to complete the implementation by hand.

As an example, suppose we wish to add an operation to `PurchaseOrder` that removes an item from the order. Without such an operation, an item could still be removed by retrieving the whole list representing the `items` reference, finding the item, and removing it from the list. However, we could imagine that a `removeItem` operation might conveniently do some additional processing, like checking to see if the removal makes the order complete and, if so, updating its status.

We update the model to include the `removeItem` operation (Figure 9.19):

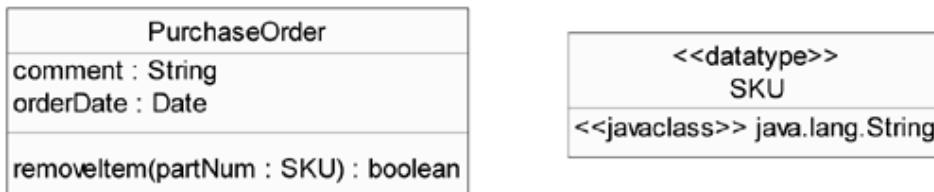


Figure 9.19. Class `PurchaseOrder` with an operation.

This operation has a single `SKU`-typed parameter, identifying the item to be removed by its part number. Its return value indicates whether the removal was successful.

With this addition, the following signature is included in the `PurchaseOrder` interface:

```
boolean removeItem(String partNum);
```

Recall that the `SKU` data type maps to `String`, the type used for the generated method's parameter.

The corresponding implementation class includes a stub implementation for the method. To remind you that you need to provide the actual implementation, the generated stub throws an `UnsupportedOperationException`.

```
/**
 * @generated
 */
public boolean removeItem(String partNum) {
    // TODO: implement this method
    throw new UnsupportedOperationException();
}
```

As always, when you modify this method to provide the appropriate behavior, you should also remove the `@generated` Javadoc comment that precedes it.

Because their behavior cannot be specified, modeled operations serve only as a formal definition of a class' behavioral interface and, of course, as the basis for generation of method stubs. As such, they provide no overwhelming advantage over simply adding the methods by hand to the interface and class. Nonetheless, some people see considerable value in being able to regard a model definition as a complete API specification, which is why, for example, operations are included in the Ecore model.

Class Inheritance

As we saw in Chapter 5, Ecore supports inheritance by allowing classes to specify any number of other classes as its supertypes. In this section, we explore how inheritance in a model is reflected in the generated code.

Single Inheritance

ExtendedPO2 expands the modeling of an address to accommodate those located outside of the United States. It introduces an abstract class, **Address**, as a base for different types of addresses, including **USAddress** (Figure 9.20):

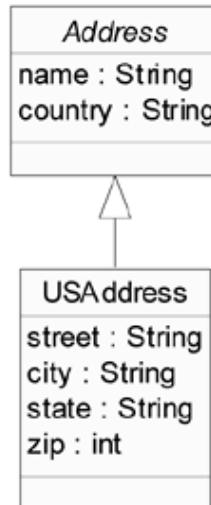


Figure 9.20. Single inheritance.

The EMF generator handles single inheritance as you would expect: the generated interface and implementation class for an **EClass** extend the interface and implementation for its **eSuperType**. Thus, the interface declaration for **USAddress** is as follows:

```
public interface USAddress extends Address
```

The declaration of the implementation class looks like this:

```
public class USAddressImpl extends AddressImpl implements USAddress
```

Note that the interface and class still inherit indirectly from **EObject** and **EObjectImpl**, which remain the bases of **Address** and **AddressImpl**, respectively.

Multiple Inheritance

Like Java itself, EMF supports multiple interface inheritance, but each implementation class can only extend one base. Therefore, when we model multiple inheritance, we need to identify which of the supertypes should be used for the implementation base class. The others will then be treated as mixin interfaces, with their features re-implemented in the derived implementation class.

Consider the following example from ExtendedPO2 (Figure 9.21):

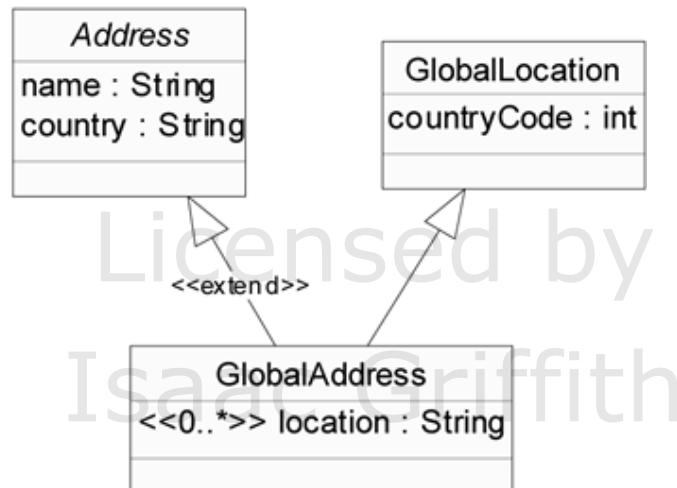


Figure 9.21. Multiple class inheritance.

Here, **GlobalAddress** derives from two classes: **Address** and **GlobalLocation**. We have identified **Address** as the implementation base with an `<<extend>>` stereotype.¹³ When generated, the **GlobalAddress** interface extends the two supertype interfaces:

```
public interface GlobalAddress extends Address, GlobalLocation
```

As usual, the **GlobalAddressImpl** class implements this interface and extends the implementation base class. It also includes implementations of the mixed-in `getCountryCode()` and `setCountryCode()` methods:

```
public class GlobalAddressImpl extends AddressImpl
    implements GlobalAddress
{
    public int getCountryCode() {
        ...
    }

    public void setCountryCode(int newValue) {
        ...
    }
}
```

The multiple inheritance also results in the generation of override implementations for `eDerived-StructuralFeatureID()` and `eBaseStructuralFeatureID()`, as we will describe in Sections 9.6.9 and 9.6.10, respectively.

¹³Actually, it is the first **EClass** specified by **eSuperType** in the core model that is used as the implementation base class. The UML stereotype is needed to indicate that **Address** should be first in the Ecore representation.

Interface Inheritance and Implementation

So far, we have not discussed inheritance involving interface-only classes, those modeled by an **EClass** with its **interface** attribute set to `true`. In fact, such classes are primarily meant to be used in inheritance: they may be specified as supertypes for other interface-only classes, modeling interface inheritance, or for ordinary classes, modeling interface implementation.

Let's illustrate this latter case by modifying the example from the previous section (Figure 9.22):

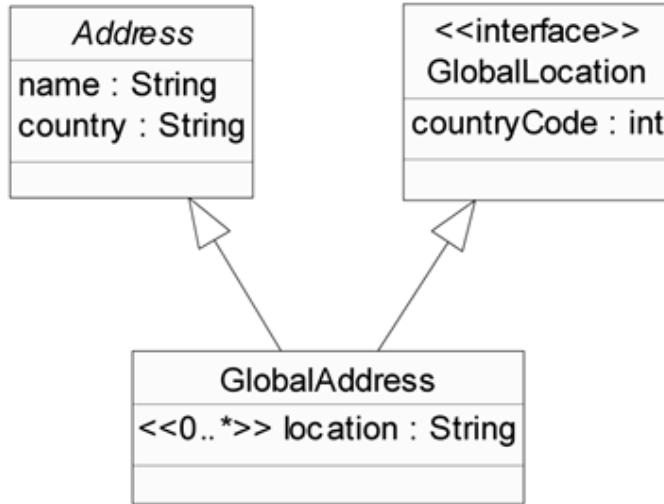


Figure 9.22. Interface inheritance.

If **GlobalLocation** is never going to be instantiated itself, we can make it an interface, suppressing the generation of an implementation class. Since **Address** is now the only supertype with an implementation and, hence, the only option for the implementation base, we can safely remove the `<<extend>>` stereotype.

Implementing an interface like this involves generating implementations for all of its features. So, the generated **GlobalAddressImpl** class is identical to the one we looked at in the previous section. The only difference here is that no **GlobalLocationImpl** class is generated.

Recall that we can also model non-EMF interfaces, specifying only the name of the Java interface and not any attributes, references, or operations. In the case where such an interface is implemented by a class, the only thing affected is the `extends` clause of the interface generated for the implementing class. Since the model specifies nothing about the external interface, we will need to code, by hand, implementations of any methods that it declares.

Reflective Methods

In addition to the accessor methods that are generated specifically to handle each attribute and reference in a class, the generated Java implementation class includes efficient implementations of the reflective accessor methods that are declared by the **EObject** interface. Although you are unlikely to ever modify these implementations, you may be interested in knowing how they work if you write code that uses reflection to access structural features.

Such reflective clients may make use of the `eGet()`, `eSet()`, `eIsSet()`, and `eUnset()` methods. In addition, there are several methods that do not appear in the **EObject** interface and are generally only used by generated and framework code. These methods include `eInverseAdd()`, `eInverseRemove()`, `eBasicRemoveFromContainer()`, `eDerivedStructuralFeatureID()`, and `eBaseStructuralFeatureID()`. This section briefly describes all of these methods.

Feature IDs

Before we talk about the various reflective methods, we should understand a little more about feature IDs. We have already come across them several times in this chapter, but we have yet to provide a complete discussion.

Basically, a feature ID is an integer value that uniquely identifies a structural feature within a class. In particular, it is the position of the feature in the list returned by the `getEAllStructuralFeatures()` method of an **EClass**. For easy access, this ID is also generated as a static `int` constant in the interface for the package that contains the class. As we will see in a moment, it is used by the various reflective methods in efficient `switch` statements that select a course of action based on the specified feature.

The assignment of feature IDs occurs when model code is generated. It is quite straightforward for classes that only use single inheritance, but it gets a little more complicated when multiple inheritance is involved. For classes with no supertypes, features are assigned IDs sequentially, beginning with 0. For classes with a single supertype, inherited features have the same ID as in the supertype, and locally defined features are assigned IDs sequentially starting with the next available value.

However, this simple scheme breaks down when a class has more than one supertype, as features inherited from different ones would have conflicting IDs. We handle this problem by offsetting the feature IDs coming from every supertype, except for one. The special supertype whose feature IDs are maintained is the same one that we use as the implementation base, as described in Section 9.5.2. IDs for features inherited from each of the other supertypes are then offset as necessary to ensure uniqueness. Finally, IDs are assigned to locally defined features sequentially, starting with the next available value. As you will soon see, the `eDerivedStructuralFeatureID()` and `eBaseStructuralFeatureID()` methods provide a mapping between a base ID, with no offset, and the corresponding offset value that is used in a particular subclass.

Feature IDs that are negative have a special purpose: they represent container references that have no feature of their own. Recall from Section 9.3.4 that `EObjectImpl` defines two fields, `eContainer` and `eContainerFeatureID`, to generically keep track of an object's container and the feature through which that object is accessed. However, for containment associations that are not bidirectional, there is no container feature to store. In this case, we use the containment feature ID instead, but subtract it from -1 so that we can distinguish it from a real container feature ID, which would be a positive value.¹⁴ We'll see this in use when we look at the `eInverse()` methods, below.

The `eGet()` Method

The `eGet()` method takes an instance of `EStructuralFeature` to identify the desired attribute or reference, and a flag indicating whether or not proxy resolution is needed; it returns the value associated with the specified feature. For example, the following code is generated to implement `eGet()` in `PurchaseOrderImpl`:

```
public Object eGet(EStructuralFeature eFeature, boolean resolve) {
    switch (eDerivedStructuralFeatureID(eFeature.getFeatureID(),
        eFeature.getContainerClass())) {
        case PPOPackage.PURCHASE_ORDER__COMMENT:
            return getComment();
        case PPOPackage.PURCHASE_ORDER__ORDER_DATE:
            return getOrderDate();
        case PPOPackage.PURCHASE_ORDER__ITEMS:
            return getItems();
        case PPOPackage.PURCHASE_ORDER__BILL_TO:
            return getBillTo();
```

¹⁴You might think it would be a bit clearer if we just negated the value, but since feature numbering starts at 0, -0 would be indistinguishable from +0.

```

    case PPOPackage.PURCHASE_ORDER__SHIP_TO:
        return getShipTo();
    }
    return eDynamicGet(eFeature, resolve);
}

```

The implementation includes a switch statement that selects among the features known to be supported by the class, including inherited features, and delegates to the appropriate specific `get()` method. For primitive-typed attributes, the value is wrapped in the appropriate object wrapper before being returned. If no match is found, `eDynamicGet()` is called to handle any dynamic features that may have been created.

The `resolved` flag needs to be checked for proxy-resolving, non-containment references. As an example, let's look again at the `previousOrder` reference introduced in Section 9.3.1. The case for it looks like this:

```

case EPO1Package.PURCHASE_ORDER__PREVIOUS_ORDER:
    if (resolve) return getPreviousOrder();
    return basicGetPreviousOrder();
}

```

Notice the call to the `basicGetPreviousOrder()` method. This method is generated just for this purpose. It simply returns `previousOrder` without doing the proxy resolution.

The `eSet()` Method

The `eSet()` method takes an instance of `EStructuralFeature` to identify the desired attribute or reference, and a value of compatible type; it updates the feature accordingly. The following implementation of `eSet()` is generated in `PurchaseOrderImpl`:

```

public void eSet(EstructuralFeature eFeature, Object newValue) {
    switch (eDerivedStructuralFeatureID(eFeature.getFeatureID(),
                                       eFeature.getContainerClass())) {
        case PPOPackage.PURCHASE_ORDER__COMMENT:
            setComment((String)newValue);
            return;
        case PPOPackage.PURCHASE_ORDER__ORDER_DATE:
            setOrderDate((Date)newValue);
            return;
        case PPOPackage.PURCHASE_ORDER__ITEMS:
            getItems().clear();
            getItems().addAll((Collection)newValue);
            return;
        case PPOPackage.PURCHASE_ORDER__BILL_TO:
            setBillTo((USAddress)newValue);
            return;
        case PPOPackage.PURCHASE_ORDER__SHIP_TO:
            setShipTo((USAddress)newValue);
            return;
    }
    eDynamicSet(eFeature, newValue);
}

```

The implementation switches on the feature ID and delegates to the appropriate specific `set()` method. If no match is found, it calls the `eDynamicSet()` method to handle any dynamic features that may be present.

The `eIsSet()` Method

The `eIsSet()` method takes an instance of `EStructuralFeature` to identify the desired attribute or reference, and tests whether that feature has been set. This implementation of `eIsSet()` is generated in `PurchaseOrderImpl`:

```

public boolean eIsSet(EStructuralFeature eFeature) {
    switch (eDerivedStructuralFeatureID(eFeature.getFeatureID(),
                                       eFeature.getContainerClass())) {
        case PPOPackage.PURCHASE_ORDER__COMMENT:
            return COMMENT_EDEFAULT == null ?
                   null : !COMMENT_EDEFAULT.equals(comment);
        case PPOPackage.PURCHASE_ORDER__ORDER_DATE:
            return ORDER_DATE_EDEFAULT == null ?
                   nullDate != null : !ORDER_DATE_EDEFAULT.equals(orderDate);
        case PPOPackage.PURCHASE_ORDER__ITEMS:
            return items != null && !getItems().isEmpty();
        case PPOPackage.PURCHASE_ORDER__BILL_TO:
            return billTo != null;
        case PPOPackage.PURCHASE_ORDER__SHIP_TO:
            return shipTo != null;
    }
    return eDynamicIsSet(eFeature);
}

```

It includes a `switch` statement that selects among features supported by the class. For any feature that is unsettable, it delegates to the specific `isSet()` method for that feature. For other features, the method compares the feature's current value to its default: `true` is returned if they match; otherwise, `false` is returned. If the feature is not found, `eDynamicIsSet()` is called to handle any dynamic features that may be present.

None of the features in this example are unsettable. Let us assume, as we did in Section 9.3.8, that `shipTo` is, so that we can illustrate the pattern for this case:

```

case PPOPackage.PURCHASE_ORDER__SHIP_TO:
    return isSetShipTo();

```

The `eUnset()` Method

The `eUnset()` method takes an instance of `EStructuralFeature` to identify the desired attribute or reference, and unsets or resets that feature. The following implementation is generated in `PurchaseOrderImpl`:

```

public void eUnset(EStructuralFeature eFeature) {
    switch (eDerivedStructuralFeatureID(eFeature.getFeatureID(),
                                       eFeature.getContainerClass())) {
        case PPOPackage.PURCHASE_ORDER__COMMENT:
            setComment(COMMENT_EDEFAULT);
            return;
        case PPOPackage.PURCHASE_ORDER__ORDER_DATE:
            setOrderDate(ORDER_DATE_EDEFAULT);
            return;
        case PPOPackage.PURCHASE_ORDER__ITEMS:
            getItems().clear();
            return;
        case PPOPackage.PURCHASE_ORDER__BILL_TO:
            setBillTo((USAddress)null);
            return;
        case PPOPackage.PURCHASE_ORDER__SHIP_TO:
            setShipTo((USAddress)null);
            return;
    }
    eDynamicUnset(eFeature);
}

```

Once again, the implementation switches on the feature ID. For any feature that is unsettable, it delegates to the specific `unset()` method for that feature. A non-unsettable feature is just set to its default value. If the feature is not found, `eDynamicUnset()` is called to handle any dynamic features that may be present.

As in the previous section, to illustrate the pattern for unsettable features, let's assume that `shipTo` is unsettable. Its case would then look like this:

```
case PPOPackage.PURCHASE_ORDER__SHIP_TO:
    unsetShipTo();
    return;
```

The `eInverseAdd()` Method

The `eInverseAdd()` method is called to add to or set the opposite end of a bidirectional reference and is generated only for classes that include such references. It is called from the generated `set()` methods, as we saw in Section 9.3.2, and from the `EList` implementation classes for bidirectional references. You will likely never need to call this method directly. Because PrimerPO includes no bidirectional references, we will look at the implementation generated for `PurchaseOrderImpl` from the ExtendedPO2 model:

```
public NotificationChain eInverseAdd(InternalEObject otherEnd,
                                     int featureID,
                                     Class baseClass,
                                     NotificationChain msgs) {
    if (featureID >= 0) {
        switch (eDerivedStructuralFeatureID(featureID, baseClass)) {
            case EPO2Package.PURCHASE_ORDER__ITEMS:
                return ((InternalEList).getItems()).basicAdd(otherEnd, msgs);
            case EPO2Package.PURCHASE_ORDER__CUSTOMER:
                if (customer != null)
                    msgs = ((InternalEObject)customer).eInverseRemove(this,
                                                               EPO2Package.CUSTOMER__ORDERS, Customer.class, msgs);
                return basicSetCustomer((Customer)otherEnd, msgs);
            default:
                return eDynamicInverseAdd(otherEnd, featureID, baseClass, msgs);
        }
    }
    if (eContainer != null)
        msgs = eBasicRemoveFromContainer(msgs);
    return eBasicSetContainer(otherEnd, featureID, msgs);
}
```

When the `featureID` is positive, representing an explicitly navigable reference, the method switches on this value to select the proper implementation for “adding” the inverse reference. A case is included for every bidirectional reference of the class. If the feature is multiplicity-many, it *adds* to the reference. Otherwise, it removes the previous value, if necessary, and then *sets* the reference.

When the `featureID` is negative, as discussed in Section 9.6.1, it represents a containment feature ID from the opposite end. In this case, the method removes the object from a previous container, and then sets its `eContainer` and `eContainerFeatureID` fields by calling `eBasicSetContainer()`.

The `eInverseRemove()` Method

The `eInverseRemove()` method is called to remove or unset the opposite end of a bidirectional or containment reference and is generated only for classes that include such references. It can be called from generated `set()` or `eInverseAdd()` methods, or from the `EList` implementation classes for bidirectional or containment references. Again, we look at the implementation generated for `PurchaseOrderImpl` from the ExtendedPO2 model:

```

public NotificationChain eInverseRemove(InternalEObject otherEnd,
                                       int featureID,
                                       Class baseClass,
                                       NotificationChain msgs) {
    if (featureID >= 0) {
        switch (eDerivedStructuralFeatureID(featureID, baseClass)) {
            case EPO2Package.PURCHASE_ORDER__ITEMS:
                return ((InternalEList)getItem()).basicRemove(otherEnd, msgs);
            case EPO2Package.PURCHASE_ORDER__BILL_TO:
                return basicSetBillTo(null, msgs);
            case EPO2Package.PURCHASE_ORDER__SHIP_TO:
                return basicSetShipTo(null, msgs);
            case EPO2Package.PURCHASE_ORDER__CUSTOMER:
                return basicSetCustomer(null, msgs);
            default:
                return eDynamicInverseRemove(otherEnd,
                                              featureID, baseClass, msgs);
        }
    }
    return eBasicSetContainer(null, featureID, msgs);
}

```

Like `eInverseAdd()`, this method switches on a positive `featureID` to select the appropriate implementation. For a negative `featureID`, it resets the `eContainer`.

The `eBasicRemoveFromContainer()` Method

The `eBasicRemoveFromContainer()` method is called to remove an object from its container and is generated only for classes that include bidirectional container references. The implementation generated for `ItemImpl` from the ExtendedPO2 model looks like this:

```

public NotificationChain eBasicRemoveFromContainer(
    NotificationChain msgs) {
    if (eContainerFeatureID >= 0) {
        switch (eContainerFeatureID) {
            case EPO2Package.ITEM__ORDER:
                return ((InternalEObject)eContainer).eInverseRemove(this,
                    EPO2Package.PURCHASE_ORDER__ITEMS,
                    PurchaseOrder.class, msgs);
            default:
                return eDynamicBasicRemoveFromContainer(msgs);
        }
    }
    return ((InternalEObject)eContainer).eInverseRemove(this,
        EOPPOSITE_FEATURE_BASE - eContainerFeatureID, null, msgs);
}

```

This method switches based on the `eContainerFeatureID` field to select the appropriate implementation for the current container feature.

The `eDerivedStructuralFeatureID()` Method

As discussed in Section 9.6.1, a feature ID uniquely identifies each structural feature within a given class. When a class uses multiple inheritance, the IDs for some of its inherited features are offset from their base values. The mappings from base feature IDs to those appropriate to a given class are defined by `eDerivedStructuralFeatureID()`.

`EObjectImpl` provides an implementation of this method that simply returns the feature ID that it is passed, without offsetting it. Only those classes with more than one supertype require an override to be generated.

We saw an example of a class from ExtendedPO2 that uses multiple inheritance, **GlobalAddress**, in Section 9.5.2. Here is the override that is generated in **GlobalAddressImpl** as a result:

```
public int eDerivedStructuralFeatureID(int baseFeatureID,
                                      Class baseClass) {
    if (baseClass == GlobalLocation.class) {
        switch (baseFeatureID) {
            case EPO2Package.GLOBAL_LOCATION_COUNTRY_CODE:
                return EPO2Package.GLOBAL_ADDRESS_COUNTRY_CODE;
            default: return -1;
        }
    }
    return
        super.eDerivedStructuralFeatureID(baseFeatureID, baseClass);
}
```

As you can see, it only needs to adjust the ID for features inherited from mixin classes. In this case, there is only one such feature: **countryCode**, which is defined in **GlobalLocation**. For anything else, the unadjusted value returned by the **EObjectImpl** implementation is appropriate.

Notice that this method is used heavily by the other reflective methods: it computes the value on which they switch in order to distinguish between the features of the class.

The **eBaseStructuralFeatureID()** Method

The **eBaseStructuralFeatureID()** method defines the mappings from feature IDs appropriate to a given class to base feature IDs; that is, it does the reverse of **eDerivedStructuralFeatureID()**.

This method is most commonly utilized by adapters in determining the affected feature from a notification. When an object sends a notification, it includes the ID of the feature relative to the class that implements it, not the one that declares it. Of course, the recipient of the notification should not need to know what class is implementing the feature. So, an Adapter, in its **notifyChanged()** method, wants to obtain the base feature ID. It calls **getFeatureID()** on the Notifier, specifying the class that it is adapting. This method uses **eBaseStructuralFeatureID()** to adjust the ID before returning it.

Again, an **eBaseStructuralFeatureID()** override is only generated for classes with more than one supertype. The implementation generated in **GlobalAddressImpl** is as follows:

```
public int eBaseStructuralFeatureID(int derivedFeatureID,
                                   Class baseClass) {
    if (baseClass == GlobalLocation.class) {
        switch (derivedFeatureID) {
            case EPO3Package.GLOBAL_ADDRESS_COUNTRY_CODE:
                return EPO3Package.GLOBAL_LOCATION_COUNTRY_CODE;
            default: return -1;
        }
    }
    return
        super.eBaseStructuralFeatureID(derivedFeatureID, baseClass);
}
```

Notice that this method is identical to **eDerivedStructuralFeatureID()**, only with the values reversed. This should not be too surprising, since it performs the same mapping, but in the opposite direction.

Factories and Packages

So far, we have looked at all of the interfaces and classes that correspond to the **EClassifiers** contained in an **EPackage**. In addition to these, EMF always generates two interfaces, with implementation classes, corresponding to the **EPackage** itself: a factory and a package.

The factory, as its name implies, is used for creating instances of model classes, while the package provides some static constants and convenience methods for accessing the metadata for the model.

While you aren't strictly required to use the factory or package in your application, EMF does encourage the use of the factory for object instantiation by making the constructors for model classes protected. Thus, client code that resides in another package cannot just use `new` to create instances directly. You can, however, manually change the visibility of the constructors to `public`, if that's what you really want.

The factory interface for the primer purchase order model looks like this:

```
public interface PPOFactory extends EFactory
{
    PPOFactory eINSTANCE = new PPOFactoryImpl();

    PurchaseOrder createPurchaseOrder();
    Item createItem();
    USAddress createUSAddress();

    PPOPackage getPPOPackage();
}
```

The base interface for generated factories, `EFactory`, declares a generic `create()` method for creating instances of model classes. It also declares the `createFromString()` and `convertToString()` methods, introduced in Section 9.2.2, that convert a data type instance into a string and back.

The generated factory interface adds to this interface a specific `create()` method for each instantiable class in the package, that is, for each **EClass** for which both `abstract` and `interface` are `false`. The factory implementation class implements each of these methods simply, by using `new` to create an instance of the appropriate class. For example:

```
public PurchaseOrder createPurchaseOrder() {
    PurchaseOrderImpl purchaseOrder = new PurchaseOrderImpl();
    return purchaseOrder;
}
```

The implementation also includes a specialized version of the `createFromString()` and `convertToString()` methods for each **EDataType** in the package. The generic versions of `create()`, `createFromString()`, and `convertToString()` are all implemented by calling out to the appropriate specialized version, as determined by the specified **EClass** or **EDataType**.

The generated factory interface also adds an accessor for its corresponding package, `getPPOPackage()`, and a static constant reference, `eINSTANCE`, to an implementation of the factory.

You may be thinking that it seems strange to see a dependency on the implementation class, right in the interface—it would seem to defeat the purpose of interface-implementation separation. The answer to this concern is that the statically referenced `PPOFactoryImpl` is intended to be the *default* factory implementation, not necessarily the *only* one.¹⁵ The `eINSTANCE` reference is a convenient way to access this default factory, if that's what you're using. If you want to use a different, specialized factory implementation, then you lose this convenience and will need to locate your factory in some other way.¹⁶

The generated package interface defines some properties of the package, includes a static constant reference to an implementation of the package, and provides convenient access to all the metadata in the model. For the primer purchase order model, it looks like this:

```
public interface PPOPackage extends EPackage
{
    String eNAME = "ppo";
    String eNS_URI = "http://com/example/ppo.ecore";
    String eNS_PREFIX = "com.example.ppo";

    PPOPackage eINSTANCE = PPOPackageImpl.init();

    static final int PURCHASE_ORDER = 0;
    static final int PURCHASE_ORDER__COMMENT = 0;
    static final int PURCHASE_ORDER__ORDER_DATE = 1;
    static final int PURCHASE_ORDER__ITEMS = 2;
    static final int PURCHASE_ORDER__BILL_TO = 3;
    static final int PURCHASE_ORDER__SHIP_TO = 4;
    static final int PURCHASE_ORDER_FEATURE_COUNT = 5;
    static final int ITEM = 1;
    static final int ITEM__PRODUCT_NAME = 0;
    static final int ITEM__QUANTITY = 1;
    ...

    EClass getPurchaseOrder();
    EAttribute getPurchaseOrder__Comment();
    EAttribute getPurchaseOrder__OrderDate();
    EAttribute getPurchaseOrder__Items();
    EReference getPurchaseOrder__PreviousOrder();
    ...
}
```

As you can see, the generated interface extends the `EPackage` base interface, and includes convenient static constants that define values to be used for the `name`, `nsURI` and `nsPrefix` attributes of the `EPackage`.

Just like the factory, you can access an instance of the generated package implementation via the `eINSTANCE` field.

The metadata is available in two forms: as `int` constants and as instances of Ecore classes, themselves. The `int` constants are the classifier and feature IDs that have come up at various points in this chapter. They provide the most efficient way to pass around meta information and, as we discussed in Section 9.6.10, to determine which feature has changed when handling notifications. We will discuss EMF adapters in greater detail in Chapter 13.

A generated set of accessors provides direct access to the metadata in the core model objects. As discussed in Chapter 5, these objects can then be examined to dynamically reveal all of the details of the model. They can also be used with the reflective API, to specify which element of the model should be operated on.

¹⁵You may have noticed above that we described the `eINSTANCE` field as a reference to “an implementation,” and not “the implementation.”

¹⁶This is not to imply that you will be on your own if you need a specialized factory implementation. EMF provides a convenient package registry facility that you can use to locate packages and factories. We discuss it in Section 13.1.2.

Notice that `eINSTANCE` is initialized with the value returned by `PPOPackageImpl.init()`. This static method not only creates and returns an instance of the implementation class, but also programmatically builds the core model for the package, which can then be accessed via the generated metadata accessors. This approach is preferred to that of reading the model from its serialized form because it gives better performance and ensures that the model's metadata stays in sync with the classes that realize it.

Switch Classes and Adapter Factories

The EMF generator can also optionally generate two utility classes for each **EPackage**: a switch class and a skeleton adapter factory class. The switch class implements a switch-like callback mechanism that is used for dispatching based on a model object's type. The adapter factory is a convenient base for building factories that demand-create type-specific adapters for modeled objects. The switch class is used in the implementation of the adapter factory.

For example, the generated code for the primer purchase order model can include `PPOSwitch` and `PPOAdapterFactory` classes. The switch class looks like this:

```
public class PPOSwitch
{
    protected static PPOPackage modelPackage;

    public Object doSwitch(EObject theEObject) {
        EClass theEClass = theEObject.eClass();
        if (theEClass.eContainer() == modelPackage) {
            switch (theEClass.getClassifierID()) {
                case PPOPackage.PURCHASE_ORDER: {
                    PurchaseOrder purchaseOrder = (PurchaseOrder)theEObject;
                    Object result = casePurchaseOrder(purchaseOrder);
                    if (result == null) result = defaultCase(theEObject);
                    return result;
                }
                case PPOPackage.ITEM: {
                    ...
                }
                case PPOPackage.US_ADDRESS: {
                    ...
                }
                default: return defaultCase(theEObject);
            }
        }
        return defaultCase(theEObject);
    }

    ...
}
```

The switch is executed by calling `doSwitch()`. If the specified `EObject` is an instance of one of the classes belonging to the package, this method switches on the classifier ID. This particular model includes no inheritance, but speaking generally, each `case` walks up the inheritance hierarchy from the actual type of the object to `EObject`, calling out to a specific `case()` handler method for each class. It stops when one of these methods returns a non-null value, which `doSwitch()` then returns.

As generated, each of the handler methods simply returns `null`. For example,

```
public Object casePurchaseOrder(PurchaseOrder object) {
    return null;
}
```

Thus, `doSwitch()` will always fall through all the cases and return `null`. So, to use the switch class, you will need to subclass it and provide overrides for whichever `case()` handlers you want to return something.

Before looking at how the adapter factory uses the switch class, let's briefly discuss what we expect an adapter factory to do. The class declaration gives us a hint:

```
public class PPOAdapterFactory extends AdapterFactoryImpl
{
    ...
}
```

We are subclassing `AdapterFactoryImpl`, which implements `AdapterFactory`. If we look at this interface, we find that the following two methods are the most important to clients:

- `isFactoryForType()` tells the client whether the adapter factory supports a given `type`. In this context, `type` is an arbitrary object for which any significance is imposed by the factory itself.
- `adapt()` takes an `object` for which an adapter is desired and the `type` of adapter required. If the object is a `Notifier` and if the factory supports the `type`, it returns an appropriate adapter; otherwise, it returns the `object` itself.

Now, looking at the base class, we find that `isFactoryForType()` always returns `false`, and that `adapt()` gets the adapters for the `object`, if it is a `Notifier`, and returns the first that is an adapter for the specified `type`. If there is no appropriate adapter, it calls `createAdapter()` to make one. This method just uses `new` to create an instance of `AdapterImpl`.

Thus, an `AdapterFactoryImpl`-based adapter factory really only needs to do two things:

- Override `isFactoryForType()` to determine whether the specified `type` is supported
- Override `createAdapter()` to create and return an appropriate adapter for a given `target`

Now, we can look at the whole generated adapter factory pattern to see how it does these things:

```
public class PPOAdapterFactory extends AdapterFactoryImpl
{
    protected static PPOPackage modelPackage;

    public boolean isFactoryForType(Object object) {
        if (object == modelPackage) {
            return true;
        }
        if (object instanceof EObject) {
            return
                ((EObject)object).eClass().getEPackage() == modelPackage;
        }
        return false;
    }
    ...
}
```

We see that this is an adapter factory for either the `EPackage` that represents this package or for any `EObject` that is an instance of a class belonging to it. The `createAdapter()` method is implemented using the switch class, like this:

```
public Adapter createAdapter(Notifier target)
{
    return (Adapter)modelSwitch.doSwitch((EObject)target);
}

protected PPOSwitch modelSwitch =
    new PPOSwitch()
{
    public Object casePurchaseOrder(PurchaseOrder object) {
```

```

        return createPurchaseOrderAdapter();
    }
    public Object caseItem(Item object) {
        return createItemAdapter();
    }
    public Object caseUSAddress(USAddress object) {
        return createUSAddressAdapter();
    }
    public Object defaultCase(EObject object) {
        return createEObjectAdapter();
    }
}
...

```

The switch class is used to select the type of the target; based on that, it calls to a `createAdapter()` handler method that creates the appropriate adapter. Since this is just a skeleton adapter factory, these handlers all just return `null`. For example:

```

public Adapter createPurchaseOrderAdapter() {
    return null;
}

```

So, to use the adapter factory, you will need to subclass it and provide overrides for `createAdapter()` handlers to create and return the desired type-specific adapters.

We will discuss the generated adapter factory and switch classes further in Chapter 13.

Customizing Generated Classes

At various points throughout this chapter, we have referred to modifications you must make, by hand, to generated Java code. In particular, the fact that EMF provides no support for modeling behavior necessitates hand-coding for volatile structural features, operations, and data type to string mappings.

If you are going to be modifying generated code by hand, you need to be able to regenerate it without worrying about losing the changes that you have made. Fortunately, the EMF code generator provides a simple mechanism for specifying which elements should be replaced by regenerated versions and which should be left alone.

For example, let's add some new behavior, an `isFollowup()` method, to our purchase orders. We simply declare the new method in the `PurchaseOrder` interface:

```

public interface PurchaseOrder ...
{
    boolean isFollowup();
    ...
}

```

Then, we add its implementation to the `PurchaseOrderImpl` class:

```

public class PurchaseOrderImpl ...
{
    public boolean isFollowup() {
        return getPreviousOrder() != null;
    }
}

```

The EMF generator won't wipe out this change because it isn't a generated method to begin with. Every method and field generated by EMF includes a Javadoc comment that contains an `@generated` tag, like this:

```
/** ...
 * ...
 * @generated
 */
public String getComment() {
    return comment;
}
```

Any method in the file that doesn't carry this tag, like `isFollowup()`, will be left untouched whenever we regenerate. In fact, if we want to change the implementation of a generated method, we can do that by removing¹⁷ the `@generated` tag from it¹⁸:

```
/** ...
 * ...
 * @generated
 */
public String getComment() {
    // our custom implementation ...
}
```

Now, because of the missing `@generated` tag, the `getComment()` method is considered to be user code. If we regenerate the model, the generator will detect the collision and discard the generated version of the method.

Actually, before discarding a generated method, the generator first checks if there is another generated method in the file with the same name, but with "Gen" appended. If it finds one, then it redirects the body of the newly generated version there, instead of just discarding it. For example, if we want to extend the generated `getComment()` implementation instead of completely discarding it, then we can do that by simply renaming it, like this:

```
/** ...
 * ...
 * @generated
 */
public String getCommentGen() {
    return comment;
}
```

We then add our override as a non-generated method. We can make it do whatever we want:

```
public String getComment() {
    String result = getCommentGen();
    if (result == null)
        result = ... ;
    return result;
}
```

If we regenerate, the generator will detect the collision with our user version of `getComment()`, but because we also have the `@generated` version in the class, it will redirect the newly generated implementation to `getCommentGen()` instead of discarding it.

When we regenerate, the comment associated with a generated interface, class, method, or field always replaces the existing comment in the file. However, every generated Javadoc comment includes a section, delimited by `<!-- begin-user-doc -->` and `<!-- end-user-doc -->`, where you can enter arbitrary text that will be retained during regeneration.

¹⁷A convention for "virtually" removing `@generated` tags is to append the word NOT. We'll use this trick extensively in Chapter 12.

¹⁸If you know ahead of time that you're going to want to provide your own custom implementation for some feature, then a better approach is to model the attribute as volatile. As we have seen, this instructs the generator to generate only a skeleton method body in the first place, which you are then expected to implement.

The user section of the Javadoc comment associated with an interface may also contain a special control directive used during merging: the `@extends` tag, followed by a comma-separated list of interfaces. The generated interface's `extends` clause will be merged with this list. This allows you to add non-generated super-interfaces, without removing the `@generated` tag.

Chapter 10. EMF. Edit Generator Patterns

Attempting to generate an interactive application is a fairly tricky undertaking. In generating a data model implementation, the problem is well defined and well contained: to be useful, the implementation must correctly and efficiently realize the semantics of the model. Its external interfaces need to be well understood by the developer who wishes to use it as a component in the application being built.

By comparison, the problem that EMF.Edit tries to solve is much more open-ended. The code that it generates is meant to be a starting point for the development of an application. Its modification, not just its use, will be necessary.

Thus, EMF.Edit must suggest a design and provide utilities that integrate well with its environment, while offering enough flexibility so that it can be molded into the developer's desired application. After all, we may think we're pretty smart, but there's no way that we can predict all of the things that you might want your editor to become.

In this chapter, we explore the various patterns employed in generating code for the edit and editor plug-ins for an EMF model, discussing how the generated code functions and how it fits into the overall EMF.Edit design. We will see how some details are hidden in generic framework code that uses Ecore's reflective API, and how others are pushed right to the surface to make it easy to modify visible behavior. And, we will offer suggestions about how these hooks can be used, so that you can start customizing your editor.

As in Chapter 9, we will use the primer purchase order model as our example, showing and discussing the code that is generated from it.

Like the EMF.Edit framework itself, generated EMF.Edit code is divided into two parts: UI-independent code is placed in an edit plug-in, and UI-dependent code is placed, by default, in a separate editor plug-in. The UI-independent portion facilitates the viewing and editing of model objects by providing the item providers that we first discussed in Chapter 3. The following types of classes are generated in the edit plug-in:

- Item provider
- Item provider adapter factory
- Plug-in

The UI-dependent portion uses the Eclipse workbench framework, including JFace and SWT, to provide the interface for creation and editing of EMF models. The following types of classes are generated in this plug-in:

- Editor
- Action bar contributor
- Wizard
- Plug-in

In addition to Java code, the generator produces the following content for each of the two EMF.Edit plug-ins:

- A manifest file, named *plugin.xml*
- A property file, named *plugin.properties*
- An *icons*/subdirectory

As we describe the generated code patterns, we will also need to make reference to these files.

Item Providers

We introduced item providers in Section 3.2, describing them as adapters that provide the interfaces to support viewing and editing of model objects. At that time, we said they are the most important objects in EMF.Edit, so it's not surprising that we will focus heavily on them now. It is the item providers that determine how each type of object will be displayed and how it will respond to editing commands; we need to understand how the generated item provider code does this, and how to add to or modify it effectively.

By default, a different item provider implementation is generated for each class in the core model. The inheritance hierarchy of these classes follows that of the model. Recall from Chapter 9 that, although EMF supports multiple interface inheritance, each class can only extend one implementation base class, and any structural features inherited from other superclasses will be re-implemented in the derived implementation class. The same pattern will be followed among item providers: if class B extends class A, then B's item provider class will extend A's item provider class and will take responsibility for all of the features introduced in B and any of its mixin interfaces. However, we can specify in the generator model that generation of the item provider for any given class should be suppressed, in which case the item provider for its derived class will extend that of *its* implementation base class.

In much the same way as implementations for roots in the model's class hierarchy extend `EObjectImpl`, their item providers extend a framework class called `ItemProviderAdapter`. The Template Method design pattern [5] is used extensively in this base class, as it implements most of the default item provider functionality itself, and calls to methods for which overrides are generated to supply details specific to the particular model or class. As we discuss the generated methods, we will also have to describe how they fit into the template methods of the base class.

Throughout this section, we will look at the item provider that gets generated for the `PurchaseOrder` class, and see how it works with its base class to perform the four item provider roles that we described in Chapter 3. As a reminder, those roles are as follows:

1. Implement content and label provider functions
2. Provide a property source (property descriptors) for EMF objects
3. Act as a command factory for commands on their associated model object
4. Forward EMF model change notifications on to viewers

We begin with the class declaration:

```
public class PurchaseOrderItemProvider extends ItemProviderAdapter
    implements IStructuredItemContentProvider,
               ITreeItemContentProvider,
               IItemLabelProvider,
               IItemPropertySource,
               IEditingDomainItemProvider
```

As we've already emphasized, the base class is `ItemProviderAdapter`. We also notice that five interfaces are implemented, corresponding collectively to the item provider roles. In the following subsections, we will discuss the methods that are generated to help implement these interfaces.

Content and Label Provider

As we described in Section 3.2.1, item providers support the implementation of content and label providers for the viewers by implementing `ITreeItemContentProvider`, `IStructuredItemContentProvider`, `IItemLabelProvider`, and `ITableItemLabelProvider`.

We begin with `ITreeItemContentProvider`. Believe it or not, implementing the four methods declared by this interface really comes down to answering the following question: What will we consider to be the children of a given object? This decision is exposed in the generated item provider, as follows:

```
public Collection getChildrenReferences(Object object) {
    if (childrenReferences == null) {
        super.getChildrenReferences(object);
        childrenReferences.add(
            PPOPackage.eINSTANCE.getPurchaseOrder_Items());
        childrenReferences.add(
            PPOPackage.eINSTANCE.getPurchaseOrder_BillTo());
        childrenReferences.add(
            PPOPackage.eINSTANCE.getPurchaseOrder_ShipTo());
    }
    return childrenReferences;
}
```

Here we see the answer for purchase orders: a purchase order's children will include all of the objects contained via its `items`, `billTo`, and `shipTo` references. By default, the set of children of an object includes all objects accessible through its containment references. However, we can specify in the generator model that only a given subset of containment references should be considered to contribute children; the generated code simply adds these references to the `childrenReferences` list.

There still remains some distance between deciding which references correspond to an object's children and implementing the `getParent()`, `getChildren()`, `hasChildren()`, and `getElements()` methods declared by the interface. However, once we have decided that the child relationship corresponds to containment and specified the subset of containment references from which children are to be obtained, these methods become generic, straightforward applications of the reflective `EObject` API. They are implemented by the base class, `ItemProviderAdapter`, as follows:

- `getParent()` returns the container of the object; if the object is not contained but belongs to a resource, it returns that resource.
- `getChildren()` iterates over the `childrenReferences` list, building and returning a collection of the objects it obtains from those references.
- `hasChildren()` returns whether that collection of the objects obtained from the `childrenReferences` is non-empty.
- `getElements()` simply calls `getChildren()` and returns its result.

The last of these methods, `getElements()`, is inherited from `IStructuredItemContentProvider`. The only member of that interface, it returns the objects to display as rows in a table, items in a list, or root-level nodes in a tree, for a given input object. Implementing it via `getChildren()` is certainly a reasonable default.

It's also a good candidate for customization, as you might very well want to define some other relationship between an input object, such as the selection, and the objects that get displayed in a given viewer. Moreover, EMF.Edit's interpretation of "children" as objects accessible through containment references is not cast in stone. You could certainly override `getParent()` and `getChildren()` to provide a different interpretation. One good approach is to add an additional set of item providers that offers an alternate view on the model, an idea that we will discuss further in Section 10.3.

We now turn our attention to providing labels: the `IItemLabelProvider` interface includes the `getImage()` and `getText()` methods, which return an icon and a text label, respectively, for a given object. The following implementations are generated in `PurchaseOrderItemProvider`:

```
public String getImage(Object object) {
    return getResourceLocator().getImage(
        "full/obj16/PurchaseOrder");
}

public Object getText(Object object) {
    String label = ((PurchaseOrder)object).getComment();
    return label == null || label.length() == 0 ?
        getString("_UI_PurchaseOrder_type") :
        getString("_UI_PurchaseOrder_type") + " " + label;
}
```

The call to `getResourceLocator()` returns the object directly responsible for supplying text and image resources—such an object implements the `ResourceLocator` interface. This method is defined in `ItemProviderAdapter` to return the EMF.Edit plug-in class, but an override is generated as follows:

```
public ResourceLocator getResourceLocator() {
    return PrimerPOEditPlugin.INSTANCE;
}
```

This returns the generated plug-in class that has access to the text and image resources specific to our purchase order model. This class looks for image resources in the `icons`/subdirectory of the plug-in's location; our call to `getImage()` returns a generated icon representing a purchase order.

The `getString()` method is implemented in `ItemProviderAdapter` to look up an externalized string for the given key, using the `ResourceLocator` returned by `getResourceLocator()`. Eclipse and EMF use Java's `ResourceBundle` mechanism to supply externalized strings. Each plug-in can include a property file, called `plugin.properties`, that lists key-value pairs, where the values are localized strings. In the `getText()` method above, we're looking for the string keyed by `_UI_PurchaseOrder_type`, which is generated as follows:

```
_UI_PurchaseOrder_type = Purchase Order
```

The generated property file includes one such string for each class and each feature in the core model.

So, the `getText()` method retrieves a string that gives the type of the object, in this case "Purchase Order". It appends to that the value of one of the object's attributes, in this case, `comment`, and returns the result. The particular attribute to use is selected in the generator model; by default, a single-valued attribute, preferably one whose name is or ends with "name", is chosen.

We have now seen several opportunities to customize the label provider behavior of your item providers. Beyond setting the `label` attribute for a class in the generator model, you can change the externalized string that identifies the class, by editing the property file, and replace the generic icon that gets generated for the class. Or, you could just remove the `@generated` tag and then change the implementations of `getImage()` and `getText()`, so that they return something else entirely.

You may have noticed that there was one interface introduced in Section 3.2.1 that is not implemented by `PurchaseOrderItemProvider`: `ITableItemLabelProvider`, which supports label providers for a table viewer. Its two methods, `getColumnImage()` and `getColumnText()`, are not implemented in `ItemProviderAdapter`, either. Instead, the following trick is used in the label provider, `AdapterFactoryLabelProvider`: If it fails to obtain an adapter that supports `ITableItemLabelProvider`, it tries for `IItemLabelProvider` and just delegates the calls to `getImage()` and `getText()`, dropping the parameter that specifies a column. We see the effect in the **Table** page of the generated editor: each row in the table is populated with one value repeated in each column. You can easily improve on this behavior by changing the item provider to implement `ITableItemLabelProvider`. We'll see exactly how to do this in Section 14.2.2.

Item Property Source

Section 3.2.2 described the rather complex object interactions through which item providers populate a property sheet for an object. Fortunately, the role of the item provider itself is quite simple: it only needs to create and return a list of `ItemPropertyDescriptor`s, which, you may recall, get wrapped by `PropertyDescriptor`s to implement the `IPropertyDescriptor` interface required by the property sheet.

`PurchaseOrderItemProvider` implements `getPropertyDescriptors()`, one of the three methods of `IItemPropertySource`, with the following generated code:

```
public List getPropertyDescriptors(Object object) {
    if (itemPropertyDescriptors == null) {
        super.getPropertyDescriptors(object);
        addCommentPropertyDescriptor(object);
        addOrderDatePropertyDescriptor(object);
    }
    return itemPropertyDescriptors;
}
```

This method calls out to another generated method, which actually creates the `ItemPropertyDescriptor`, for each feature that is considered to be a property. By default, this includes all attributes and non-containment references, although, once again, we can specify a different choice via the generator model.

The methods that actually create the `ItemPropertyDescriptor`s look like this:

```
protected void addCommentPropertyDescriptor(Object object) {
    itemPropertyDescriptors.add(
        new ItemPropertyDescriptor(
            ((ComposeableAdapterFactory)adapterFactory).
                getRootAdapterFactory(),
            getString("_UI_PurchaseOrder_comment_feature"),
            getString("_UI_PropertyDescriptor_description",
                "_UI_PurchaseOrder_comment_feature",
                "_UI_PurchaseOrder_type"),
            PPOPackage.eINSTANCE.getPurchaseOrder_Comment(),
            true,
            ItemPropertyDescriptor.GENERIC_VALUE_IMAGE));
}
```

They simply use `new` to create the appropriate descriptor, and everything else is taken care of automatically: the feature value will be accessed via the reflective `EObject` API, and an editor widget appropriate for its type will be supplied.

The item provider only needs to supply the following information, as parameters to the `ItemPropertyDescriptor` constructor:

- The adapter factory that created this item provider, or its root, if adapter factory composition is being used¹
- The label to be displayed for the property, which is also used as the property's ID
- The description, which displays in the status bar when the property is selected
- The feature on which the property is based
- Whether to allow editing of the property
- The icon to be displayed with the property

In the generated code, externalized strings are used to supply the label and description. The label is the name of the feature, "Ship To", which is retrieved in the same manner as the class name was in `getText()` above. The description is obtained from the three-parameter form of `getString()` that is defined by the `IItemProviderAdapter` base class. This method gets externalized strings for its second and third arguments using the single-parameter form, and then substitutes those results into the string retrieved for its first. To be concrete, the generated resource file includes the following:

```
_UI_PropertyDescriptor_description = The {0} of the {1}
```

EMF uses `java.text.MessageFormat` to perform substitution in externalized strings. Even if you're not already familiar with this mechanism, you can probably guess that "{0}" and "{1}" mark where the two substitutions should occur. The resulting string will be "The Comment of the Purchase Order".

By default, whether editing of the property is allowed depends on the feature's `changeable` attribute, but this can be controlled via the generator model.

For attributes, the icon is selected based on the type of the feature. For references, the icon parameter is omitted. In this case, the property descriptor gets an adapter for the referenced object that supports `IItemLabelProvider`, which then supplies the icon via `getImage()`.

In addition to `getPropertyDescriptors()`, the `IItemPropertySource` interface includes two other methods, both of which are implemented by the base class. The first, `getPropertyDescriptor()`, returns the property descriptor associated with a given ID. It simply calls `getPropertyDescriptors()` and tests the IDs of the resulting descriptors, returning the first one for which it matches. The other, `getEditableValue()`, allows for some transformation of the value to be edited. In general, this is not needed, so the base class implementation just returns the object passed to it, unchanged.

Once again, many opportunities present themselves for customization. While the generator model provides control over which features of a class correspond to editable and non-editable properties, you may wish to base additional properties on features of other related classes. To do so, you would just need to add code to `getPropertyDescriptors()` to provide the extra property descriptors.

By modifying the calls to the `IPropertyDescriptor` constructor, you can control the label, description, and icon for the property; adding an extra string argument will cause the properties to be grouped into categories. Or, instead of adding `IPropertyDescriptor`s themselves, you can subclass `IItemPropertyDescriptorDecorator` and override behavior to control, for example, when editing is allowed or how to set up the widget to be used for it. An example in Section 14.2.1 will demonstrate both the inclusion of properties for features of other classes and the use of this decorator pattern.

¹We will discuss adapter factory composition in Section 10.2.

Command Factory

In Section 3.3, we described EMF.Edit's command framework for model editing, and mentioned that item providers play a role as the command factory by implementing the `IEditingDomainItemProvider` interface. In particular, `createCommand()` is the method that is called to correctly instantiate command objects, based on the information supplied in a `CommandParameter`. This may involve factoring an `AddCommand`, `RemoveCommand`, or `MoveCommand` that operates on multiple objects into multiple commands, each operating on a single object, and wrapping them in a `CompoundCommand`. It may also involve supplying some additional information to the command.

The `ItemProviderAdapter` base class provides a `createCommand()` implementation that generically supports all of the EMF.Edit commands listed in Section 3.3.2, except for the clipboard commands, which are simple extensions of other commands and require no additional item provider support.

The template method pattern is used again to offer specialized support to the `createCommand()` implementation. If a class has more than one reference that contributes children, the following method will be generated in its item provider:

```
protected EReference getChildReference(Object object,
                                      Object child) {
    return super.getChildReference(object, child);
}
```

This method is called when creating an `AddCommand` or `MoveCommand` when no target reference is supplied. By default, it just invokes the base class implementation, which iterates over the objects returned by `getChildrenReferences()`, described in Section 10.1.1, and returns the first one whose type is compatible with the given child object. However, if we want to ensure that any or all objects get added to a certain reference, we can modify this method to return it.

There are three additional methods that make up the `IEditingDomainItemProvider` interface. As explained in Section 3.3.3, the editing domain uses `getParent()` to find the object owner to whose item provider `createCommand()` is delegated. It is also required by `DragAndDropCommand`, as is `getChildren()`. Fortunately, these two methods are already implemented to support content providers, as we described in Section 10.1.1. The interface's last method, `getNewChildDescriptors()`, forms part of the support mechanism for `CreateChildCommand`, which we will discuss in Section 10.1.5.

Change Notification

We described the item provider's final role in Section 3.2.4: it notifies the model's central change notifier of interesting state changes to its associated model object. As a standard EMF adapter for that object, the item provider is notified of all changes via its `notifyChanged()` method. This method just needs to decide whether the notification is important and, if so, forward it.

The following implementation is generated in `PurchaseOrderItemProvider`:

```
public void notifyChanged(Notification notification) {
    switch (notification.getFeatureID(PurchaseOrder.class)) {
        case PPOPackage.PURCHASE_ORDER__COMMENT:
        case PPOPackage.PURCHASE_ORDER__ORDER_DATE:
        case PPOPackage.PURCHASE_ORDER__ITEMS:
        case PPOPackage.PURCHASE_ORDER__BILL_TO:
        case PPOPackage.PURCHASE_ORDER__SHIP_TO:
    {
        fireNotifyChanged(notification);
        return;
    }
}
```

```

        }
    }
    super.notifyChanged(notification);
}
}

```

Notifications will be forwarded by calling `fireNotifyChanged()` for changes to those features corresponding to the generated case statements: `comment`, `orderDate`, `items`, `shipTo`, and `billTo`. This set of features for which to provide change notification is specified in the generator model. By default, it includes containment references—recall from Section 10.1.1 that these define the content structure—and all of the attributes. If you change the way the content provider methods work, you will probably need to make corresponding changes here as well.

The `ItemProviderAdapter` base class provides the `fireNotifyChanged()` implementation that passes the notification along to the adapter factory, which acts as the model's central change notifier, as well as to its own registered listeners.²

Object Creation

One useful feature of EMF.Edit-based editors is object creation support, which we first saw in action in Section 4.4. Both the context-sensitive pop-up menu and the editor-contributed pull-down menu contain **New Child** and **New Sibling** submenus, under which appear all of the candidate object types appropriate for the selected object. Figure 10.1 illustrates this for a “Purchase Order” in our model: the possible children are **Item**, **Bill To US Address**, and **Ship To US Address**.

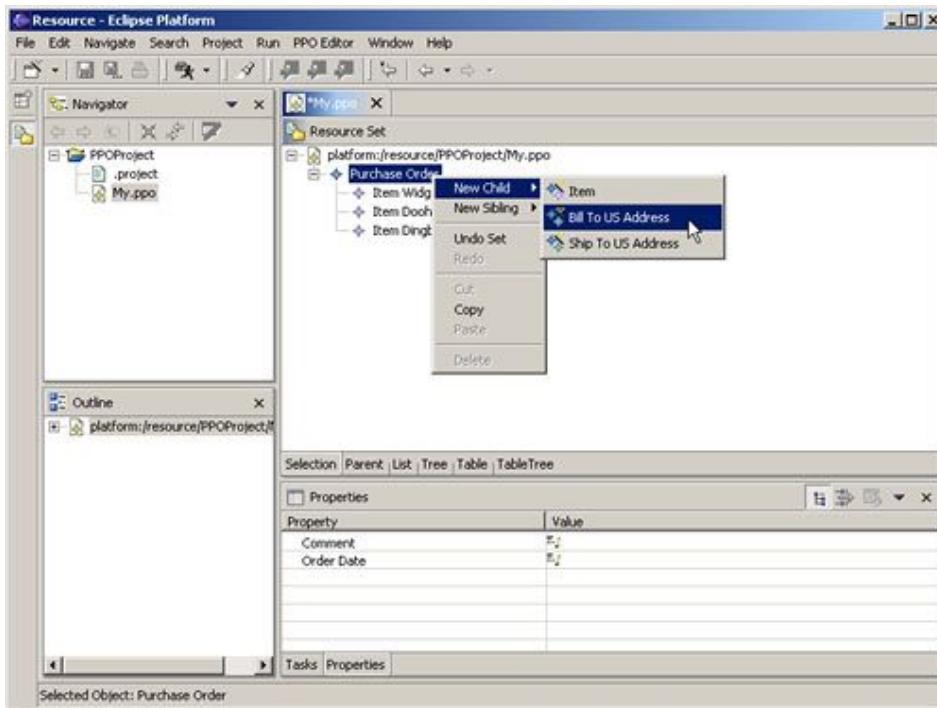


Figure 10.1. Creating child for Purchase Order.

For all other commands, support is needed from the item providers only for creation of the command, as we saw in Section 10.1.3. However, for `CreateChildCommand`, which provides child and sibling³ object creation functionality, the item providers play an important part in three roles:

²The item provider is also an `IChangeNotifier` itself, in order to support the use of the Decorator pattern [5]: an `ItemProviderDecorator` registers as a listener for the item provider that it decorates, so that it can forward notifications to its own adapter factory.

1. Determining candidates for object creation
2. Creating the command
3. Controlling the command's appearance

To understand what these roles mean and why they are required, let's take a top-down look at the object creation support, and compare it with the way in which other commands work.

In Section 3.1.1, we described Eclipse's action mechanism, in which operations are carried out by objects that implement the `IAction` interface. In addition to the behavior provided by an action, it can also control its appearance by providing information to the UI element that represents it, such as the text and icon for a menu item. EMF.Edit provides several standard actions to perform operations like cut, copy, paste, and delete, which all work in the same way. They are registered to be notified when the selection changes, and they respond to such a notification by creating the command that performs the correct operation on the selected objects. The command's creation may be assisted by the editing domain and the relevant item provider, as we described in Section 3.3.3, and the action's behavior is then delegated to that command. The Eclipse workbench maintains "global actions" for these standard operations that uniformly control their appearance; our command-delegating actions are simply registered as handlers for these global actions.

However, the two actions for object creation, `CreateChildAction` and `CreateSiblingAction`, need to behave somewhat differently. For any given object, we will need one action for each candidate we wish to offer for child and sibling creation. This may mean zero, one, or more actions, with each delegating to a `CreateChildCommand` that adds that particular object. So, these actions should not be sensitive to selection changes themselves; instead, we need the single object that holds the actions, called the action bar contributor, to respond to selection changes by creating a whole new set of actions for the selected object.

How does the action bar contributor know how many object creation actions to create and how they relate to the different objects that could be created? Not surprisingly, it asks the item providers. The usual pattern is followed, with the request being made of the editing domain:

```
newChildDescriptors = domain.getNewChildDescriptors(object, null);
newSiblingDescriptors = domain.getNewChildDescriptors(null, object);
```

The editing domain simply delegates to the item provider using the familiar adapter factory delegation pattern. For child creation, it adapts on the specified object; for sibling creation, it adapts on the parent of the specified sibling:

```
public Collection getNewChildDescriptors(Object object,
                                         Object sibling) {
    if (object == null) {
        object = getParent(sibling);
    }
    IEditingDomainItemProvider adapter =
        (IEditingDomainItemProvider) adapterFactory.adapt(
            object, IEditingDomainItemProvider.class);
    return editingDomainItemProvider != null ?
        editingDomainItemProvider.getNewChildDescriptors(object,
                                                       this,
                                                       sibling) :
        Collections.EMPTY_LIST;
}
```

³Sibling creation is implemented via a `CreateChildCommand` for which the owner is the selection's parent.

From the point of view of the action bar contributor, the result is just a collection of arbitrary objects to use as keys for its child or sibling creation actions. Each of these actions then creates a `CreateChildCommand`, with the appropriate child descriptor object, using the usual command creation mechanism as described in Section 3.3.3. When the item provider handles `createCommand()`, it knows how to interpret the child descriptor object, which it created earlier, in order to return a command that will add the correct object to the correct place in the model.

The implementation of `CreateChildCommand`, itself, relies upon either `SetCommand` or `AddCommand`, depending on whether the child is being created under a multiplicity-one or multiplicity-many feature. As a subclass of the basic framework `CommandWrapper`, it delegates to whatever command is returned by its `createCommand()` method, which is implemented as follows:

```
protected Command createCommand() {
    if (owner == null || feature == null || child == null) {
        return UnexecutableCommand.INSTANCE;
    }
    if (feature.isMany()) {
        return index == NO_INDEX ?
            AddCommand.create(domain, owner, feature, child) :
            AddCommand.create(domain, owner, feature, child, index);
    }
    else if (owner.eGet(feature) == null) {
        return SetCommand.create(domain, owner, feature, child);
    }
    else {
        return UnexecutableCommand.INSTANCE;
    }
}
```

Since there are no global actions to control the appearance of the object creation actions, they need to provide their own sensible implementations for methods like `getText()`, `getToolTipText()`, and `getImage()`. The `CreateChildCommand` implements the `CommandActionDelegate` interface so that the actions may also delegate these methods to it. It also provides constructor forms that take a helper object, implementing the `CreateChildCommand.Helper` interface, to handle these matters of appearance in a model-specific way. When an item provider creates a `CreateChildCommand`, it passes itself as the helper, so that it ultimately controls the appearance of the action.

We already know how item providers participate in command creation. Now that we have described their other roles in the process of child and sibling creation, let's take a closer look at how the generated code actually fills those roles.

Determining Candidates

Item providers determine which objects can be created as children of an object by implementing the `getNewChildDescriptors()` method of the `IEditingDomainItemProvider` interface. In addition to each possible object itself, this method also needs to specify the reference of the parent object to which it would be added. The parent may be able to accommodate the same type of object under more than one reference; in order to expose this choice to the user, we need to include the reference in the description of the child to be created. Moreover, if the feature is multiplicity-many, we may wish to specify an index at which the object would be added. To represent these different aspects of each child, we use an instance of `CommandParameter`, which we first met in Section 3.3.2, as the descriptor.

The `ItemProviderAdapter` base class provides an implementation of `getNewChildDescriptors()` that creates a new list and passes it, along with the parent object, to `collectNewChildDescriptors()` to be populated. This method is overridden in the generated item providers; for example, in `PurchaseOrderItemProvider`, the following is generated:

```

protected void collectNewChildDescriptors(
    Collection newChildDescriptors, Object object) {
super.collectNewChildDescriptors(newChildDescriptors, object);
newChildDescriptors.add(createChildParameter(
    PPOPackage.eINSTANCE.getPurchaseOrder_Items(),
    PPOFactory.eINSTANCE.createItem()));
newChildDescriptors.add(createChildParameter(
    PPOPackage.eINSTANCE.getPurchaseOrder_BillTo(),
    PPOFactory.eINSTANCE.createUSAddress()));
newChildDescriptors.add(createChildParameter(
    PPOPackage.eINSTANCE.getPurchaseOrder_ShipTo(),
    PPOFactory.eINSTANCE.createUSAddress()));
}

```

We use a convenience method of the base class, `createChildParameter()`, to create a `CommandParameter` for each parent feature/child object pair, which we add to the collection of descriptors.

The generator considers the same set of containment references that were specified as contributing children by `getChildrenReferences()`, which we discussed in Section 10.1.1. For each of these references, it finds the compatible classes: the class corresponding to its type and all of its subclasses. Code is generated to add a descriptor for each non-abstract class compatible with each child-providing reference. In our case, we have three such references—`items`, `billTo`, and `shipTo`—and one compatible class for each.

The list of descriptors that is finally returned to `getNewChildDescriptors()` will then be modified by the base class implementation as follows: if a sibling object is specified, then for each `CommandParameter` with a multi-valued feature, the index is set so that the new object will be added following the sibling as closely as possible.

Controlling Appearance

As we described above, `CreateChildCommand` declares an inner interface, `CreateChildCommand.Helper`, which a delegate can implement in order to control matters of appearance in a model-specific way. `ItemProviderAdapter` implements this interface and, in `createCreateChildCommand()`, passes itself as an argument to the `CreateChildCommand` constructor.

`ItemProviderAdapter` also provides default implementations for the helper methods. For `getCreateChildText()`, `getCreateChildToolTipText()`, and `getCreateChildDescription()`, externalized strings are retrieved from the generated edit plug-in's property file, based on the classes and references involved. The `ResourceLocator` mechanism described in Section 10.1.1 is used for this. Similarly, `getCreateChildImage()` returns an appropriate icon from the model plug-in's *icon*/directory. Finally, `getCreateChildResult()` takes the new child object and returns, as the collection of objects that should be considered the command's result,⁴ a singleton list of that same object.

A generated item provider may include an override of the `getCreateChildText()` method. The simple string returned by the base class implementation, the name of the class of the child object, is generally sufficient. However, if a given parent class includes multiple references contributing children that share compatible classes, then we generate an override that uses the feature names to distinguish among them. For example, in `PurchaseOrderItemProvider`, the following is generated:

⁴Recall from Section 3.3.1 that the result of a command can be used in compound commands as the input to its following command. In addition, `CreateChildCommand` returns this same result when `getAffectedObjects()` is called after an `execute()` or `redo()`.

```

public String getCreateChildText(Object owner, Object feature,
                                 Object child, Collection selection) {
    boolean qualify =
        feature == PPOPackage.eINSTANCE.getPurchaseOrder_BillTo() ||
        feature == PPOPackage.eINSTANCE.getPurchaseOrder_ShipTo();
    return getString(
        qualify ? "_UI_CreateChild_text2" : "_UI_CreateChild_text",
        new Object[] {
            getTypeText(child),
            getFeatureText(feature),
            getTypeText(owner) });
}

```

Here, we see that the class name needs to be qualified if the feature is **billTo** or **shipTo**, since they share **USAddress** as their type. We are using the general form of `getString()`, which uses the objects specified in its array argument for the substitutions indicated by the externalized string. The two relevant strings from the generated property file are as follows:

```

_UI_CreateChild_text = {0}
_UI_CreateChild_text2 = {1} {0}

```

The second includes the feature name as qualification, while the first does not.

Customizing object creation can begin with changing the externalized strings and replacing the generic icons that are used by the `Helper` methods. Beyond that, you can add to or remove from the options offered for children of a particular class by modifying its item provider's implementation of `collectNewChildDescriptors()`. If you wish to create objects in a state other than their default, you can add initialization of the child objects that are included in the descriptors. You could even offer two different children that are instances of the same class, but initialized differently, provided that you override the `Helper` methods to distinguish between them and label them accordingly.

Item Provider Adapter Factories

In order to create the item providers for the classes defined in a model, an item provider adapter factory is generated for each model package. This class extends the adapter factory utility class generated as part of the model plug-in. The declaration for the item provider adapter factory for our primer purchase order model is as follows:

```

public class PPOItemProviderAdapterFactory extends PPOAdapterFactory
    implements ComposeableAdapterFactory, IChangeNotifier

```

As we described in Chapter 9, implementing an adapter factory essentially involves answering two questions:

- **Which adapter types are supported by the adapter factory?** An adapter type can be any arbitrary object; any significance is imposed by an adapter and its factory.
- If a given target object has no adapter of a given supported type, how should a new adapter be created?

Recall that the generated adapter factory implementation for a model package extends the `AdapterFactoryImpl` framework class, overriding the `isFactoryForType()` and `createAdapter()` methods to answer those questions. The supported types include the Ecore package for the model and instances of all classes in that package. Adapter creation is implemented, using the generated switch class and the Factory Method pattern [5], to switch on the Ecore class of the target and call out to a series of methods corresponding to the inheritance hierarchy for that class, moving

upward until one returns a non-null value. These factory methods are all implemented to return null, with the intention that they will be overridden in subclasses to return the desired adapters. Indeed, we will soon see that our generated item provider adapter factory does exactly that, overriding the methods that correspond to the classes for which we want item providers.

First, let's look at how the item provider adapter factory overrides the implementation of `isFactoryForType()` to define which types it supports. Returning to the generated `PPOItemProviderAdapterFactory` class, we find the following:

```
protected Collection supportedTypes = new ArrayList();
...
public PPOItemProviderAdapterFactory() {
    supportedTypes.add(IItemContentProvider.class);
    supportedTypes.add(ITreeItemContentProvider.class);
    supportedTypes.add(IItemPropertySource.class);
    supportedTypes.add(IEditingDomainItemProvider.class);
    supportedTypes.add(IItemLabelProvider.class);
    supportedTypes.add(ITableItemLabelProvider.class);
}
...
public boolean isFactoryForType(Object type) {
    return supportedTypes.contains(type) ||
        super.isFactoryForType(type);
}
```

Notice that we use the implemented item provider interfaces as the supported adapter types. You may be wondering about the call to the superclass method in the second half of the boolean expression. For some reason, we also want to return `true` if the type is the Ecore package for the model, as we described above, but we will not be able to explain that reason until the end of this section.

For now, let's return to the factory method overrides that actually create the item provider adapters. One will be generated for each item provider class in our model, and the form of each will depend on whether the generator model specifies to use the Stateful or Singleton item provider pattern for that class.

As we described in Section 3.2.5, the Stateful pattern involves creating a new item provider, which can carry additional state information for each object in an instance model. In our generated `PPOItemProviderAdapterFactory`, the factory method for stateful purchase order adapters would simply look like this:

```
public Adapter createPurchaseOrderAdapter() {
    return new PurchaseOrderItemProvider(this);
}
```

If, however, we opted to use the Singleton pattern, a single instance of the item provider would be used for all objects of this type. It could only define behavior, accessing the information stored in the model objects themselves. The following code would be generated, instead:

```
protected PurchaseOrderItemProvider purchaseOrderItemProvider;

public Adapter createPurchaseOrderAdapter() {
    if (purchaseOrderItemProvider == null) {
        purchaseOrderItemProvider =
            new PurchaseOrderItemProvider(this);
    }
    return purchaseOrderItemProvider;
}
```

Note that in both cases, the item provider adapter factory passes itself to the constructor, ensuring that item providers can always find the factories that created them.

Beyond the ordinary EMF adapter factory behavior discussed above, item provider adapter factories in EMF.Edit have two additional responsibilities:

- Providing change notification for the model by implementing `IChangeNotifier`
- Supporting composition of the model with others by implementing `ComposeableAdapterFactory`

The first of these, which we have already discussed in Section 3.2.4, requires little further explanation. The implementation is delegated to an instance of `ChangeNotifier`, a framework class that maintains a list of listeners to which notifications are sent, as follows:

```
protected IChangeNotifier changeNotifier = new ChangeNotifier();

public void addListener(
    INotifyChangedListener notifyChangedListener) {
    changeNotifier.addListener(notifyChangedListener);
}

public void removeListener(
    INotifyChangedListener notifyChangedListener) {
    changeNotifier.removeListener(notifyChangedListener);
}
```

We'll see the implementation of the remaining `IChangeNotifier` method, `fireNotifyChanged()`, after some enlargement on the topic of model composition. EMF.Edit supports editors for objects from more than one EMF model by applying the Composite pattern [5] to the item provider adapter factories for those models. An instance of the `ComposedAdapterFactory` convenience class acts as a single access point for the content and label providers and the editing domain. It maintains a list of adapter factories that implement the `ComposeableAdapterFactory` interface, forwarding its clients' requests to the appropriate adapter factory child and forwarding all of its children's change notifications to its registered listeners.

In order to allow the item provider adapter factory generated for any model to be reused in this way, the `ComposeableAdapterFactory` interface is always implemented, as follows:

```
protected ComposedAdapterFactory parentAdapterFactory;

public void setParentAdapterFactory(ComposedAdapterFactory parent)
{
    this.parentAdapterFactory = parent;
}

public ComposeableAdapterFactory getRootAdapterFactory() {
    return parentAdapterFactory == null ?
        this : parentAdapterFactory.getRootAdapterFactory();
}
```

The `setParentAdapterFactory()` method is called by a `ComposedAdapterFactory` when the item provider adapter factory is added to or removed from its list of children. The `parentAdapterFactory` reference is required by `fireNotifyChanged()`. In addition to notifying registered listeners, it needs to pass the notification up to the parent, if set:

```

public void fireNotifyChanged(Notification notification) {
    changeNotifier.fireNotifyChanged(notification);
    if (parentAdapterFactory != null) {
        parentAdapterFactory.fireNotifyChanged(notification);
    }
}

```

Finally, we can now explain that mysterious call to the superclass method in the item provider adapter factory's definition of `isFactoryForType()`. To select which child should be used as a delegate, `ComposedAdapterFactory`'s implementation of `adapt()` actually makes two calls to `isFactoryForType()`: one for the given type, and another for the Ecore package of the given target, if it is an `EObject`. In order to be selected, the adapter factory must return `true` for both.

In general, it is not necessary to do any customization of the generated item provider adapter factories by hand. If you wish to provide item providers for classes that are not a part of the model, you can handle them with a separate item provider adapter factory and include both factories in a `ComposedAdapterFactory`.

Editor

As we now begin to look at the code that depends upon the Eclipse workbench UI framework, our approach will change from the previous sections. Rather than aiming for completeness, we will focus only on the code that anchors the EMF.Edit mechanisms that we have already seen in this and previous chapters. Though much of the remainder of the generated code would likely serve as a useful example of Eclipse UI programming, we consider that discussion to be outside of the scope of this book.

We start with the editor class itself. This class implements the workbench part through which the user can edit instances of the EMF model. The plug-in's generated manifest file registers it as the editor for such documents⁵ by declaring an extension to the `org.eclipse.ui.editors` extension point.

The editor is central to the EMF.Edit-based design in that it creates most of the important participants that we described in Chapter 3, including the adapter factory, command stack, editing domain, viewers, content providers, and label providers. The generated implementation extends the `MultiPageEditorPart` workbench abstract base to provide multiple, tabbed pages, using one for each of its viewers.

For our purchase order model, the editor's class declaration is as follows:

```

public class PPOEditor extends MultiPageEditorPart implements
    IEditingDomainProvider, ISelectionProvider, IMenuListener

```

As an `ISelectionProvider`, the editor passes along notifications when the selection changes in its active viewer. This mechanism is used by the global action handlers to create new commands and by the action bar contributor to create new object creation actions, as described in Section 10.1.5. As an `IMenuListener`, the editor is notified when a pop-up menu is about to show in one of its viewers. It then calls on its action bar contributor to populate this menu with items for the global and object creation actions.

Since the editor uses a single adapter factory, command stack, and editing domain, these are created in the constructor. In the previous section, we discussed composition of adapter factories. The generated editor uses a `ComposedAdapterFactory` rather than the model's factory itself, so that its views may include objects that are not part of the model:

⁵That is, it is registered as the editor for files with a filename extension that was chosen by the generator for instances of the model. That extension is just the lowercased package prefix.

```

protected ComposedAdapterFactory adapterFactory;
protected AdapterFactoryEditingDomain editingDomain;

public PPOEditor() {
    super();
    List factories = new ArrayList();
    factories.add(new ResourceItemProviderAdapterFactory());
    factories.add(new PPOItemProviderAdapterFactory());
    adapterFactory = new ComposedAdapterFactory(factories);
    ...
}

```

In particular, `ResourceItemProviderAdapterFactory` is also included. It creates item providers for `Resource` and `ResourceSet`, which manage the persistent form of instance models. Thus, the views are able to include items representing the resource in which the model resides and the set of associated resources that could contain cross-referenced objects.

Next, the constructor creates a command stack for the editor. It uses the `BasicCommandStack` implementation from the common command framework, which we discussed in Section 3.3.1, and attaches a listener to it:

```

BasicCommandStack commandStack = new BasicCommandStack();
commandStack.addCommandStackListener(
    new CommandStackListener()
{
    public void commandStackChanged(final EventObject event) {
        getContainer().getDisplay().asyncExec(
            new Runnable()
            {
                public void run() {
                    firePropertyChange(IEditorPart.PROP_DIRTY);
                    Command mostRecentCommand =
                        ((CommandStack)event.getSource()).
                            getMostRecentCommand();
                    if (mostRecentCommand != null) {
                        setSelectionToViewer(
                            mostRecentCommand.getAffectedObjects());
                    }
                }
            });
    }
});
...

```

Whenever a command is executed, undone, or redone, the property sheet will be updated and the selection changed to whatever the command returns as its affected objects.

Finally, the constructor creates the editing domain, passing it the adapter factory and command stack:

```

editingDomain = new AdapterFactoryEditingDomain(adapterFactory,
                                                commandStack);
}

```

During its initialization, the editor's base class invokes `createPages()`, allowing it to supply the controls that should be contained in each tabbed page. The editor creates its various viewers—each within a pane, which provides a title bar. It also creates an appropriate content and label provider for each viewer, performs any additional initialization that the particular viewer needs, creates a pop-up menu for it, and adds a page for it. The following initialization of the **Selection** page with a simple tree is illustrative:

```

public void createPages() {
    ...

    ViewerPane viewerPane = ...
    selectionViewer = (TreeViewer)viewerPane.getViewer();

    selectionViewer.setContentProvider(
        new AdapterFactoryContentProvider(adapterFactory));
}

```

```

selectionViewer.setLabelProvider(
    new AdapterFactoryLabelProvider(adapterFactory));

selectionViewer.setInput(editingDomain.getResourceSet());
viewerPane.setTitle(editingDomain.getResourceSet());
...
createContextMenuFor(selectionViewer);
int pageIndex = addPage(viewerPane.getControl());

...
}

```

Note that we set the tree's input to the resource set. Recall from Section 10.1.1 that our item providers implement `getElements()` to return the children of the specified object. Thus, as generated, the root object of the tree will be the singular child of the resource set: the resource corresponding to the file in which the model is saved.

As the editor class directly defines the appearance and behavior of the application that the user experiences, it offers virtually limitless possibilities for customization. Most likely, you will want to modify the various viewers, perhaps with the exception of the one that we just examined, to make them show something more useful for your model, or else remove them altogether. The example provided in Section 14.2.2 demonstrates such a modification.

In Section 10.1.1, we mentioned the approach of adding an additional set of item providers to provide an alternate view on the model. To do so, you would also need to write an alternate item provider adapter factory to create them, and you would use that adapter factory in creating the viewer's content and label providers.

Action Bar Contributor

We introduced the action bar contributor in Section 10.1.5 as an object that holds actions. Let's now expand on that brief description. Not only does the action bar contributor define the actions available in a class of editors, but it provides the menu items and toolbar buttons through which these actions are accessed. The action bar to be associated with an editor is named in the extension to the `org.eclipse.ui.editors` extension point that is declared in the plug-in manifest file.

The action bar contributors that we generate extend `EditingDomainActionBarContributor`, an EMF.Edit framework class that registers the command-based handlers for global actions, including cut, copy, paste, delete, undo, and redo. It also populates pop-up menus with items corresponding to these actions, as we mentioned in the previous section.

For our purchase order model, the action bar contributor's class declaration is as follows:

```

public class PPOActionBarContributor
extends EditingDomainActionBarContributor
implements ISelectionChangedListener

```

It implements `ISelectionChangedListener` so that it can respond to changes in the selection, as we described in Section 10.1.5. That is, it queries the appropriate item providers, via the editing domain, for descriptors of the children and siblings that could be created for the selected object:

```

public void selectionChanged(SelectionChangedEventArgs event) {
    ...

    Collection newChildDescriptors = null;
    Collection newSiblingDescriptors = null;

    ISelection selection = event.getSelection();
    if (selection instanceof IStructuredSelection &&
        ((IStructuredSelection)selection).size() == 1) {

        Object object =

```

```

        ((IStructuredSelection)selection).getFirstElement();
EditingDomain domain =
    ((IEDitingDomainProvider)activeEditorPart).
        getEditingDomain();
newChildDescriptors =
    domain.getNewChildDescriptors(object, null);
newSiblingDescriptors =
    domain.getNewChildDescriptors(null, object);
}
...

```

It then creates an action for each descriptor and populates the menus with corresponding menu items:

```

createChildActions = generateCreateChildActions(
    newChildDescriptors, selection);
createSiblingActions = generateCreateSiblingActions(
    newSiblingDescriptors, selection);

if (createChildMenuManager != null) {
    populateManager(createChildMenuManager,
                    createChildActions, null);
    createChildMenuManager.update(true);
}
if (createSiblingMenuManager != null) {
    populateManager(createSiblingMenuManager,
                    createSiblingActions, null);
    createSiblingMenuManager.update(true);
}

```

Note that `createChildMenuManager` and `createSiblingMenuManager` correspond to the **New Child** and **New Sibling** submenus that appear under the editor's pull-down menu. These menus are created in the action bar contributor's `contributeToMenu()` method, which is automatically invoked when the associated editor is loaded. At that time, `contributeToToolBar()` is also called, but our generated action bar contributor adds no buttons to the toolbar. Finally, on the user's right-click, items are added to the pop-up menu in `menuAboutToShow()`. If you wish to add your own actions to an editor, these are the methods that you should modify in order to make those actions available through the menus or toolbar items.

Wizard

In Section 4.4, we saw the simple two-page wizard that creates an instance of an EMF model. This functionality is provided by a generated class that extends the `Wizard` JFace abstract base and is identified in the plug-in's manifest file by an extension to the `org.eclipse.ui.newWizards` extension point.

For our purchase order model, `PPOModelWizard` is generated:

```
public class PPOModelWizard extends Wizard implements INewWizard
```

We will focus on the code that actually creates the model, once the **Finish** button has been clicked. In the `performFinish()` method, we find the following:

```

public boolean performFinish() {
    ...

    ResourceSet resourceSet = new ResourceSetImpl();
    URI fileURI = URI.createPlatformResourceURI(
        modelFile.getFullPath().toString());
    Resource resource = resourceSet.createResource(fileURI);
    EObject rootObject = createInitialModel();
    if (rootObject != null) {

```

```

        resource.getContents().add(rootObject);
    }
    resource.save(Collections.EMPTY_MAP);

    ...
}

```

Recall that an EMF model's persistent form is managed by a `Resource`; adding a model object to the resource's contents ensures that it, and all of its contents, will be included in that persistent form. Just as we first saw in Section 2.5.2, this code uses the basic framework `ResourceSetImpl` to create a `Resource` that serializes into EMF's default XMI format. The resource's filename, relative to the Eclipse workspace, was set according to the user's selection and must be converted to a platform scheme URI. A single root object is added to the resource, and the resource is saved.

The root object is created using the factory, according to the name of the object that the user selected as the model object, as follows:

```

EObject createInitialModel() {
    EClass eClass = (EClass)ppoPackage.getEClassifier(
        initialObjectCreationPage.getInitialEClassName());
    EObject rootObject = ppoFactory.create(eClass);
    return rootObject;
}

```

The wizard also offers many possibilities for customization. You could easily modify `createInitialModel()` to build a more complex model, most likely according to additional information gathered from the user.

Plug-Ins

One singleton plug-in class is generated for each of the edit and editor plug-ins, as a central provider of text and image resources and logging capabilities to all other classes in the plug-in. We have used these classes to access resources at various points throughout this chapter. Now, we'll discuss in more detail how they work.

The Eclipse core includes a class that represents a plug-in, `Plugin`, which can be subclassed and identified in the manifest file using the `class` attribute of the top-level `plugin` element. That class is then instantiated when the plug-in is first used. The plug-in's resource bundle, which is populated from its `plugin.properties` file, is then accessible through the plug-in class, as are the plug-in's location and the platform's logging facilities.

EMF builds on top of Eclipse's plug-in mechanism a more flexible system for providing the needed resources and logging, allowing EMF plug-ins to run within the platform or independently of it. An abstract, common `EMFPlugin` class is defined that implements the `ResourceLocator` and `Logger` interfaces. It includes an abstract `getPluginResourceLocator()` method and a `getPluginLogger()` method that just calls `getPluginResourceLocator()`, casting and returning the same result. For the methods belonging to the two implemented interfaces, `EMFPlugin` checks if the corresponding one of these two methods returns a non-null result. If so, it delegates to that object. Otherwise, it provides its own default implementations: it looks on the classpath for a `plugin.properties` file and `icons/` subdirectory (under the same relative path as the plug-in class file), and prints log entries to `System.err`. This default behavior is provided for the case where the plug-in is running stand-alone, but it can be overridden.

For the case where the plug-in is running within Eclipse, `EMFPlugin` includes a static member class, `EclipsePlugin`, that subclasses Eclipse's `Plugin` class and implements the resource access and logging methods using the platform facilities.

The generated plug-in class for our purchase order's edit plug-in extends `EMFPlugin` and is a singleton, accessible through its static `INSTANCE` field:

```
public final class PrimerPOEditPlugin extends EMFPlugin
{
    public static final PrimerPOEditPlugin INSTANCE =
        new PrimerPOEditPlugin();
    ...
}
```

It also contains a private `plugin` field that can hold an instance of its own `Implementation` static member class, a subclass of `EMFPlugin.EclipsePlugin`. This field is set by `Implementation`'s constructor and returned by `getPluginResourceLocator()`:

```
private static Implementation plugin;

public ResourceLocator getPluginResourceLocator() {
    return plugin;
}
public static class Implementation extends EclipsePlugin
{
    public Implementation(IPluginDescriptor descriptor) {
        super(descriptor);
        plugin = this;
    }
}
```

It is this `Implementation` class that is identified as the plug-in class in the manifest file. Thus, if Eclipse is running, it is instantiated, the `plugin` field is set to it, `getPluginResourceLocator()` and `getPluginLogger()` will return it, and `EMFPlugin` will use it for resource access and logging.

The class for the editor plug-in works exactly the same way, as do those for all of the EMF framework plug-ins.

Chapter 11. Running the Generators

Central to EMF's value as a programming tool is its ability to generate high-quality code. Moreover, the EMF code generator is highly configurable, offering the user a great deal of control over the model and editor implementations that it produces. We can affect the selection of generator patterns, as we have seen over the last two chapters, as well as the naming of classes and their organization in packages and plug-ins. If necessary, we can even customize the templates that specify these patterns, giving us total control over each line of generated code.

This chapter describes the code generation tools provided by EMF. It begins with an overview of the code generation process and a summary of the content produced by it. It then discusses, in detail, both the Eclipse-based generator GUI and the command-line generator tools. It concludes by taking a look inside the generator, at the format of the templates that control the content it produces. This final, advanced topic may be of interest to those who wish to customize EMF's standard code generation patterns and to those starting to think about new ways in which the generator could be used to produce custom content of their own design.

EMF Code Generation

Code generation in EMF is primarily based upon a core model. As we have seen throughout the last two chapters, we generate classes that correspond directly to the packages, classes and enumerated types defined within such a model; the members of these generated classes depend upon attributes, references, operations, and enum literals.

As we first explained in Chapter 2, code generation is also affected by additional information that does not constitute part of the data model itself, but instead resides in a generator model made up of decorators for core model objects. A generator model ties together one or more interdependent core models that are to be generated together and also allows access to other referenced core models, for which no new code is to be generated.

A generator model has as its root a **GenModel** object, which has attributes for the generator options that apply globally to the model. It also includes references to two sets of **GenPackage** objects: one set decorates the **EPackage** objects that are the roots of the core models for which code is to be generated, and the other decorates the roots of the referenced core models. Like **GenPackage**, the remainder of the classes that make up the generator model decorate their correspondingly named Ecore classes and include attributes for their related generator options.

Generating a file is a two-step process. First, content is generated by a template-driven engine called JET, or Java Emitter Templates. Templates express the EMF code patterns in a JSP-like [7] syntax. The engine transforms each template into the source of a Java class that will mix fixed template data with the model-specific results of call backs; it then compiles, dynamically loads, and uses those classes to emit the desired content. Template class call backs are to the generator model objects, which find the model-specific details either among their own attributes or in their decorated core model objects. To speed up the generation process, templates can be converted into template classes and compiled ahead of time. We refer to these as *static templates*, while we call the raw JSP-like form *dynamic templates*. In the second step of generation, a source merge utility, called

JMerge, merges the emitted content into the pre-existing file, if there is one. Merging of Java files is based on @generated Javadoc tags, as described in Section 9.9. Property files are merged according to their keys; key-value pairs with new keys are added, but existing keys are never replaced. Other files, including images and plug-in manifest files, are neither merged nor replaced; in order to regenerate these files, they must first be manually deleted.

It is the generator model objects that actually drive code generation. They know what content they contribute to each of the three plug-ins—model, edit, and editor—and create the emitters and mergers involved in generating each file. The four generator model classes that create content are **GenModel**, **GenPackage**, **GenClass**, and **GenEnum**. As a result, we refer to four code generation units: model, package, class, and enum.

The following three tables summarize the content that can be produced for each of these units, in the three generated EMF plug-ins. Table 11.1 shows the content for the model plug-in. This includes a plug-in manifest file; a property file; all of the interfaces and classes corresponding to the Ecore packages, classes, and enums that we described in Chapter 9; and an XML Schema that defines the serialized form of instance models for each package.

Table 11.1. Generated Model Content

<i>Unit</i>	<i>Description</i>	<i>File Name</i>	<i>Subpkg.</i>	<i>Opt.</i>
Model	Plug-In Class			Y
	Plug-In Manifest	plugin.xml		N
	Property File	plugin.properties		N
Package	Package Interface	PrefixPackage.java		N
	Package Class	PrefixPackageImpl.java	impl	N
	Factory Interface	PrefixFactory.java		N
	Factory Class	PrefixFactoryImpl.java	impl	N
	Switch	PrefixSwitch.java	util	Y
	Adapter Factory	PrefixAdapterFactory.java	util	Y
	Resource Factory	PrefixResourceFactoryImpl.java	util	Y
	Resource	PrefixResourceImpl.java	util	Y
	XMI Schema	PrefixXMI.xsd		Y
Class	XML Schema	PrefixXML.xsd		Y
	Interface	Name.java		N
Class	Class	NameImpl.java	impl	N
	Enum	Name.java		N

The names of generated classes and interfaces generally depend upon the names of the corresponding core model objects; for packages, these names depend instead on the **prefix** attribute that is specified in the generator model. The Java package in which all of these classes and interfaces reside has the same name as the Ecore package. The Java packages can be made subpackages of a **basePackage**, also an attribute in the generator model. We will see how to modify the prefix and base package in Section 11.2.1. Implementation and utility code resides in a further **impl** or **util** subpackage, as indicated by the fourth column in the table. The final column specifies whether the creation of each piece of content is optional.

As we described in Chapter 10, the generated code in the edit and editor plug-ins requires plug-in classes to provide text and image resources and logging capabilities. Although the generated code for the model plug-in does not require such support, you may wish to have a plug-in class available for use by your additional, hand-written model code. This is why the table includes such a class, but with no file name: it can be generated but, by default, it is not.

Table 11.2 summarizes the UI-independent portion of the editor code and its related content, which is generated in the edit plug-in.

Table 11.2. Generated Edit Content

Unit	Description	File Name
Model	Plug-In Class	EditPlugin.java
	Plug-In Manifest	plugin.xml
	Property File	plugin.properties
Package	Adapter Factory	PrefixItemProviderAdapterFactory.java
Class	Item Provider	NameItemProvider.java

The generated classes all reside in a `provider` subpackage of the corresponding model package. If the model contains more than one package, the first will be selected, by default, to hold the single plug-in class. However, this default can be changed, along with the name of the class itself, as we will see in Section 11.2.

In addition to the content listed in the table, various sample icons are generated to represent the classes and the object creation actions; these are all placed in the `icons`/subdirectory of the project.

Table 11.3 summarizes the UI-dependent portion of the editor code and its related content, which is generated in the editor plug-in.

Table 11.3. Generated Editor Content

Unit	Description	File Name
Model	Plug-In Class	EditorPlugin.java
	Plug-In Manifest	plugin.xml
	Property File	plugin.properties
Package	Editor	PrefixEditor.java
	Action Bar Contributor	PrefixActionBarContributor.java
	Wizard	PrefixModelWizard

The generated classes reside in a `presentation` subpackage of the corresponding model package. Again, the plug-in class goes with the first package by default, and a sample icon for the wizard is created and placed in the `icons`/project subdirectory.

We have already mentioned that the three generated plug-ins can be combined into two, or even just one. In particular, the following combinations are supported:

- Model and edit
- Edit and editor
- Model, edit, and editor

If you try the other remaining combination—model and editor—the code will generate, but that combined plug-in will have a circular dependency with the edit plug-in, which cannot be resolved.

When plug-ins are combined, no conflicting model-level contents are generated. Instead, they are also combined appropriately and included as part of the editor plug-in, if it is among those combined, or the edit plug-in, otherwise.

The Generator GUI

The EMF generator is typically operated through an Eclipse-based GUI, which we first saw in Chapter 4. It is, itself, a simple EMF.Edit-based editor, registered to handle resources with a *.genmodel* extension. As a result, it is opened by double-clicking on any such file in the Navigator or Package Explorer view.

The generator's interface is quite simple: it consists of a single tree viewer, showing the objects in the loaded generator model: the model, packages, classes, structural features, operations, enums, enum literals, and data types. Figure 11.1 shows the generator with a "Supplier" model loaded. We won't look closely at this model, called ExtendedPO3, until Chapter 12; however, it is an illustrative example as it has references into a version of the purchase order model. Notice the different icon for "EPO3", indicating that it is a referenced package of the "Supplier" model.

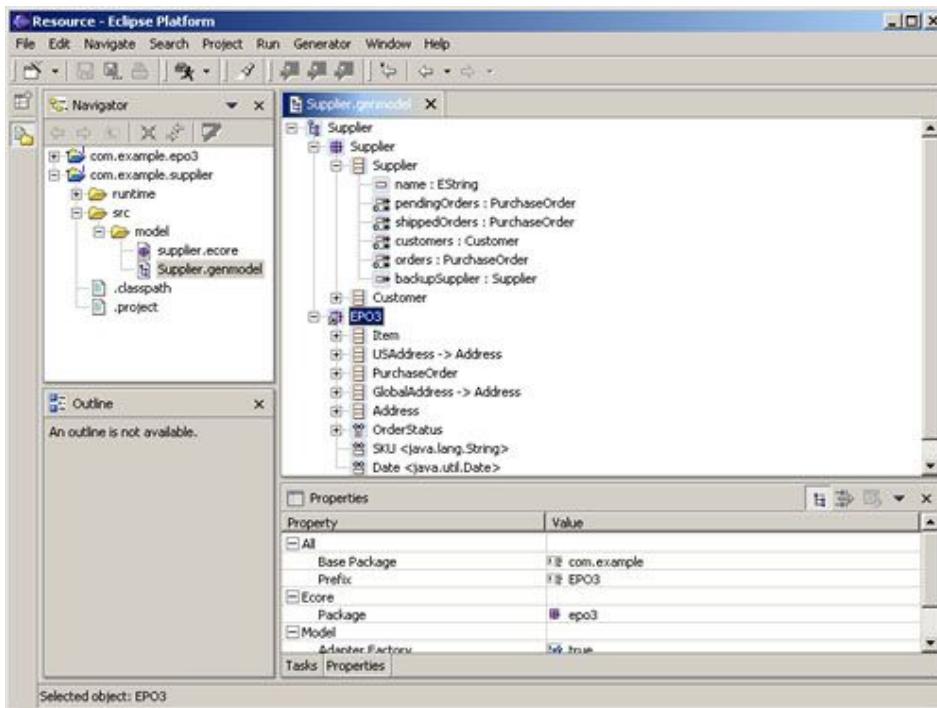


Figure 11.1. EMF generator with "Supplier" model loaded.

The generator interface allows us to do three things:

1. Reload the generator and underlying core models from their original source
2. Modify the generator model attributes
3. Generate code

To reload the models, we simply select **Reload...** from the **Generator** pull-down menu, which is illustrated in Figure 11.2. This launches the same wizard that was used to create the core and generator models initially. We can then step through the wizard, reviewing the selections we made at that time and changing anything, if desired. When we click **Finish**, the core models are updated. The generator model is also updated to reflect structural changes, while existing attribute settings are maintained.

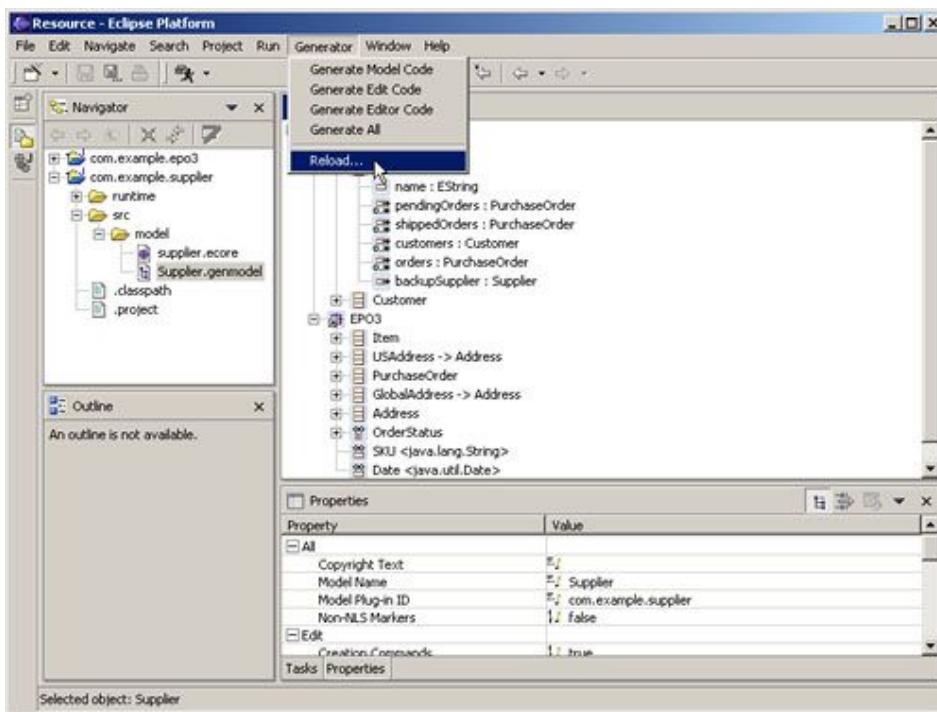


Figure 11.2. Generator pull-down menu.

As different objects are selected in the generator's tree viewer, their attributes are displayed in the Properties view. These properties give us the ability to control what code is generated, how it is organized, and what patterns are used in it; we discuss them in detail in the following subsection. It is worth noting that the referenced packages, like “EPO3”, are actually references into other generator models. The attributes of any objects under such packages cannot be edited, so the properties are read-only. In order to change these attributes, you will need to edit the referenced generator model directly.

Code is generated by selecting an object from the generator model, right-clicking, and selecting **Generate Model Code**, **Generate Edit Code**, **Generate Editor Code**, or **Generate All** from the pop-up menu, as illustrated in Figure 11.3.

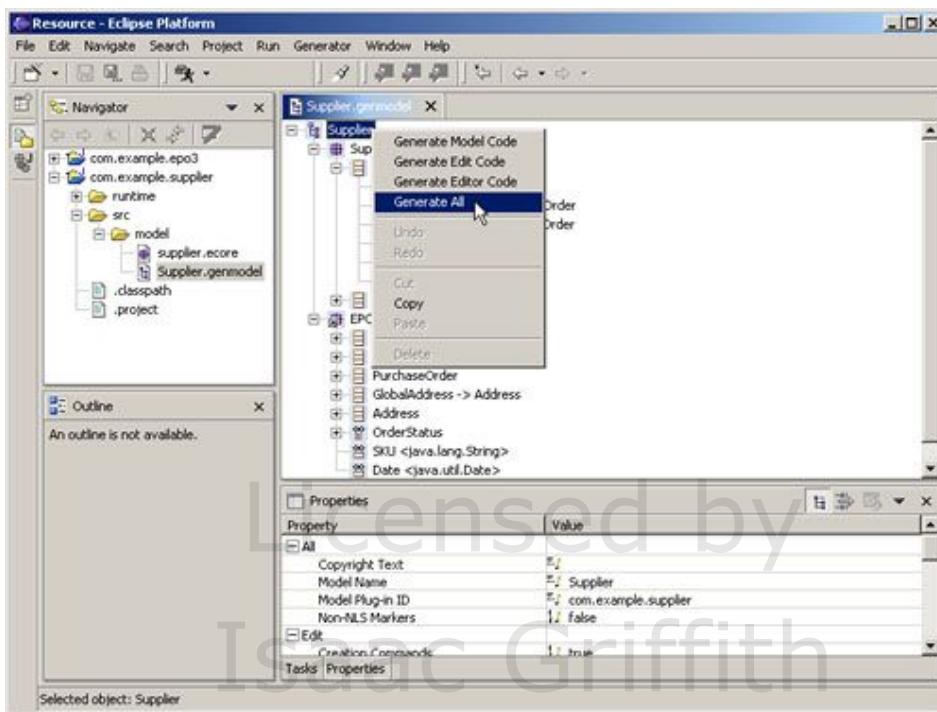


Figure 11.3. Generator pop-up menu.

These actions are also available in the **Generator** pull-down menu. The code and other content generated corresponds to the selected object and all of its children, as specified by Tables 11.1, 11.2, and 11.3. So, for example, when the root-level model object is selected, **Generate All** generates all three plug-ins completely. If, however, a package is selected, only the code that corresponds to that package and its classes and enums is generated. Even if it is the only package in the model, plug-in manifests, property files, and plug-in classes are not generated at the package level.

Some or all of these menu items may be disabled based on the type or state of the selected object. For example, there is never any content to be generated for a feature or a data type. Also, you cannot generate any content at all unless a destination directory is specified in the generator model object, as we will see below. Moreover, nothing can ever be generated for referenced packages.

At this point, one caveat is required: after reloading the model or changing some properties, you may need to regenerate more than you think. For example, after adding an attribute to a class, simply regenerating the class is not sufficient. The metamodel will also be affected by this change. Since the metamodel is initialized and accessed via the package class, the package will need regeneration, as well. In general, it's safest to just regenerate the whole model whenever you make a change.

Generator Model Properties

We now return to the generator model properties. We describe the properties of each type of object in the generator model, explaining their impact on the code to be generated. Each property is displayed under one of six different categories in the property sheet:

- Ecore
- All
- Model
- Edit

- Editor
- Templates & Merge

The first category, “Ecore”, is present for all generator model objects that decorate core model objects—that is, all but the model object. It contains a single read-only property indicating the decorated Ecore object. The second category, “All”, contains editable properties that affect naming and code patterns common across all three plug-ins; the next three categories, “Model”, “Edit”, and “Editor”, contain only those applicable to a single one. The final category, “Templates & Merge”, present only on the model object, contains editable properties that affect the basic operation of the JET and JMerge engines.

Model Properties

The model object has properties that determine the naming and contents of files corresponding to the model generation unit, that apply to all code produced for the generator model, and that offer control over the low-level operation of the code generator:

Copyright Text—All generated Java source files include a static string `copyright` field initialized to this value. If it is empty, as by default, no such field is generated.

Model Name—The names of the generated plug-ins are formed from this value. These names are declared in the manifest files, as the `name` attribute of the `plugin` element.

Model Plug-In ID—Eclipse uses a unique string to identify each of its registered plug-ins. The plug-in declares this string in its manifest file, as the `id` attribute of the `plugin` element. This property determines the IDs to be declared for each generated plug-in. The model plug-in uses this value exactly. If edit and editor plug-ins are generated, “.edit” and “.editor” are appended to form their IDs. By default, this value is set to the project name, as per the PDE convention. To ensure uniqueness, the recommended convention is to begin plug-in IDs, and hence project names, with a reversed domain name.

Non-NLS Markers—Eclipse’s Java compiler has the ability to flag non-externalized strings as a warning or error, in order to facilitate enablement of National Language Support (NLS). EMF-generated code does not use hard coded strings for messages that the user will see; however, string literals do appear frequently, for example, as keys for lookup of externalized strings in a property file. This property controls whether to include comments that mark those literals as non-translatable, so that the compiler will not flag them.

Runtime Jar—Each plug-in declares the libraries that make up its runtime in the `runtime` section of its manifest file. By default, this property’s value is `false`, and the `runtime`/project subdirectory is specified, to make available the class files contained in it. If you plan on packaging these classes in a JAR file, it should be changed to `true`. Then, a JAR file, named for the plug-in ID, is specified.

Generate Schema—EMF can generate an XML Schema that defines the serialized form of the model, if enabled via this property. A schema is generated for each package that provides no specialized resource,¹ hence using the default XMI resource implementation, and for each package that uses a resource derived from either of EMF’s XML or XMI implementations.

Model Directory—This property controls the location, relative to the workspace root, into which model code is generated. Note that the first segment of this path corresponds to the project name, and the remainder of the path specifies the subdirectory under the project’s base at which the source tree is rooted. Of course, further subdirectories will be created according to the package structure of the generated code. Model code cannot be generated while this value is empty.

¹The resource type is a property of the package; we describe its available options in the following subsection.

Model Plug-In Class—This property specifies the fully qualified class name of the singleton plug-in class for the model. As we explained in Section 11.1, you may wish to generate such a class, analogous to those required in the edit and editor plug-ins, to provide resources and logging capabilities for use by any additional model code you write. If so, specify a package and class name here. By default, this value is empty and no plug-in class is generated.

Creation Commands—Support for child and sibling creation is optionally included in the generated code according to this value. Specifically, disabling creation commands suppresses generation of code in the item providers and action bar contributors, as described in Sections 10.1.5 and 10.4, respectively, and of supporting text and image resources.

Edit Directory—This property controls the location, relative to the workspace root, into which edit code is generated. It can be the same as the model directory, in which case the model and its edit support are combined into the same plug-in. Edit code cannot be generated while this property is empty.

Edit Plug-In Class—This property specifies the fully qualified class name of the singleton plug-in class for the model's edit support. Code for the edit plug-in cannot be generated while this value is empty, unless this plug-in is combined with the model and/or editor plug-in, in which case only one plug-in class needs to be specified among them.

Editor Directory—This property controls the location, relative to the workspace root, into which editor code is generated. It can be the same as the edit directory, to include the editor and edit support in the same plug-in. If it is also the same as the model directory, all three pieces are combined in a single plug-in. Editor code cannot be generated while this value is empty.

Editor Plug-In Class—This property specifies the fully qualified class name of the singleton plug-in class for the model's editor. Code for the editor plug-in cannot be generated while this value is empty, unless this plug-in is combined with the edit and, optionally, model plug-ins, in which case only one plug-in class needs to be specified among them.

Dynamic Templates—As discussed in Section 11.1, code generation can be based on dynamic or static templates. EMF ships with dynamic and static forms of its templates, and this property determines which form to use. If you switch to dynamic templates, any changes you make to them are reflected in the generated code.

Force Overwrite—This property specifies whether the generator should attempt to overwrite read-only files. By default, it is set to `false` and any existing, read-only files are left unchanged.

Redirection Pattern—Generator output can be redirected to alternate filenames according to the value of this property. If not empty, it acts as the pattern used to form such filenames, where “{0}” is replaced by the usual filename. So, for example, a value of “{0}.test” will cause the generator to append a `.test` extension to each output filename.

Template Directory—This property specifies the location, relative to the workspace root, in which to look for custom dynamic templates. EMF's templates are found in the `templates`/subdirectory of the `org.eclipse.emf.codegen.ecore` plug-in. To modify them, copy part or all of the contents of this directory, maintaining the subdirectory structure, to another location, and specify it as the value of this property. Modifications to any templates that are found in this directory are picked up by the generator; if any are missing, the ordinary templates are used instead.

Update Classpath—This property specifies whether the generator should, every time code is generated, add entries to each project's class path for its framework and generated EMF dependencies. If `false`, the class path will only be updated when projects are automatically created.

Package Properties

Each package includes one read-only property that indicates the decorated Ecore package, as well as the following editable properties:

Base Package—If this value is not empty, it is used as the package of which all generated packages are made subpackages. This allows you to easily generate code with globally unique package names, without modeling empty, nested packages.

Prefix—The names of the package-related classes are formed from this value, as specified by Table 11.1. Also, the extension for resources containing instances of this model is the lowercased form of this value.

Adapter Factory—This property determines whether to generate the switch and adapter factory classes, described in Section 9.8, as part of the model plug-in. If the edit plug-in is to be generated, this value should be `true`, as its item provider adapter factory extends this adapter factory.

Resource Type—This property specifies whether to create a resource and resource factory implementation that can be customized for the package and, if so, which base classes to extend. The following values are possible:

- **None**—No specialized implementations are created; the default EMF serialization, XMI, is used.
- **Basic**—The implementations extend the most basic framework base classes, `ResourceImpl` and `ResourceFactoryImpl`. These resource implementation are not complete; several methods must be re-implemented or exceptions will be thrown on loading and saving.
- **XML**—The implementations extend `XMLResourceImpl` and `XMLResourceFactoryImpl` to provide XML serialization and deserialization.
- **XMI**—The resource implementation extends `XMIResourceImpl`, the default EMF resource that specializes the XML serialization and deserialization to conform to the XMI specification. The factory implementation extends `ResourceFactoryImpl`.

For all values but the first, the custom resource factory is registered to be used for resources containing model instances, as determined from their filename extension. This is accomplished by declaring, in the plug-in manifest file, an extension to the `org.eclipse.emf.ecore.extension_parser` extension point.

Class Properties

In addition to the read-only property that indicates the decorated Ecore class, each class includes the following editable properties:

Image—This property determines whether to generate an icon file to represent instances of the class.

Label Feature—This property indicates which of the class' single-valued attributes is used in the `getText()` method of its item provider adapter, as described in Section 10.1.1.

Provider Type—This property determines whether to generate an item provider for this class and, if so, which of the patterns described in Section 10.2—Singleton or Stateful—should be used by its adapter factory to create it.

Feature Properties

In addition to the read-only property that indicates the decorated Ecore attribute or reference, each feature includes the following editable properties:

Children—This property specifies whether objects referenced via this feature are to be considered children. As explained in Sections 10.1.1 and 10.1.5, this value affects `getChildrenReferences()` and `collectNewChildDescriptors()` implementations generated in the item provider for the feature's containing class. By default, it is `true` for all containment references and `false` for all other references and attributes.

Notify—This property determines whether notifications for this feature are to be forwarded to the model's central change notifier. This is reflected in the implementation of `notifyChanged()` generated in the item provider for the feature's containing class, as described in Section 10.1.4.

Property Type—This property specifies whether a property descriptor should be contributed for this feature, resulting in an entry in the property sheet, and whether it should be editable or read-only. This affects the implementation of `getPropertyDescriptors()` generated in the item provider for the feature's containing class, as described in Section 10.1.2.

Code Formatter Preferences

In addition to the generator model properties discussed in the previous subsection, the behavior of the EMF generator is controlled by the JDT's Java Code Formatter preferences. Of course, while the generator model properties apply only to a particular model, these preferences apply globally.

Preferences are available from the **Window** pull-down menu. From the preference tree on the left of the dialog, expand “Java” and select “Code Formatter”, as illustrated in Figure 11.4. The EMF generator respects the following three settings, which are located on the **New Lines** and **Style** tabs: **Insert a new line before an opening brace**, **Insert tabs for indentation, not spaces** and, if that is not checked, **Number of spaces representing an indentation level**.

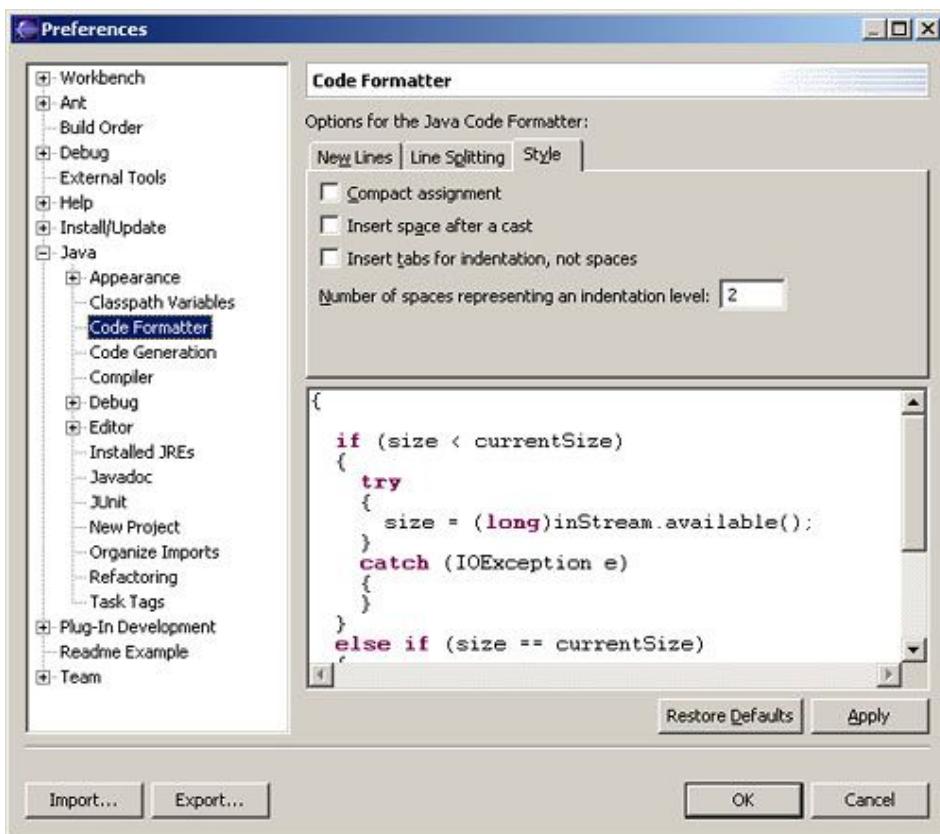


Figure 11.4. Code formatter properties.

The Command-Line Generator Tools

For those who love to script, a command-line interface to the EMF generator is provided as an alternative to the integrated Eclipse UI. There is also a utility for building core and generator models from a Rational Rose class model; the other model sources—annotated Java and XML Schema—are not supported on the command line.

Since the generator has non-interface dependencies, including resource management and XML parsing, we run it under the Eclipse platform, but without launching the workbench interface. We call this a *headless* invocation.

The following subsection gives some background on headless invocation. After that, we'll take a look at the two command-line tools: Rose2GenModel and Generator.

Headless Invocation

The familiar *eclipse* or *eclipse.exe* executable simply starts up a Java Virtual Machine and executes `org.eclipse.core.launcher.Main` from *startup.jar*, a JAR file that it adds to the class path. The launcher initializes the platform and then, by default, starts the workbench UI. We can instruct the launcher to start another Eclipse application instead—Rose2GenModel or Generator—via a command-line option.

One problem with the Eclipse executable is that, depending on the platform, it may block the standard out and standard error streams (`System.out` and `System.err`) from writing to the console. Since we expect a decently behaved command-line utility to output text, we will just directly execute the JVM ourselves.

For our examples, we will always assume that our JVM executable is named *java* and can be found in the path. Further, we expect that *startup.jar*, which is located in the main Eclipse directory, can be found in the class path. Note that this is most easily accomplished by setting the `CLASSPATH` environment variable to the JAR file's full pathname.

With these assumptions in mind, our command line will always start like this:

```
> java org.eclipse.core.launcher.Main -noupdate -data C:\temp\emf-ws  
-application ...
```

For the sake of readability, we will continue to wrap command lines as above.

The “-application” launcher option is followed by the ID for the application to run. Application IDs are defined in plug-in manifest files; each is associated with a class that implements the `IPlatformRunnable` interface.

The “-data” option specifies a location for the workspace. The workspace is not of particular interest to us in this context, since our utilities will map projects to whatever locations are appropriate. However, if we don't specify a location for the workspace, it will automatically be created in the current working directory and metadata will be stored in it. Things break down if that directory is, or is under, one of the directories to which we try to map a project. So, just to be safe, we specify a location under the system-wide temporary directory. We consider this to be transient data, and we must delete it between headless invocations. If we fail to do so, mysterious errors may result when a project that already exists should be mapped to a new location.

The “-noupdate” option should always be used on headless invocations to suppress the runtime configuration step that is performed on Eclipse's first startup. Since we have no intention of retaining our workspace, every startup may be considered its first, so we just skip this process.

The above command line works well for Windows users, but, of course, Linux and UNIX users would modify it slightly:

```
$ java org.eclipse.core.launcher.Main -noupdate -data /tmp/emf-ws
  -application ...
```

In general, both UNIX and Windows path separators (“/” and “\”) are accepted. For the sake of brevity and consistency, we will continue to show command lines in the Windows style, and leave it to UNIX users to make the required simple substitutions mentally.

Rose2GenModel

Rose2GenModel provides a command-line interface for building core and generator models from a Rose class model. Since it serves the same role as the EMF Model Wizard, which we discussed in Chapter 4, we'll demonstrate it using the same example model: the primer purchase order, which is defined in *PrimerPO.mdl*.

In the simplest case, it is invoked as follows:

```
> java org.eclipse.core.launcher.Main -noupdate -data C:\temp\emf-ws
  -application org.eclipse.emf.codegen.ecore.Rose2GenModel
  PrimerPO.mdl
```

Here, we just pass Rose2GenModel the name of the Rose file, which we assume is located in the current working directory. A core model called *ppo.ecore* is created for the single root package in the Rose model. A generator model called *PrimerPO.genmodel* is then created to decorate it. Had there been more than one root package in the Rose model, one core model would have been created for each and the generator model would have, by default, decorated the first. As created, the generator model is suitable for generating the model plug-in only, as no target directories are specified for the edit and editor plug-ins. If you look at the generator model, you will notice that the **GenModel** element contains no **editDirectory** or **editorDirectory** attribute. Also, you might notice that the value of **modelDirectory** is the absolute path to the current working directory, while the values we have seen before were simpler locations relative to the workspace root. This different value is specially encoded to allow the generator to correctly map the target project to its actual target directory. By default, this is just the current working directory, but we'll see how to change that in the next example.

Suppose that we want to re-create the familiar three plug-in pattern for code generation with the CLI tools and, further, that we wish to have those three project directories be subdirectories of some arbitrary directory on our file system. If the Rose model resides in that directory, we simply change to it and issue the following command:

```
> java org.eclipse.core.launcher.Main -noupdate -data C:\temp\emf-ws
  -application org.eclipse.emf.codegen.ecore.Rose2GenModel
  PrimerPO.mdl com.example.ppo\src\model\PrimerPO.genmodel
  -modelProject com.example.ppo src
  -editProject com.example.ppo.edit src
  -editorProject com.example.ppo.editor src
```

Here, we have included the optional argument following the Rose file name that specifies the path-name to use for the created generator model. We have also used the “-modelProject”, “-editProject”, and “-editorProject” options to specify targets for each of the three plug-ins. Each of these options is followed by two additional arguments: first the directory to which the project should map, and then the target directory, relative to that, for generated code.

Rose2GenModel can take several more options, which we have not demonstrated. The most important ones offer control over which core models the generator model should include and reference. The “-package” option, which can be repeated, is followed by the name of an Ecore package to include. This name can then, optionally, be followed by four additional arguments specifying the

namespace prefix, namespace URI, base package name, and prefix for the package. These values are ignored if any of them are specified in the Rose model itself.² Similarly, the “-refPackage” is used to specify those Ecore packages that are to be referenced by the generator model. It can also be used repeatedly and be followed by the same additional arguments.

The “-pathMap” option allows us to specify a mapping for any path map symbols that are used in the Rose file. It is followed by one or more symbol/directory argument pairs. The “-templatePath” and “-copyright” options correspond directly to the copyright text and template directory properties described in Section 11.2.1. Each is followed by a single argument, the value to which the appropriate generator model attribute should be set.

A summary of Rose2Ecore's arguments and options is displayed by invoking it with no arguments at all:

```
> java org.eclipse.core.launcher.Main -noupdate -data C:\temp\emf-ws
    -application org.eclipse.emf.codegen.ecore.Rose2GenModel
```

Generator

Generator provides a command-line interface for generating code from a generator model and one or more core models, which are typically created by Rose2GenModel.

In the previous subsection's first example, we created a basic generator model and core model for the primer purchase order. The following command generates code from it:

```
> java org.eclipse.core.launcher.Main -noupdate -data C:\temp\emf-ws
    -application org.eclipse.emf.codegen.ecore.Generator
    PrimerPO.genmodel
```

In this most basic form, we simply specify the generator model. Because we didn't do anything special to specify a location when we created the model, the project directory is mapped to the same directory, and generated code is not placed in any further subdirectory. Rather, the package corresponding to our outermost package, *com/*, shows up in the current working directory. By default, only the model plug-in is generated.

We can explicitly specify which plug-ins to generate with the “-model”, “-edit”, and “-editor” options. In the second example of the previous subsection, we created a generator model and a core model that would support generation of all three plug-ins and provide the familiar project and directory structure for generated content. Here is the corresponding generator command, to be executed from the same location:

```
> java org.eclipse.core.launcher.Main -noupdate -data C:\temp\emf-ws
    -application org.eclipse.emf.codegen.ecore.Generator
    -model -edit -editor com.example.ppo\src\model\PrimerPO.genmodel
```

The generator provides several more simple option flags, which correspond to generator model attributes described in Section 11.2.1. Dynamic templates are used if “-dynamicTemplates” is specified; recall that the template directory can be set in the Rose2GenModel step. To attempt to force overwriting of existing, read-only files, use “-forceOverwrite”. Alternately, to facilitate comparisons of existing and generated content, specify “-diff” and the generator will use a redirection pattern of “.{0}.new” for the generated files, instead of merging them into the existing versions. To generate an XML Schema along with the model plug-in, use the “-generateSchema” option. To include non-NLS markers in generated code, specify “-nonNLSMarkers”.

²Section 4.2.2 described how to set them in Rose.

Finally, the CLI-based generator can use a generator model that was created by the EMF Model Wizard, instead of one from Rose2GenModel. Recall that such a model has, as its values for the **modelDirectory**, **editDirectory** and **editorDirectory** attributes, workspace-based locations in which the first segment specifies a project name, and the remainder specifies a subdirectory under that project. In this case, we need to provide a final command-line argument: the actual location to which the project should map, since it is not encoded in the attribute itself. Assuming that we are in the parent of that directory and that the generator model is in the *src/model*/subdirectory of it, we would use the following command to generate the model plug-in code from the generator model that we created back in Section 4.2.2:

```
> java org.eclipse.core.launcher.Main -noupdate -data C:\temp\emf-ws
   -application org.eclipse.emf.codegen.ecore.Generator
   com.example.ppo\src\model\PrimerPO.genmodel com.example.ppo
```

Once again, a usage summary can be displayed by invoking the generator with no arguments at all:

```
> java org.eclipse.core.launcher.Main -noupdate -data C:\temp\emf-ws
   -application org.eclipse.emf.codegen.ecore.Generator
```

The Template Format

In this chapter, we have explained that EMF code generation is template-based, and that additions and modifications can be made to the templates and picked up by the generator on the fly. However, in order for this information to be useful, we really need to understand the template format itself.

As previously mentioned, JET templates are based on the powerful JavaServer Pages (JSP) syntax. In fact, the template engine was built using code from Tomcat, the Apache Software Foundation's (www.apache.org) Servlet and JSP implementation.

Template files primarily contain fixed content that is meant to be output exactly as it appears.³ Mixed in with this text are a number of JSP-like tags that are evaluated and interpreted by the template engine in various ways. The different types of tags are summarized in Table 11.4.

Table 11.4. JET Template Tags

Type	Syntax	Description
JET Directive	<%@ jet attributes %>	Declares the start of a template
Include Directive	<%@ include file="URI" %>	Includes another template
Expression	<%= expression %>	Substitutes in expression result
Scriptlet	<% code %>	Executes the code fragment

Based on JSP Reference Card (<http://java.sun.com/products/jsp/syntax/1.2/card12.pdf>)

Notice that these tags constitute only a subset of those available in JSP. In particular, neither declarations nor comments are supported. Let us now look at a simple template that illustrates the typical use of these tags.

Imagine that we want to generate an additional package containing a validator for each Ecore class. For the sake of simplicity, our template will just define an interface that includes a single method to validate each of its structural features. This method takes a single parameter of the correct type for the feature and returns a boolean value indicating whether the feature validated successfully. Our validator interface extends an imaginary `EMFValidator` framework base class that might, for example, declare a generic, reflective `eValidate()` method.

The template, which we call `Validator.javajet`, is defined as follows:

³Actually, that's not entirely true, as the generator may replace tabs by spaces and insert line breaks before braces, according to the preferences described in Section 11.2.2.

```

<%@ jet
   imports="java.util.* org.eclipse.emf.codegen.ecore.genmodel.*"%>
<%@ include file="../Header.javajet"%>
<%GenClass genClass = (GenClass)argument;
   GenPackage genPackage = genClass.getGenPackage();%>

package <%=genPackage.getInterfacePackageName()%>.validation;
import org.eclipse.emf.validation.EMFValidator;

public interface <%=genClass.getInterfaceName()%>Validator
    extends EMFValidator
{
<%for (Iterator i=genClass.getGenFeatures().iterator();
      i.hasNext();)
{
   GenFeature f = (GenFeature)i.next();%>
<%if (f.isChangeable()) {%
   boolean validate<%=f.getCapName()%>(<%=f.getType()%> value);
<%}%>
<%}%>
}

```

We have emboldened the fixed content to make it stand out more clearly from the directives, expressions, and scriptlets.

The template begins with a JET directive, which provides information about the class into which it will be transformed. Here, we are not referring to the class that will eventually be generated, but the intermediate class that will be created and used to emit that content. Imports for this template class are given.

Next, we have an include directive. This causes the engine to process the specified template, inserting the result into its output. In this case, we include a standard header that we intend to be used for copyright information, or anything else desired at the top of each generated file.

Following this is the template's first scriptlet. When the template is executed, it is passed a single special parameter called `argument`. We use as this parameter a generator model object which, throughout the template, provides the details about the specific object for which we are generating. Since we generate one validator interface for each Ecore class, it is a `GenClass` that we are passed. The first scriptlet casts the object appropriately and obtains its `GenPackage` for convenience.

Next, we see some fixed content with details left to be filled in as the results of expressions. For example, in the `package` line, we will insert the name of the model interface package that is obtained from the `GenPackage` object.

Repeated and conditional inclusion of content is provided using normal Java control structures contained in scriptlets. We see a `for` loop that iterates over the class' structural features and an `if` statement that ensures that the feature is changeable.

To understand the effects of these code segments, it is helpful to consider the class into which the template is transformed. The template's fixed content appears in this class as a series of string literals. A single `StringBuffer` is created; the string literals and the results of expressions are appended to it in sequence. The contents of the scriptlets are interspersed among the `append()` calls, as they appear in the template. Thus, you can think of the transformation from template to template class as something like "turning the template inside out." The code contained in the scriptlets is actually used to control the flow through the `append()` statements.

If we were to use this template to generate validator interfaces for our primer purchase order model, the validator for the `PurchaseOrder` class would look like this:

```

/**
 * <copyright>
 * </copyright>
 *
 * %W%

```

```
* @version %I% %H%
*/
package com.example.ppo.validation;
import org.eclipse.emf.validation.EMFValidator;

public interface PurchaseOrderValidator
    extends EMFValidator
{
    boolean validateComment(java.lang.String value);
    boolean validateOrderDate(java.util.Date value);
    boolean validateItems(org.eclipse.emf.common.util.EList value);
    boolean validateBillTo(com.example.ppo.USAddress value);
    boolean validateShipTo(com.example.ppo.USAddress value);
}
```

Notice that this code would be a little more readable if we could drop the object qualification from each of the parameter types, and insert the necessary `import` statements instead. Fortunately, the `GenModel` implementation includes an import manager mechanism that can help us to do this. You can see it in use throughout the templates that ship with EMF.

After this discussion, you're probably wondering how we would go about hooking up a new template to the EMF generator. Unfortunately, right now, we wouldn't. At the time of this book's writing, the generator is not easily extensible. In fact, we would actually have to change and recompile the generator model code in order to accomplish this. Clearly, a mechanism for dynamic extensibility is a desirable thing, and there are plans to add one in the future. In the meantime, you still have the ability to modify or replace any of the existing templates and have your changes picked up by the generator, as explained in Section 11.2.1.

1942365

Chapter 12. Example—Implementing a Model and Editor

Now that you are familiar with the capabilities of the EMF generator, it's time to look at a more realistic example. In this chapter we will generate the ExtendedPO2 model, which includes some common real-world features that require special treatment when generating with EMF. They include:

1. References that do not require proxy resolution
2. Volatile attributes and references
3. Custom data types
4. References with a restricted set of valid targets

We will show how these features can be implemented using a combination of generator control options and custom implementation in the generated classes. We'll then consider an alternative design, ExtendedPO3, which uses multiple packages to implement the same model. We'll look at some of the pros and cons of splitting the model into multiple packages and consider the issues it brings to the table. Finally, we'll show how the ExtendedPO3 editor can be enhanced to concurrently edit multiple instance documents with cross-document references between them.

Getting Started

The ExtendedPO2 model is shown in Figure 12.1. We've already looked at some of the features of this model in previous chapters, but we will now consider all of the issues that need to be addressed when implementing it.

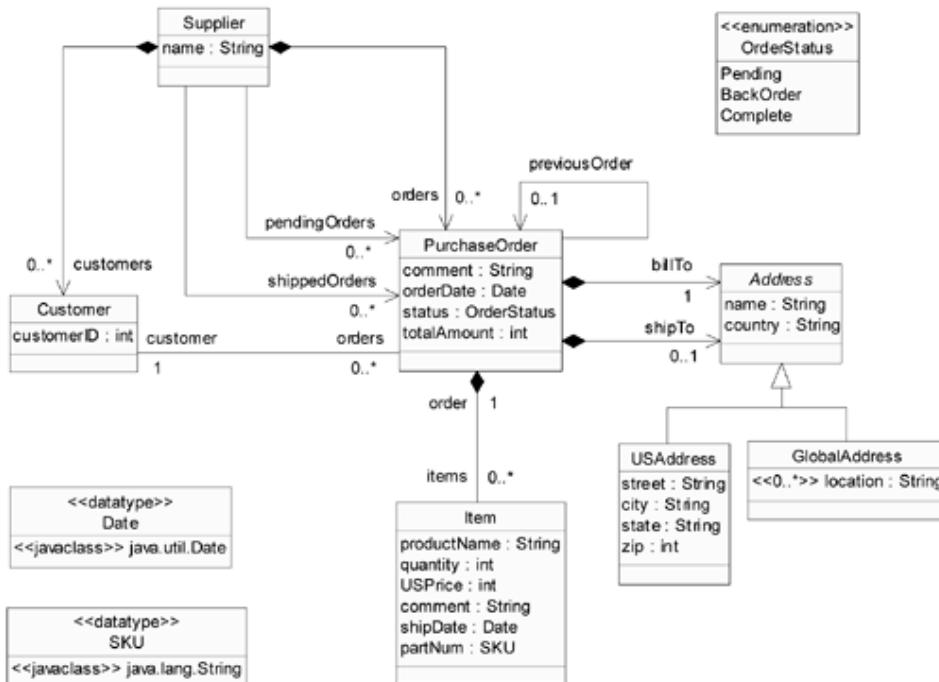


Figure 12.1. The ExtendedPO2 model.

As you can see, the ExtendedPO2 model is a superset of the PrimerPO model, which we introduced in Chapter 4. Like PrimerPO, this model includes a **PurchaseOrder** class that aggregates items and addresses, but in this model the addresses need not be in the United States (that is, instances of class **USAddress**). An abstract base class, **Address**, is used as the target of the **shipTo** and **billTo** references. Class **USAddress** extends from **Address** and contains the typical U.S. address components (street, city, state, and zip code). Another concrete class, **GlobalAddress**, is used to represent addresses anywhere else in the world. Notice how we use a single multi-valued **String** attribute (**location**) to represent the address components in this case.

Class **PurchaseOrder** has two new features that weren't in the PrimerPO model: **previousOrder** and **totalAmount**. The **previousOrder** reference will be used to identify a previous purchase order associated with the same customer, if there is one. The **totalAmount** attribute is, as you'd expect, the total amount of the order, that is, the sum of the individual item amounts.

The ExtendedPO2 model includes two more classes: **Supplier** and **Customer**. Class **Supplier** acts as the root container for both purchase orders and customers. Notice that in addition to the containment (black diamond) association from class **Supplier** to class **PurchaseOrder** (that is, **orders**), there are two more references, **pendingOrders** and **shippedOrders**, between them. These two references will be used to locate the subsets of **orders** whose **status** attributes are "Pending" and "Complete", respectively.

Because **pendingOrders** and **shippedOrders** are simple-to-compute subsets of the "orders" feature, we've defined them to be transient, volatile, and non-changeable. This is not obvious from the UML diagram in Figure 12.1 because, as discussed in Chapter 8, these are extended non-UML properties of the model that are specified separately. In the corresponding core model they look like this:

```

<eClassifiers xsi:type="ecore:EClass" name="Supplier">
  ...
  <eReferences name="pendingOrders" eType="#//PurchaseOrder"
    changeable="false" volatile="true" transient="true"
    upperBound="-1" resolveProxies="false"/>
  <eReferences name="shippedOrders" eType="#//PurchaseOrder"
    changeable="false" volatile="true" transient="true"
    upperBound="-1" resolveProxies="false"/>

```

```

changeable="false" volatile="true" transient="true"
upperBound="-1" resolveProxies="false"/>
...
</eClassifiers>

```

In addition to these two references, there is one other volatile feature in this model. The **totalAmount** attribute, in class **PurchaseOrder**, can be computed from the individual item amounts and therefore we have declared it to be volatile, transient, and non-changeable, and will also compute it on-the-fly.

Class **Customer** is used to represent a supplier's customers. Each purchase order of a supplier is associated with exactly one customer. A customer, however, can have many purchase orders, as indicated by the bidirectional one-to-many association between class **Customer** and class **PurchaseOrder**. Because a single supplier is intended to aggregate purchase orders and their corresponding customers, we know that this association will never span documents, and therefore we have optimized both ends of the association to not resolve proxies.

```

<eClassifiers xsi:type="ecore:EClass" name="PurchaseOrder">
  <eReferences name="customer" eType="#//Customer" lowerBound="1"
    resolveProxies="false" eOpposite="#//Customer/orders"/>
  ...
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Customer">
  <eReferences name="orders" eType="#//PurchaseOrder"
    upperBound="-1" resolveProxies="false"
    eOpposite="#//PurchaseOrder/customer"/>
  ...
</eClassifiers>

```

For the same reason, we have also set **resolveProxies** to **false** for the **previousOrder** reference of class **PurchaseOrder** and for the **pendingOrders** and **shippedOrders** references of class **Supplier**.

Generating the Model

Now that we have our model, the next step is to provide it to the EMF generator using one of the techniques described in Section 4.2. Because the ExtendedPO2 model is a single package model, just like PrimerPO in Chapter 4, there's nothing new to be said about how to do that. You simply use the **New Project** Wizard to create the project and the core and generator models, and then generate the code as described in Chapter 4. You can also experiment with some of the options described in Chapter 11 if, for example, you want to generate the code into a single plug-in instead of the usual three.

After generating this model and editor, you will not be able to simply run it, without further work. If you try, you'll notice that the editor will not function properly. The problem is that, unlike the PrimerPO model from Chapter 4, the ExtendedPO2 model, as generated, is not completely implemented.

Implementing Volatile Features

Before we can run our generated ExtendedPO2 model editor, we first need to implement its volatile features: the **totalAmount** attribute in class **PurchaseOrder**, and the **pendingOrders** and **shippedOrders** references in class **Supplier**. Otherwise the model will throw exceptions whenever they are accessed. In class **PurchaseOrderImpl**, for example, the **getTotalAmount()** method has the following generated implementation:

```

/**
 * @generated
 */
public int getTotalAmount() {

```

```
// TODO: implement this method to return the 'Total Amount'
// attribute
throw new UnsupportedOperationException();
}
```

The exception-throwing implementation was generated as a placeholder for the implementation that, by declaring the feature volatile, we have agreed to provide by hand.

Before we change the generated implementation, we first need to remove the @generated tag from the method comment. Instead of actually deleting it, we will simply add NOT after the @generated tag. This works because the generator considers any change from the simple tag that it generated (alone on the line), to indicate removal of the tag. This approach makes it easy for us to keep track of the generated methods that we've taken over and modified.

Here's what our implementation looks like:

```
/**
 * @generated NOT
 */
public int getTotalAmount() {
    // TODO: implement this method to return the 'Total Amount'
    // attribute
    throw new UnsupportedOperationException();
    int totalAmount = 0;
    for (Iterator iter = getItems().iterator(); iter.hasNext(); ) {
        Item item = (Item)iter.next();
        totalAmount += item.getUSPrice() * item.getQuantity();
    }
    return totalAmount;
}
```

As you can see, we simply iterate through the purchase order items, and sum up the products of each individual item's price multiplied by its quantity.

Our other two volatile features, **pendingOrders** and **shippedOrders**, have similar implementations; they are both filtered subsets of the **orders** feature. We implement the `getPendingOrders()` method in class `SupplierImpl` by simply iterating through the **orders** reference, collecting orders that have a "Pending" value for the **status** attribute as shown:

```
/**
 * @generated NOT
 */
public EList getPendingOrders() {
    ArrayList pendingOrders = new ArrayList();
    for (Iterator iter = getOrders().iterator(); iter.hasNext(); ) {
        PurchaseOrder order = (PurchaseOrder)iter.next();
        if (order.getStatus() == OrderStatus.PENDING_LITERAL)
            pendingOrders.add(order);
    }
    return new EcoreEList.UnmodifiableEList(this,
        EPO2Package.eINSTANCE.getSupplier_PendingOrders(),
        pendingOrders.size(), pendingOrders.toArray());
}
```

Notice how we use the EMF collection class `EcoreEList.UnmodifiableEList` to return the pending orders. Although we use a simple `java.util.ArrayList` to collect the required subset of features, we cannot return this list directly. Like all multiplicity-many feature implementations in EMF, `getPendingOrders()` needs to return an `EList`, not just a `java.util.List`. You may recall from Chapter 9 that the EMF interface `EList` is essentially the same as `java.util.List` (it extends from it), only with the addition of the `move()` API, which all multi-valued EMF features support.¹

Another reason for using `EcoreEList.UnmodifiableEList` here is that the `pendingOrders` feature is declared in the model as “not changeable”. By using `EcoreEList.UnmodifiableEList`, any client attempt at modifying the returned collection will cause an exception to be thrown.

We implement the `getShippedOrders()` method in class `SupplierImpl` similarly, only there we look for orders where the `status` is “Complete”, instead of “Pending”:

```
public EList getShippedOrders() {
    ArrayList shippedOrders = new ArrayList();
    for (Iterator iter = getOrders().iterator(); iter.hasNext(); ) {
        PurchaseOrder order = (PurchaseOrder)iter.next();
        if (order.getStatus() == OrderStatus.COMPLETE_LITERAL)
            shippedOrders.add(order);
    }
    return new EcoreEList.UnmodifiableEList(this,
        EPO2Package.eINSTANCE.getSupplier_ShippedOrders(),
        shippedOrders.size(), shippedOrders.toArray());
}
```

Implementing Data Types

Now that we’ve implemented the volatile features, we can turn our attention to our custom data types: `Date` and `SKU`. In Chapter 9, we explained how each custom data type in a model produces a pair of methods in the generated factory implementation class that are responsible for converting the data type to and from `String`. The default implementation of the two conversion methods for our `SKU` data type look like this in class `EPO2FactoryImpl`:

```
/** 
 * @generated
 */
public String createSKUFromString(EDataType eDataType,
        String initialValue) {
    return (String)super.createFromString(eDataType, initialValue);
}
/** 
 * @generated
 */
public String convertSKUToString(EDataType eDataType,
        Object instanceValue) {
    return super.convertToString(eDataType, instanceValue);
}
```

Like the method generated for any other data type, the implementation of `createSKUFromString()` simply calls `super.createFromString()`, which uses Java reflection to try to create an instance of the data type from its `String` form (`initialValue`). The generated `convertSKUToString()` method also calls `super`, which simply calls `toString()` on the data type instance (`instanceValue`) to produce a `String` version of the data type.

Notice that the `createSKUFromString()` method’s return type is also `String`. If you refer back to Chapter 9, you’ll see that it is supposed to return an instance of the data type’s class, which is exactly what it is doing. Recall from Section 12.1 that the instance class of the `SKU` data type is in fact `String`. Unlike most data types, the only purpose of the `SKU` data type is to allow us to restrict its range of values. Had we not needed that, we would otherwise have simply used the EMF built-in data type, `EString`.

Here is our modified implementation of the `createSKUFromString()` method:

¹In addition to being an `Elist` implementation, `EcoreEList.UnmodifiableEList` also implements the `InternalEList` and `EStructuralFeature.Setting.Internal` interfaces, which are also required of multi-valued feature implementations.

```

/**
 * @generated NOT
 */
public String createSKUFromString(EDataType eDataType,
        String initialValue) {
    return (String)super.createFromString(eDataType, initialValue);
    if (initialValue.length() == 6 &&
        initialValue.charAt(0) >= '0' &&
        initialValue.charAt(0) <= '9' &&
        initialValue.charAt(1) >= '0' &&
        initialValue.charAt(1) <= '9' &&
        initialValue.charAt(2) >= '0' &&
        initialValue.charAt(2) <= '9' &&
        initialValue.charAt(3) == '-' &&
        initialValue.charAt(4) >= 'A' &&
        initialValue.charAt(4) <= 'Z' &&
        initialValue.charAt(5) >= 'A' &&
        initialValue.charAt(5) <= 'Z')
        return initialValue;
    else
        return null;
}

```

We simply check that the supplied string (`initialValue`) is of the right format (three digits followed by a dash and two uppercase letters) and if so, return it. If an illegal value is supplied, we throw it away and return `null` instead. Assuming that **SKU** instances are only ever created using the factory, this guarantees that we will never set an attribute of type **SKU** to an invalid value.²

Notice that because the Java type of a **SKU** attribute is simply `java.lang.String`, this design is not guaranteed safe. Clients can still set a **SKU** attribute directly to an invalid `String` value that they create using the new operator, instead of with the factory. If you want to prevent this, you have two options:

1. You could define your own **SKU** Java class (that wraps a `java.lang.String`, for example), and make its constructor protected so that it can only be created using the factory. If you then set the instance class of the **SKU** data type to this class, instead of simply `java.lang.String`, clients will be unable to set the value of a **SKU** attribute without first validating it.
2. You could override the `set()` method of all **SKU**-typed attributes and put the validation code right there. This approach is less desirable if you plan to reuse the **SKU** type for lots of attributes.

For our example model, we will simply assume that all the model clients are “good citizens” that always create **SKUs** using the factory.³

Since we've validated **SKU** values during creation, there isn't strictly a need to override the `convertSKUToString()` method; we can only ever have valid **SKUs** to convert. However, we will override it anyway for a simple reason: performance. Since we know the data type is a `java.lang.String`, we can simply return the value directly, instead of by calling super to do it indirectly. Here's our modified version:

```

/**
 * @generated NOT
 */
public String convertSKUToString(EDataType eDataType,
        Object instanceValue) {
    return super.convertToString(eDataType, instanceValue);
    return (String)instanceValue;
}

```

²The `createFromString()` method is used to validate values entered in the property sheet. A `null` value returned from `createFromString()` will stop it from accepting the entered value.

³This is certainly the case for all code generated by the EMF generator.

Let's move on to the other data type in our model: **Date**. We chose to use the `java.util.Date` class to represent dates, but we still need to decide on the string format we want to use for our dates. The default format produced by the `toString()` method in class `java.util.Date` is a little more detailed than we need (that is, it produces a string value something like this: "Tue April 01 00:00:00 EST 2003"). Consequently, we will override both the `convertDateToString()` and `createDateToString()` methods to use a simpler string format. Since we only need the date, and not the exact time, we will use a string format that looks like this: "2003.04.01".

Here is our modified `convertDateToString()` method in class `EPO2FactoryImpl`:

```
/**
 * @generated NOT
 */
public String convertDateToString(EDataType eDataType,
                                  Object instanceValue) {
    return super.convertToString(eDataType, instanceValue);
    if (instanceValue == null) return null;
    SimpleDateFormat formatter = new SimpleDateFormat("yyyy.MM.dd");
    return formatter.format((Date)instanceValue);
}
```

As you can see, we use the convenient Java date formatting class, `SimpleDateFormat` (from the `java.text` package), to do the work for us. We also use the same convenient class to implement our `createDateFromString()` method like this:

```
/**
 * @generated NOT
 */
public Date createDateFromString(EDataType eDataType,
                                 String initialValue) {
    return (Date)super.createFromString(eDataType, initialValue);
    if (initialValue == null) return null;
    SimpleDateFormat formatter = new SimpleDateFormat("yyyy.MM.dd");
    return formatter.parse(initialValue, new ParsePosition(0));
}
```

Running the ExtendedPO2 Editor

Now that we've implemented the volatile features and data types, we can run the generated ExtendedPO2 editor to edit instances of the model. You can create an instance document using the EPO2 Model wizard, in the same way as described in Section 4.4 for the PPO model. If you create an instance document choosing an instance of class **Supplier** as the root object, you can then use the EPO2 editor to populate a complete model as shown in Figure 12.2.

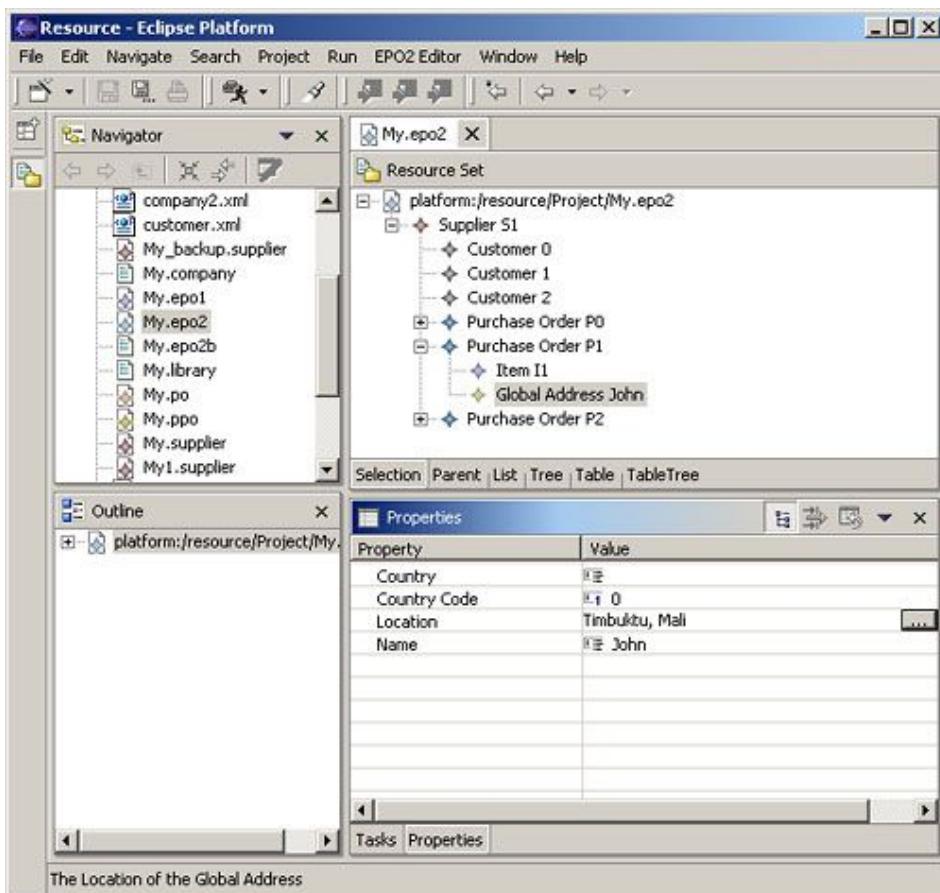


Figure 12.2. Running the EPO2 editor.

While creating and initializing the model, notice how the volatile features **pendingOrders**, **shippedOrders**, and **totalAmount** are automatically updated as expected when you change the other features that affect them. Notice also how the data types, **SKU** and **Date**, affect the display and editing of the **partNum** and **shipDate** attributes in the property sheet.

If you look at the properties of a multi-valued attribute or reference, like **pendingOrders** of class **Supplier**, you'll see how they are displayed in the property sheet as a comma-separated list of values. When a global address is selected, notice that the property value cell for the editable multi-valued **location** attribute has a button on its far right side. Pressing the button will launch the dialog shown in Figure 12.3. As you can see, you can use this dialog to add, remove, and reorder "Location" strings.

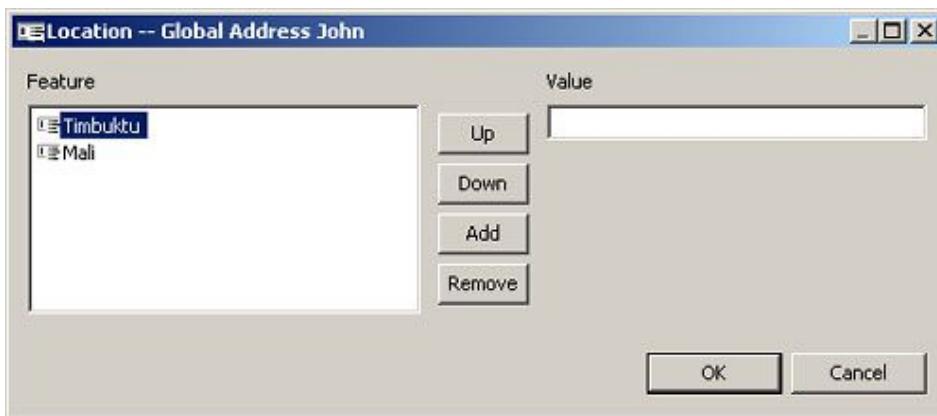


Figure 12.3. Editing the multi-valued “Location” property.

Restricting Reference Targets

Recall that class **PurchaseOrder** includes a **previousOrder** reference, which is intended to be used to identify a previous purchase order associated with the same customer, if there is one (Figure 12.4):

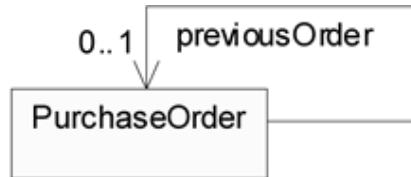


Figure 12.4. *PurchaseOrder* and its *previousOrder* reference.

We can set the previous order of a given order (for example, P1) using the property sheet in the generated editor, as shown in Figure 12.5. Notice that the drop-down combo-box includes every purchase order in the model, even P1 itself. This is because the default implementation in the framework simply populates the combo-box with all objects in the resource of a compatible type (that is, instances of class **PurchaseOrder**, including subclasses if there are any).

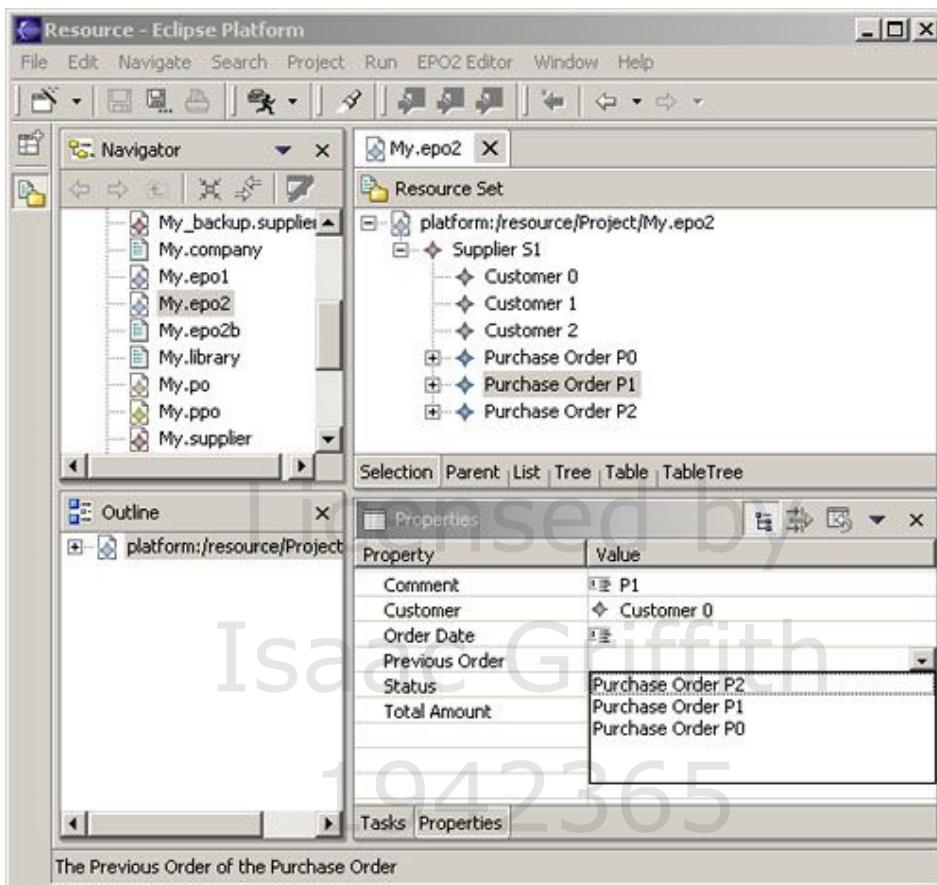


Figure 12.5. Default previous order combo-box items.

Because the previous order is supposed to reference a different order, and one associated with the same customer, we should ideally filter the orders in the combo-box to give the user a list that only includes valid targets to choose from. We can easily do this by overriding the `getComboBoxObjects()` method (in class `ItemPropertyDescriptor`) for the **previousOrder** property descriptor. The easiest way to do it is by creating an anonymous `ItemPropertyDescriptor` subclass in the `addPreviousOrderPropertyDescriptor()` method in class `PurchaseOrderItemProvider`, like this:

```
/**
 * @generated NOT
 */
protected void addPreviousOrderPropertyDescriptor(Object object) {
    itemPropertyDescriptors.add(
        new ItemPropertyDescriptor(
            (ComposeableAdapterFactory)adapterFactory,
            getRootAdapterFactory(),
            getString("_UI_PurchaseOrder_previousOrder_feature"),
            getString("_UI_PropertyDescriptor_description",
                "_UI_PurchaseOrder_previousOrder_feature",
                "_UI_PurchaseOrder_type"),
            EPO2Package.eINSTANCE.getPurchaseOrder_PreviousOrder(),
            true)
    {
        protected Collection getComboBoxObjects(Object object) {
            PurchaseOrder order = (PurchaseOrder)object;
            Customer customer = order.getCustomer();
            ArrayList result = new ArrayList();
            if (customer != null) {
                for (PurchaseOrder o : customer.getPurchaseOrders()) {
                    if (o != order) {
                        result.add(o);
                    }
                }
            }
            return result;
        }
    });
}
```

```
if (customer != null) {
    result.addAll(customer.getOrders());
    result.remove(order);
}
return result;
});
```

As you can see, we've implemented the `getComboBoxObjects()` method by navigating to the selected order's customer and then returning all the customer's purchase orders, excluding the selected one.⁴ If we run the editor again, we'll see that the list of available targets for the previous order is now filtered appropriately. For example, assuming that purchase orders P0 and P1 in Figure 12.5 are associated with customer 0, but P2 is not, then the combo-box will now only include P0, as shown in Figure 12.6. As expected, the invalid targets, P1 and P3, are no longer among the available choices.

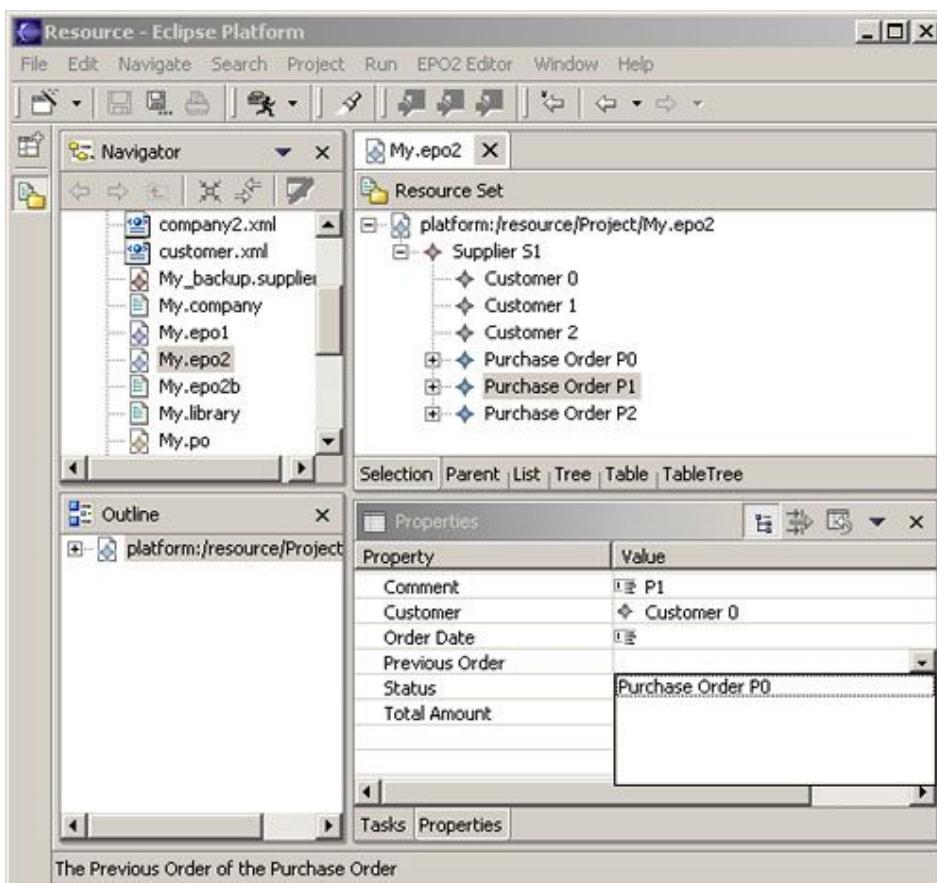


Figure 12.6. Filtered previous order combo box items.

⁴If we wanted to get fancier yet, we could take `orderDate` into account as well.

Splitting the Model into Multiple Packages

As models get larger, it's often desirable to split them into multiple packages. For example, the ExtendedPO3 model is another variation of the purchase order model, similar to ExtendedPO2, only with the classes split into two packages: **epo3** and **supplier**. The **epo3** package includes class **PurchaseOrder** and its component classes (**Item** and the three address classes) as shown in Figure 12.7.

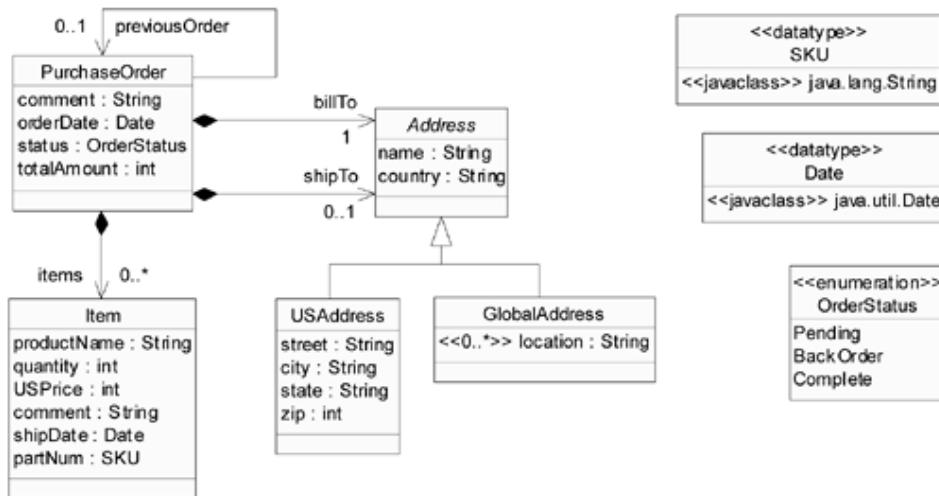
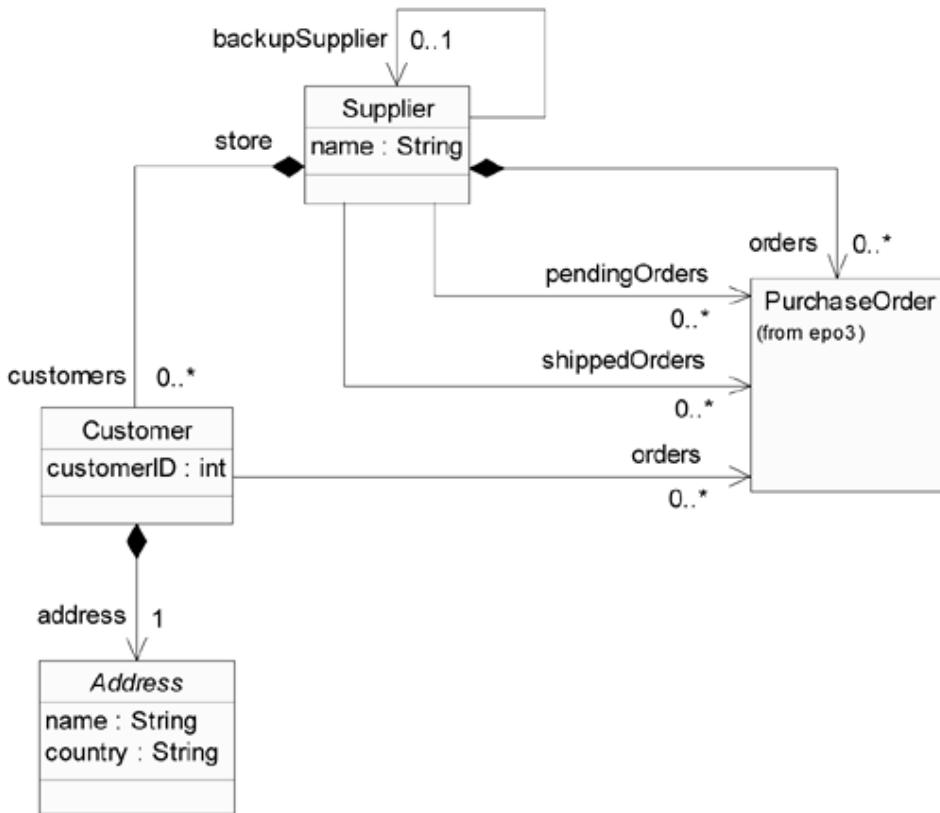


Figure 12.7. The ExtendedPO3 **epo3** package.

The remaining classes from ExtendedPO2's one and only package (**epo2**) are, in ExtendedPO3, moved to a separate package (**supplier**) as shown in Figure 12.8. If you look closely at the association between class **Customer** and class **PurchaseOrder**, you'll notice that we've changed it to be one-way instead of bidirectional as it was in ExtendedPO2. This illustrates a subtle issue that arises when separating a model into multiple packages: cyclical dependencies. When we separated the model into two packages, we had the option of either layering the package dependencies as we did (here, we made the **epo3** package have no knowledge of its use by the **supplier** package), or to simply separate them for organizational purposes but leave them codependent.

Figure 12.8. The *ExtendedPO3 supplier* package.

By changing the model to make all the associations between the packages one-way from **supplier** to **epo3**, we give ourselves the option of generating the **epo3** package alone in a project and then later generating the **supplier** package as an extension of **epo3**. If we had kept the two-way dependencies between the packages, we would have no choice but to generate them together in a single project. This is an extremely important consideration when designing large models with more than one package.

Resolving Package Dependencies

Because the **epo3** package is independent, let's assume that we decided to create an EMF project for just the **epo3** package using one of the techniques described in Chapter 4. Now, if we want to extend it with a project containing only the **supplier** package, we need to resolve the cross-package dependency first.

If we're using the annotated Java approach to define the **supplier** package, then we will need to make the **epo3** project a required one when creating the **supplier** project with the Java Project Wizard, as shown in Figure 12.9. If we don't do this, then importing the model will fail because of the unresolved references to the classes in **epo3**.

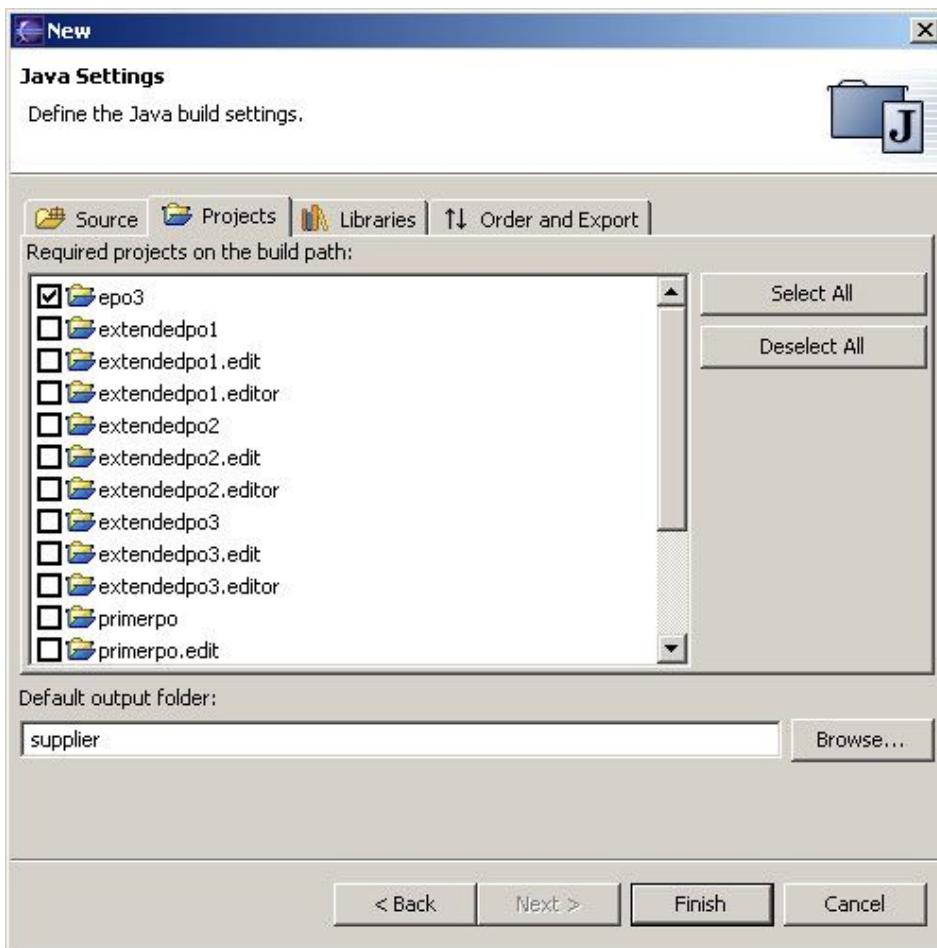


Figure 12.9. Specifying required projects in the Java Project Wizard.

Once the project dependency is set, we can then simply proceed as described in Section 4.2.1.

If instead we want to create the **supplier** package from Rational Rose, XML Schema, or from Ecore directly, then the EMF Project Wizard, instead of the Java Project Wizard, will be used to create the project for it. In this case, the dependency is resolved by locating the referenced model as shown in Figure 12.10. Notice that because we have elected to generate (that is, we have selected) the **supplier** package alone, the **Finish** button is disabled because of the unresolved references to the **epo3** package.

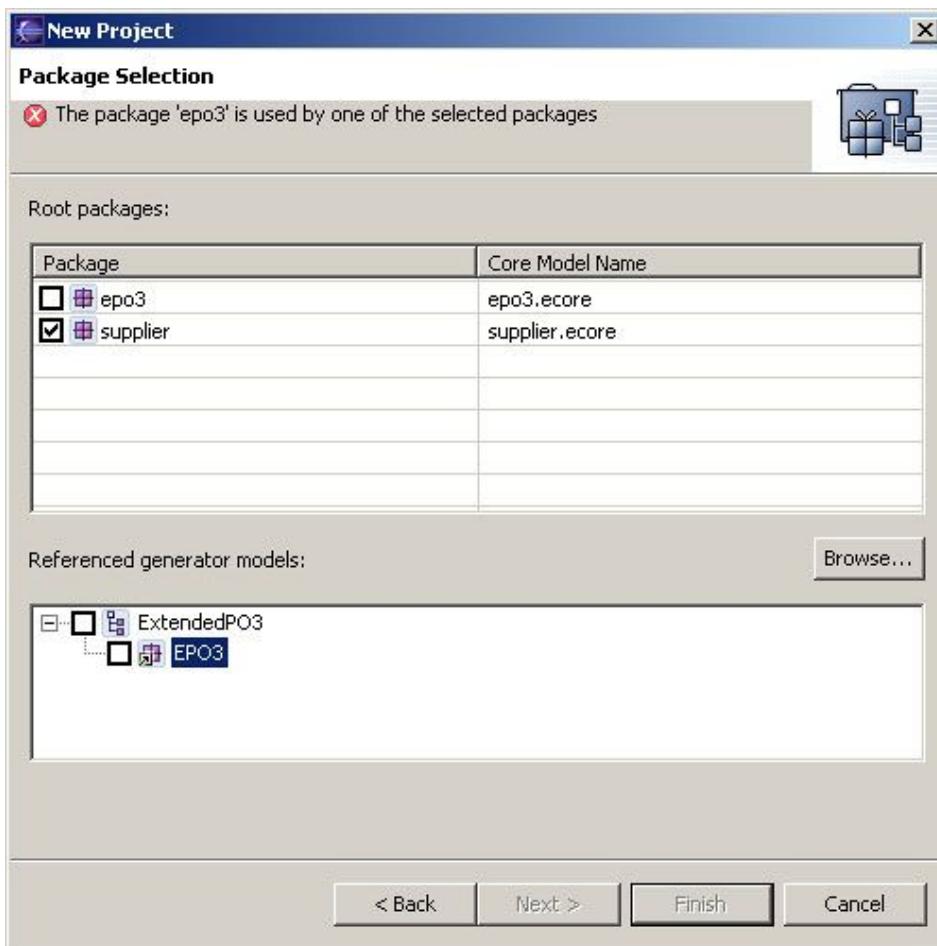


Figure 12.10. Locating referenced models using the EMF Project Wizard.

To proceed, we click the **Browse...** button to display a file selection dialog, from which we select the existing generator model containing the package that we wish to reference. Once we select that package, “EPO3”, from the generator model tree in the bottom part of the wizard dialog, the error message will disappear and the **Finish** button will be enabled. We can then proceed to generate the model in the usual way.

Restricting Reference Targets Revisited

As we described above, in order to implement the two ExtendedPO3 packages in separate projects we needed to change the model to remove the bidirectional association between class **Customer** and class **PurchaseOrder**. You may have noticed that this small design change will break the implementation we provided in Section 12.6 for filtering the choices in the “Previous Order” combo-box. We were using the `getCustomer()` method to navigate to the selected order’s customer, which we then used to locate the valid orders. Now that we’ve made the association one-way, from **Customer** to **PurchaseOrder**, we don’t have the `getCustomers()` method on interface **PurchaseOrder** any more. Without it, we’ll need to find another way of locating the customer.

The EMF framework includes a number of convenient utility classes, one of which can be used to help with this problem. The class `EcoreUtil.UsageCrossReferencer` can be used to find incoming references, or usages, of an object. We will use it to find the customer whose `order` feature references the selected purchase order, after which we can proceed to populate the purchase order list as in the previous example. Here's our new version of the `getComboBoxObjects()` method.

```


    /**
     * @generated NOT
     */
    protected void addPreviousOrderPropertyDescriptor(Object object) {
        itemPropertyDescriptors.add(
            new ItemPropertyDescriptor(
                ((ComposeableAdapterFactory)adapterFactory).
                    getRootAdapterFactory(),
                getString("_UI_PurchaseOrder_previousOrder_feature"),
                getString("_UI_PropertyDescriptor_description",
                    "_UI_PurchaseOrder_previousOrder_feature",
                    "_UI_PurchaseOrder_type"),
                EPO3Package.eINSTANCE.getPurchaseOrder_PreviousOrder(),
                true)
        {
            protected Collection getComboBoxObjects(Object object) {
                final PurchaseOrder order = (PurchaseOrder)object;
                ArrayList result = new ArrayList();
                Collection references = EcoreUtil.UsageCrossReferencer.find(
                    order, order.eResource().getResourceSet());
                for (Iterator iter = references.iterator(); iter.hasNext();) {
                    EStructuralFeature.Setting setting =
                        (EStructuralFeature.Setting)iter.next();
                    if (setting.getEStructuralFeature() == 
                        SupplierPackage.eINSTANCE.getCustomer_Orders()) {
                        Customer customer = (Customer)setting.getEObject();
                        result.addAll(customer.getOrders());
                        result.remove(order);
                        break;
                    }
                }
                return result;
            }
        });
    }
}


```

As you can see, we first call the static `find()` method on class `UsageCrossReferencer` to locate the incoming references to the selected order. Notice how we pass two arguments to the cross referencer, `order` and `order.eResource().getResourceSet()`. The first argument is the object of which we want to find usages, which in our example is the selected purchase order. The second argument is the scope in which to search.

The references returned by the cross referencer are in the form of a collection of settings (that is class `EStructuralFeature.Setting`).⁵ So, to find our order's customer we proceed to iterate through the list of settings, searching for the first one with the right feature, `SupplierPackage.eINSTANCE.getCustomer_Orders()`.⁶ Once we find the right setting, we simply retrieve the referencing customer by calling the `getEObject()` method on the setting, and then continue with the implementation as before.

There are actually four different variants of the `UsageCrossReferencer.find()` method:

⁵Settings are used in several places in EMF, mostly for the dynamic `EObject` implementation. A setting wraps a feature, a value of the feature, and the object to which the value belongs.

⁶Notice how we're assuming that there will only be one customer referencing a given purchase order, even though this is no longer enforced by the model, as it was in ExtendedPO2. If we want to guarantee this, we would need to add some filtering for valid targets of the `orders` feature of class `Customer`, to exclude purchase orders that are already associated with a customer.

```

public static Collection find(EOBJECT eObjectOfInterest,
    EOObject eObject)

public static Collection find(EOBJECT eObjectOfInterest,
    Resource resource)

public static Collection find(EOBJECT eObjectOfInterest,
    ResourceSet resourceSet)

public static Collection find(EOBJECT eObjectOfInterest,
    Collection emfObjectsToSearch)

```

The first version allows you to search for referencing objects within the containment tree rooted by the specified `eObject`. The second and third allow you to widen the search to the contents of a resource or an entire resource set, respectively. The final version lets you provide a heterogeneous list of context roots (`EOBJECTs`, `Resources`, or `ResourceSets`), all of which will be searched.

You may be wondering why we used the resource set version in our example. If we know that the previous order is always limited to other orders owned by the same supplier, then we could have narrowed the search to the current order's supplier, like this:

```
Collection references = EcoreUtil.UsageCrossReferencer.find(
    order, order.eContainer());
```

This would run faster but would not allow us to find customers from a supplier that may be different from the one managing the order. We designed it with the wide search in anticipation of the changes we're going to make in the following sections, but for now suffice to say that either approach would have worked equally well.

Editing Multiple Resources Concurrently

Another design change we made in the ExtendedPO3 model, which you may have noticed already, was to add another association on class **Supplier** (Figure 12.11):

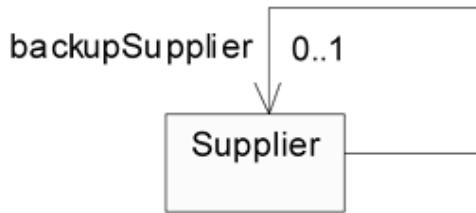


Figure 12.11. Adding an association on **Supplier**.

We added the **backupSupplier** reference to the model to represent another supplier that a given supplier may use to service orders for its customers. We made the `resolveProxies` property `true` for this reference so that we can serialize a supplier and its backup into different resources if we want to. Because of this design change, we need to reconsider our decision from Section 12.1 to declare several other references in the model as non-proxy resolving. Specifically, now that a customer's order may be filled by a backup supplier in a different resource, the `orders` reference from class **Customer** to **PurchaseOrder** and the `previousOrder` reference of class **PurchaseOrder** must now also be made to resolve proxies. We can no longer optimize these references.

With these changes we now have a version of the purchase order model that supports multiple cooperating suppliers in potentially more than one resource. But now that we've enabled the model for this, how do we make use of it? Our generated editor only edits a single resource containing a single root object, right? Well, not exactly.

One thing that you've probably noticed by now is that the viewers in a generated EMF editor always display the resource at the top, instead of the root object(s) of the resource. For example, back in Figure 12.5, the root of the tree viewer is the resource `platform://resource/Project/My.epo2` instead of the supplier S1 itself. Although this default behavior can be easily changed, and we will show how in Chapter 14, it does give us the ability to use the default generated editor to work with resources that have cross-document references.

For example, assume we have a resource, `My.supplier`, which contains a supplier instance with a cross-document reference, `backupSupplier`, to a supplier in another resource, something like this:

```
<com.example.supplier:Supplier xmi:version="2.0" ... name="S0">
  <backupSupplier href="My_backup.supplier#/"/>
</com.example.supplier:Supplier>
```

If we open this resource with the generated Supplier editor, the tree viewer will (as usual) display the supplier S0 under its resource, as shown in Figure 12.12. Notice that only the single resource, `My.supplier`, is displayed in the viewer.

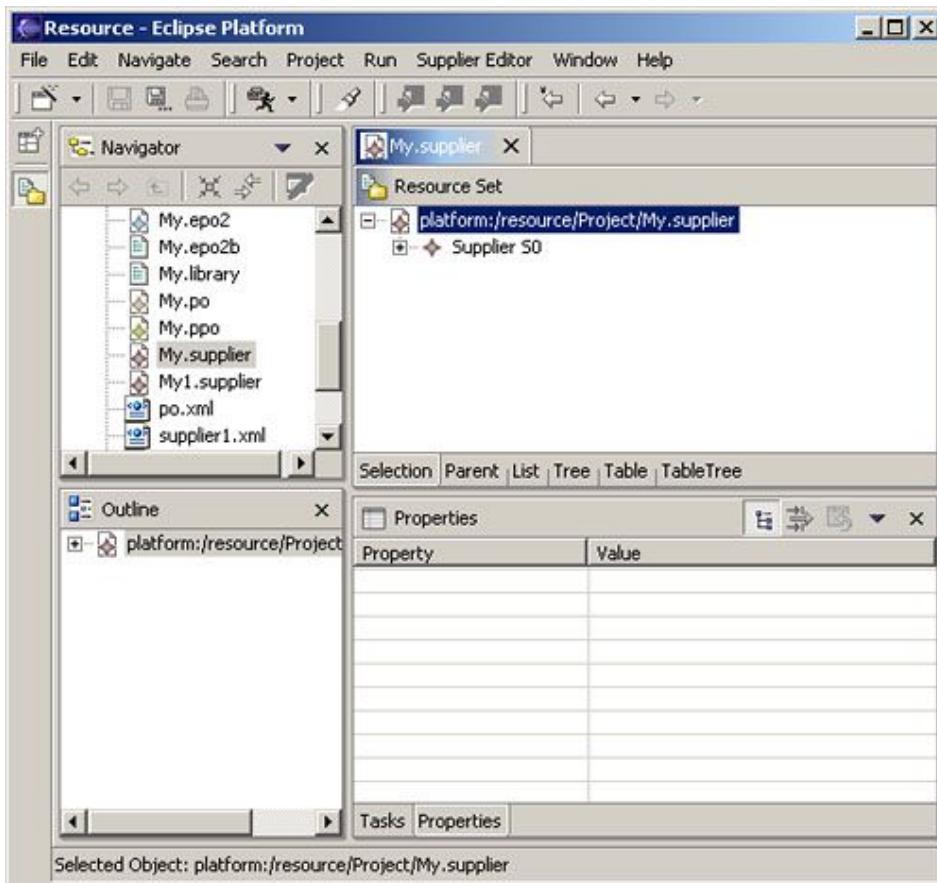


Figure 12.12. Editing a resource with cross-document references.

If we now proceed to select the supplier S0 in this viewer, the referenced resource, `My_backup.supplier`, will then also appear as shown in Figure 12.13. The `backupSupplier` reference is accessed for the first time when the supplier is selected and it is needed for display in the property sheet. The `EcoreUtil.resolve()` call in the generated `getBackupSupplier()` method causes the referenced resource, `My_backup.supplier`, to be lazily loaded at this point. We described how this all works way back in Chapter 2, if you need a refresher.

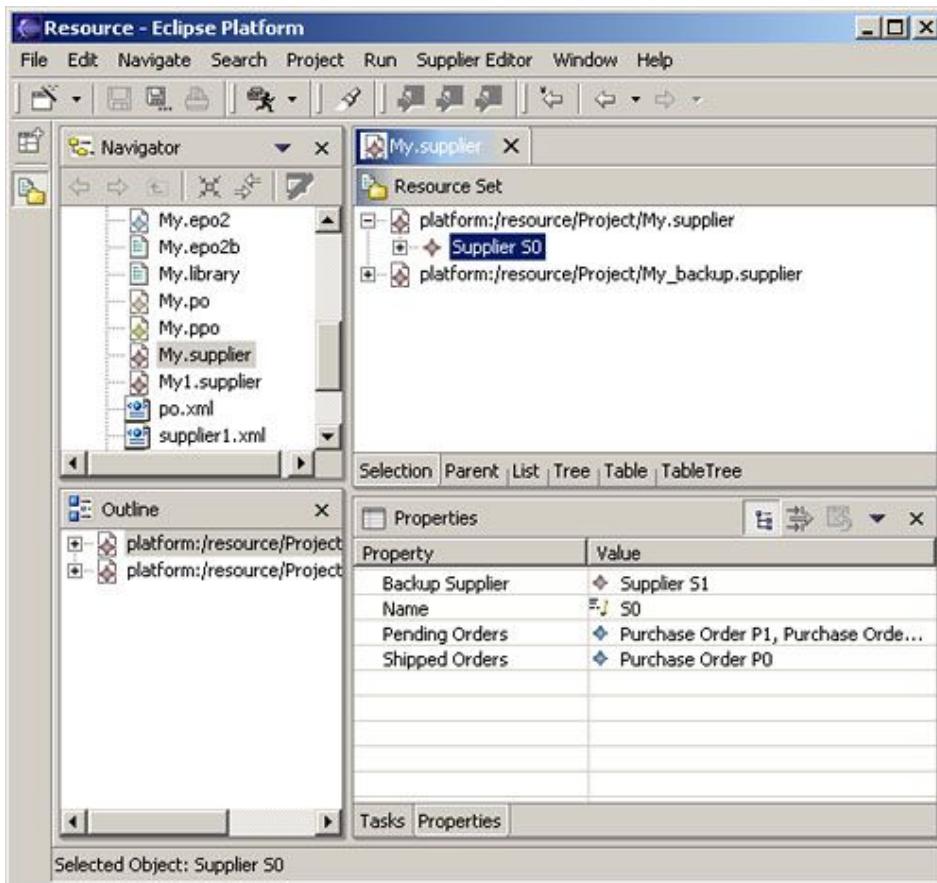


Figure 12.13. Lazy resolving of cross-document references.

We can now use the editor to initialize and add customers and purchase orders to either supplier, as shown in Figure 12.14. Notice in the property sheet that the “Orders” associated with customer 1 has been set to include purchase order P3, which is contained by the backup supplier S1. If you think back to the design decision we made in Section 12.7.2 to use the resource set as the second argument to the `UsageCrossReferencer.find()` method when we implemented the `getComboBoxObjects()` method for the `previousOrder` property, you’ll now understand why. If we had instead used the resource or supplier as the second argument, it would now be impossible to set the `previousOrder` reference of P3 to either of the other orders associated with customer 1 (that is, to P1 or P2). We would only have been able to set it to another order under S1, of which there are none in our example.

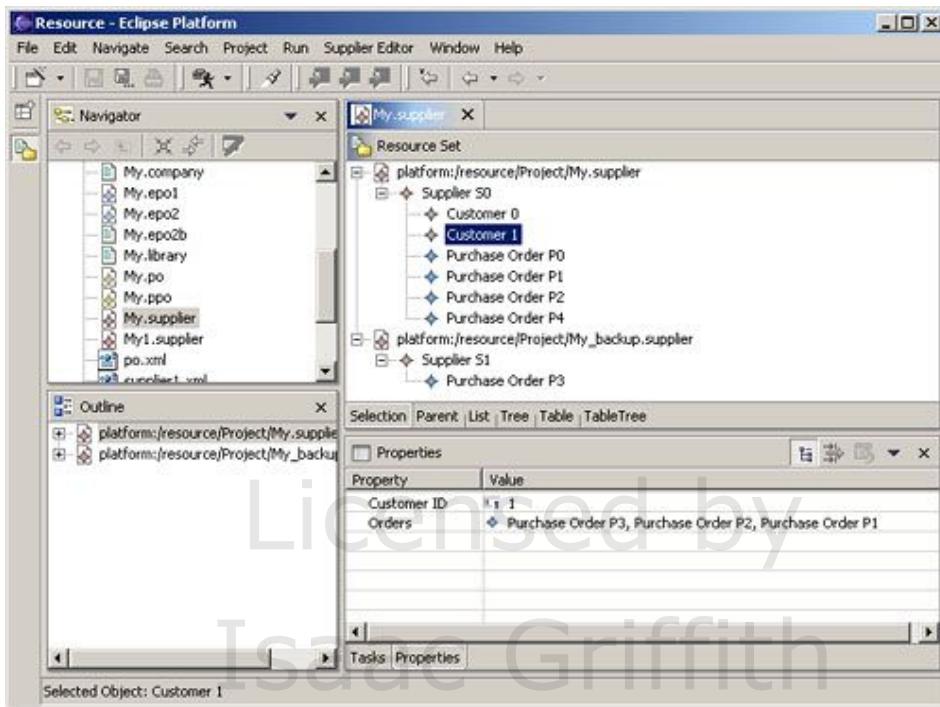


Figure 12.14. Concurrently editing two supplier resources.

Although the generated editor allows us to edit the two supplier resources concurrently, it will not by default save anything but the first resource it was opened on, when the user invokes the **Save** action. Now that we're making changes to two resources in the editor session, we need to make sure that we save both. We can do this by making the following change to the `doSave()` method in class `SupplierEditor`:

```
/*
 * @generated NOT
 */
public void doSave(IProgressMonitor progressMonitor) {
    // Do the work within an operation because this is a long running
    // activity that modifies the workbench.
    //
    WorkspaceModifyOperation operation =
        new WorkspaceModifyOperation() {
            // This is the method that gets invoked when the operation
            // runs.
            //
            protected void execute(IProgressMonitor monitor)
                throws CoreException {
                try {
                    // Save the resource to the file system.
                    //
                    Resource savedResource = (Resource)
                        editingDomain.getResourceSet().getResources().get(0);
                    for (Iterator iter = editingDomain.getResourceSet().
                        getResources().iterator(); iter.hasNext(); ) {
                        Resource savedResource = (Resource)iter.next();
                        savedResources.add(savedResource);
                        savedResource.save(Collections.EMPTY_MAP);
                    }
                }
                catch (Exception exception) {
                    exception.printStackTrace();
                }
            }
        };
    operation.run(progressMonitor);
}
```

```

};

...
}

```

As you can see, all we needed to do here was change it to call `save()` on every resource in the resource set, instead of just the assumed single one.

So now that we're able to edit and save multiple linked resources, the last question we need to answer is how to create and link to the second document in the first place. In our example, we started with a resource `My.supplier` that contained the cross-document reference to the second resource. How did we create it, you might ask?

There are several ways we could do this, including extending the editor with new actions to find and load a backup resource. For our purposes, we chose to use the simpler approach of modifying the ExtendedPO3 Wizard to create the two resources up front. All we needed to do was change the `performFinish()` method in class `SupplierModelWizard`, like this:

```

/**
 * @generated NOT
 */
public boolean performFinish() {
    try {
        // Remember the file.
        //
        final IFile modelFile = getModelFile();

        // Do the work within an operation.
        //
        WorkspaceModifyOperation operation =
            new WorkspaceModifyOperation() {
                protected void execute(IProgressMonitor progressMonitor) {
                    try {
                        // Create a resource set
                        //
                        ResourceSet resourceSet = new ResourceSetImpl();

                        // Get the URI of the model file.
                        //
                        URI fileURI = ...

                        // Create a resource for this file.
                        //
                        Resource resource = resourceSet.createResource(fileURI);

                        // Add the initial model object to the contents.
                        //
                        EObject rootObject = createInitialModel();
                        if (rootObject != null) {
                            resource.getContents().add(rootObject);
                        }

                        String backupName = modelFile.getFullPath().toString();
                        backupName =
                            backupName.substring(0,
                                backupName.indexOf(".supplier"))
                            + "_backup.supplier";
                        URI backupURI =
                            URI.createPlatformResourceURI(backupName);
                        Resource backupResource =
                            resourceSet.createResource(backupURI);

                        Supplier supplier =
                            SupplierFactory.eINSTANCE.createSupplier();
                        Supplier backupSupplier =
                            SupplierFactory.eINSTANCE.createSupplier();
                        supplier.setBackupSupplier(backupSupplier);
                    }
                }
            };
        operation.execute(null);
    } catch (CoreException e) {
        throw new RuntimeException(e);
    }
    return true;
}

```

```
    backupResource.getContents().add(backupSupplier);
    backupResource.save(Collections.EMPTY_MAP);

    resource.getContents().add(supplier);

    // Save the contents of the resource to the file system.
    //
    resource.save(Collections.EMPTY_MAP);
}
...
}
```

Here we simply changed it to create two instances of class **Supplier**, putting the first one into the current resource and the second into another resource that we create with the same base name as the first, only with “_backup” appended. Because we’re also no longer calling the `createInitialModel()` method, and thereby ignoring the user’s selected root object for the resource, we should also change the `addPages()` method so that the initial object creation page is no longer displayed:

```
/***
 * @generated NOT
 */
public void addPages() {
    ...
    addPage(initialObjectCreationPage);
}
```

With this final change, our example is complete. If you managed to follow along and successfully implement the example, you can now consider yourself to be a bona fide EMF novice. After having read this much of the book, you may have been hoping that you had advanced further than the “lowly” level of novice. Well, the next part of the book will cover several more advanced topics and provide examples to help you advance to the intermediate level or, dare we say it, become an EMF expert.

Part IV. Programming with EMF

Chapter 13. EMF Client Programming

So far, we've talked a great deal about EMF's ability to generate code from a model, and how it can produce a model from various sources. EMF is, however, not just a generator tool; it's also a powerful runtime framework. In the following sections we will discuss the functionality provided by the runtime framework, and how to effectively program using it. The main topics we will cover are

1. ***Creating objects and accessing metadata.*** You use EMF factories and packages for this.
2. ***Resource management.*** Resources are persistent containers of objects.
3. ***Object serialization.*** How to persistently save your data using XML and XMI, and how to customize the default serializer.
4. ***Object adapting.*** How to register for notifications when your objects change and how to extend the behavior of your objects.
5. ***Using the reflective API.*** Like Java, EMF has a reflective API that allows you to work with any EMF objects, regardless of the model that defines them.
6. ***Using dynamic EMF.*** Although we have talked about code generation and the benefits you can get from using it, EMF also allows you to create data without generating any code at all.

Unless otherwise noted, we'll use the ExtendedPO2 model, introduced in Chapter 12, for the examples in this chapter.

Packages and Factories

As we saw in Chapter 9, EMF generates package and factory classes that allow you to easily access a model's metadata and create instances of the modeled classes. These generated classes are subclasses of the generic Ecore classes, **EPackage** and **EFactory**. The generated package and factory are only conveniences; everything that they provide is also available using generic methods in their base interfaces. For example, the generated **EPO2Package** provides a convenient method for accessing the **PurchaseOrder** class:

```
ECClass purchaseOrderClass = epo2Package.getPurchaseOrder();
```

Without the generated class, we would simply retrieve the class by using the `getEClassifier()` method from the **EPackage** interface, like this:

```
ECClass purchaseOrderClass =
  (ECClass)epo2Package.getEClassifier("PurchaseOrder");
```

In a similar way, we can access the attributes, references, and everything else provided by the generated package. The generated methods are easier to use and more efficient, but otherwise nothing new. The **EPackage** and **EFactory** interfaces give us the added ability to use the package and factory to work with and create objects generically. This is critically important when deserializing resources using the generic XML loader, as we'll see in Section 13.3.1.

Accessing Package Metadata Generically

Here's an example of a method that prints the names of the classes and data types in a package, along with their contents. For every **EClass**, we'll print the names of its attributes and references. For an **EEnum**, we'll display the names of the enum literals, while for other **EDataTypes**, we'll just print the instance class name. Here's the method:

```
public static void printClasses(EPackage ePackage)
{
    for (Iterator iter =
        ePackage.getEClassifiers().iterator(); iter.hasNext(); ) {
        EClassifier classifier = (EClassifier)iter.next();
        System.out.println(classifier.getName());
        System.out.print("  ");

        if (classifier instanceof EClass) {
            EClass eClass = (EClass)classifier;
            for (Iterator ai =
                eClass.getEAttributes().iterator(); ai.hasNext(); ) {
                EAttribute attribute = (EAttribute)ai.next();
                System.out.print(attribute.getName() + " ");
            }
            for (Iterator ri =
                eClass.getEReferences().iterator(); ri.hasNext(); ) {
                EReference reference = (EReference)ri.next();
                System.out.print(reference.getName() + " ");
            }
        }
        else if (classifier instanceof EEnum) {
            EEnum eEnum = (EEEnum)classifier;
            for (Iterator ei =
                eEnum.getELiterals().iterator(); ei.hasNext(); ) {
                EEnumLiteral literal = (EEEnumLiteral)ei.next();
                System.out.print(literal.getName() + " ");
            }
        }
        else if (classifier instanceof EDataType) {
            EDataType eDataType = (EDataType)classifier;
            System.out.print(eDataType.getInstanceClassName() + " ");
        }

        System.out.println();
    }
}
```

As you can see, we simply iterate through the set of classifiers, returned by the `getEClassifiers()` method. For each one, we check if the type is an `EClass`, `EEEnum`, or `EDataType`, and then act accordingly. If we were to call this method with the package from `ExtendedPO2`, `EPO2Package.eINSTANCE`, it would produce the following output:

```
Item
  productName quantity USPrice comment shipDate partNum order
USAddress
  street city state zip
PurchaseOrder
  comment orderDate status totalAmount items billTo shipTo
  customer previousOrder
Address
  name country
Supplier
  name customers orders pendingOrders shippedOrders
Customer
  customerID orders
GlobalAddress
  location
GlobalLocation
  countryCode
```

```

OrderStatus
  Pending BackOrder Complete
SKU
  java.lang.String
Date
  java.util.Date

```

You can use the `EPackage` interface to access all of the metadata in a model, whether it's a generated package, like `EPO2Package` in this example, or a dynamically created one. We'll discuss dynamic packages further in Section 13.6.

Generic Object Creation

We've seen previously how the generated `EPO2Factory` can be used to create instances of model classes. For example, we could create an instance of class `PurchaseOrder` like this:

```
PurchaseOrder order = epo2Factory.createPurchaseOrder();
```

Using the `EFactory` interface, we would do the same thing like this:

```

EClass purchaseOrderClass = ...
EFactory epo2Factory = ...
PurchaseOrder order = epo2Factory.create(purchaseOrderClass);

```

As you can see, to do this we need to have access to the `ExtendedPO2` model's `EFactory` and the `EClass` for `PurchaseOrder`. If the model is generated as it is, we can access them quite simply, like this:

```

EClass purchaseOrderClass =
  EPO2Package.eINSTANCE.getPurchaseOrder();
EFactory epo2Factory = EPO2Factory.eINSTANCE;

```

Of course, if this is the case we might as well just call the generated `createPurchaseOrder()` method and be done with it. Let's instead think about an application that doesn't have access to the generated package or factory, but simply wants to create an instance of class `PurchaseOrder`, knowing only its name and package URI.

So, to create an instance, we need access to the factory and the class. Actually, all we really need is the package, because we can get the other two from it. We can get the class by calling the `getEClassifier()` method, as we saw earlier, and the factory by calling the `getEFactoryInstance()` method, which is also on the `EPackage` interface.

You may recall from Chapter 5 that `EPackage` is one of the objects in a serialized core model; it's the root:

```

<?xml version="1.0" encoding="ASCII"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi=...
  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
  name="epo2" nsURI="http://com/example/epo2.ecore"
  nsPrefix="com.example.epo2">
<eClassifiers xsi:type="ecore:EClass" name="Item">
  ...

```

So, to access the package, we could simply retrieve it by loading a serialized form of the model, that is, the `.ecore` file:

```

ResourceSet resourceSet = new ResourceSetImpl();
URI fileURI = URI.createURI(".../ExtendedPO2.ecore");
Resource resource = resourceSet.getResource(fileURI, true);
EPackage epo2Package = (EPackage)resource.getContents().get(0);

```

Although this would work, this is not the way it's usually done. Instead of loading serialized core models, EMF packages are usually accessed using the package registry. The registry provides a mapping from namespace URIs to **EPackages**. The interfaces for the registry are nested in the interface for **EPackage**:

```
public interface EPackage extends ENamedElement
{
    public interface Descriptor
    {
        EPackage getEPackage();
    }

    public interface Registry extends Map
    {
        EPackage getEPackage(String nsURI);

        Registry INSTANCE =
            new org.eclipse.emf.ecore.impl.EPackageRegistryImpl();
    }

    ...
}
```

Interface **EPackage.Registry** allows you to get a package given its **nsURI**. Since it extends from **java.util.Map**, packages are added to the registry using the **put()** method. An entry can be either an **EPackage** or an **EPackage.Descriptor**. Descriptors are primarily used to register packages during Eclipse plug-in initialization, while delaying the loading of the package until it's actually used. If you've ever looked at the generated *plugin.xml* file in an EMF model plug-in, you will have noticed that it includes one or more extensions that look something like this:

```
<extension point="org.eclipse.emf.ecore.generated_package">
<package
    uri = "http://com/example/epo2.ecore"
    class = "com.example.epo2.EPO2Package" />
</extension>
```

During plug-in initialization, this will cause an **EPackage.Descriptor** for the **EPO2Package** to be added to the registry. Later when the package with the registered URI is accessed by calling **getEPackage()** on the registry, it will, in turn, call **getEPackage()** on the descriptor, which will create and initialize the actual **EPackage**.

Generated packages that are not registered in a plug-in extension (for example, when EMF is run stand-alone) are added to the registry during their construction. In this case, a generated subclass' constructor passes to its base class, **EPackageImpl**, the package namespace URI, which is used to register the package like this:

```
protected EPackageImpl(String packageURI, EFactory factory) {
    super();
    Registry.INSTANCE.put(packageURI, this);
    ...
}
```

Once registered, the package can be accessed wherever it's needed by simply calling the **EPackage.Registry.INSTANCE.getEPackage()** method. For example, here is a method that, using the package registry, will create an instance of a class, given nothing but its name and package URI:

```
public static EObject createObject(String packageNsURI,
                                    String className) {
    EPackage ePackage =
        EPackage.Registry.INSTANCE.getEPackage(packageNsURI);
    EClass eClass = (EClass)ePackage.getEClassifier(className);

    EFactory eFactory = ePackage.getEFactoryInstance();
```

```

EObject eObject = eFactory.create(eClass);

return eObject;
}

```

Using this method, we can now create an instance of class **PurchaseOrder** like this:

```

EObject order = createObject("http://com/example/epo2.ecore",
    "PurchaseOrder");

```

Package location is particularly important when loading EMF resources. Although the package registry is used most commonly to locate generated packages during load, EMF also supports two other mechanisms for mapping a namespace URI to its corresponding EPackage: a `URIConverter` and the `xsi:schemaLocation` attribute. We'll look at how `URIConverters` can be used to locate packages in the following section. Using an `xsi:schemaLocation` attribute in serialized instance documents is the preferred approach for locating dynamic EMF models, as we'll see in Section 13.6.

The EMF Persistence API

The persistence framework of EMF is centered around four fundamental interfaces: `Resource`, `ResourceSet`, `Resource.Factory`, and `URIConverter`. These interfaces collectively define the API for saving and loading EMF models to and from persistent storage. EMF provides a default (XML) persistence implementation based on these interfaces, but the API itself is general enough to support persistence using any kind of actual backing store—XML-based or not, even stream-based or not.

In the remainder of this section we'll briefly describe each of these interfaces, and try to point out the most important aspects of their design. Then, in Section 13.3, we'll look at the default resource implementations provided by EMF.

However, before we begin to describe the interfaces, we should understand a few things about URIs, which serve a crucial role in the EMF persistence framework.

URI

URI (Uniform Resource Identifier) is an open-ended standard for identifying and locating data of various types. The data could be some file in a file system or on the Internet, an object in a file, or in a database of some kind, or any other type of data you might imagine.

EMF uses URIs extensively in the persistence framework. We already saw in Section 13.1.2 that EMF uses URIs to identify packages. It also uses URIs to uniquely identify resources and to reference the objects in them.

A URI is a string composed of three fundamental parts: a *scheme*, a *scheme-specific part*, and an optional *fragment*. EMF provides a URI implementation class, `org.eclipse.emf.common.util.URI`, which provides convenient ways of working with and manipulating a URI and its parts.

The URI scheme identifies the protocol used to access the resource. Some common ones are “file”, to directly identify the location of files, and “jar”, to identify files archived within a JAR file. In Eclipse, we use the “platform” protocol for URIs that are paths to resources in the Eclipse workspace.¹ The scheme is the first part of the URI, separated from the rest by the “:” character. For example, `platform:/resource/project/po.xml` is a URI whose scheme is “platform”.

¹EMF also provides a map, `EcoreUtil.getPlatformResourceMap()`, that can be used to map platform URIs to physical file system directories, when running EMF stand-alone. This mapping is used by EMF's default `URIConverter` implementation, allowing one to access the same EMF resources either inside or outside of Eclipse.

The scheme-specific part includes everything between the scheme and the fragment. In general, its format and interpretation depend entirely upon the scheme. However, the schemes that EMF concerns itself with all use a common hierarchical format for this part. This format may include an *authority*, a *device*, and any number of *segments*. An authority, if included, is separated from the scheme by “/”. It generally specifies a host in a networked environment. The device, if included, is preceded by “/”. Finally, the segments, if any, are also preceded and separated from each other by “/”. Together, the device and segments constitute a local path to a resource’s location. For example, here is a URI with a “file” scheme, no authority, a device, and three segments: `file:/c:/dir1/dir2/myfile.xml`.²

The URI *fragment* identifies a part of the contents of the resource specified by the scheme and authority. It’s separated from the rest of the URI by the “#” character. For example, the URI `file:/c:/dir1/dir2/myfile.xml#loc` identifies something in *myfile.xml*, which would be located using the fragment “loc”.

EMF uses URIs with fragments to reference **EObjects** in resources. Each EMF resource has a unique URI. Each object in a resource has a unique fragment that identifies that object within it.

In the following sections, we’ll see how these URIs and fragments are created and used. For more details on URI syntax, refer to the Internet Engineering Task Force RFC 2396, *Uniform Resource Identifiers (URI): General Syntax*, available at www.ietf.org/rfc/rfc2396.txt.

URIConverter

A `URIConverter` is used by the framework to normalize (that is, convert) an input URI into an actual URI of a resource. It can be used to provide mappings from namespace URIs (for example, `http://com/example/epo2.ecore`) to physical file URIs, or to redirect old (or alias) URI references to a different actual location.

A `URIConverter` maintains a set of URI to URI mappings, which can be specified by clients. The `normalize()` method is used to convert a URI to its normalized form, based on any mappings that may apply. For example, we could use a `URIConverter` to map a package namespace URI to a physical (`.ecore`) resource like this:

```
URIConverter converter = new URIConverterImpl();

URI uri1 = URI.createURI("http:///somemodel.ecore");
URI uri2 =
    URI.createURI("platform:/resource/project/somemodel.ecore");
converter.getURIMap().put(uri1, uri2);

URI normalized = converter.normalize(uri1);
System.out.println(normalized);
```

If we run this program, it will output the following:

```
platform:/resource/project/somemodel.ecore
```

As you can see, the `URIConverter.normalize()` method will simply replace a logical URI it finds (as a key) in its map with the associated value (URI). We could use a `URIConverter` like this as an alternative to the package registry that we discussed in Section 13.1.2, for locating packages.

URI mappings that end with “/” can be used to replace just a URI prefix. For example, we could do the following:

²You may have seen slightly different URIs before, like `file:///c:/dir1/dir2/myfile.xml`, and be wondering what the difference is. Because of the additional “/”, this URI specifies an empty string as its authority, while the other specifies no authority at all. Both refer to the same resource, but when compared as URIs, they are different.

```
URI uri1 = URI.createURI("http:///com/example/");
URI uri2 = URI.createURI("platform:/resource/project/");
converter.getURIMap().put(uri1, uri2);

URI uri3 = URI.createURI("http:///com/example/somemodel.ecore");
URI normalized = converter.normalize(uri3);
System.out.println(normalized);
```

This will produce the same normalized URI, as in the previous example:

```
platform:/resource/project/somemodel.ecore
```

The `normalize()` method is also recursive; if the output of a mapping matches the input to another, it will be normalized again.

In addition to normalizing URIs, a `URIConverter` is also used to create input and output streams for a URI. The `createInputStream()` and `createOutputStream()` methods normalize a specified URI, by calling the `normalize()` method, and then immediately open a stream on it. The default implementation of `URIConverter` provides support for the `file` and `platform` protocols, whether it's running inside or outside of Eclipse.

The primary use of a `URIConverter` is by resource sets, for locating resources.

Resource

The `Resource` interface represents a persistent container of `EObjects`, the actual location of which is identified by its URI. An object is added to a resource by inserting it into the contents list returned by the `getContents()` method, or by adding it to a containment association of some other object that is, itself, in the resource. Adding to a resource's contents is much like adding to an EMF containment reference; it will remove the object from any other container (or resource) that it may already be in.

The most important methods on the `Resource` interface are `save()` and `load()`, which define the actual persistent format of the objects, and the `getEObject()` and `getURIFragment()` methods, which support URI fragment lookup. Other methods allow you to access the resource's URI and resource set, turn on or off modification tracking if you want to keep track of changes in the resource,³ and to access errors or warnings that may have occurred during a `save()` or `load()` operation.

Save and Load

The `save()` and `load()` methods are used to copy a resource to and from its persistent storage. The default implementations of these methods, in class `ResourceImpl`, use a `URIConverter` (provided by the resource set) to normalize and open streams for the resource's URI, and then delegate to corresponding `doSave()` and `doLoad()` methods. These `do()` methods are unimplemented in `ResourceImpl`; they are implemented in storage-specific resource subclasses. We'll look at the EMF-provided resource implementation classes in Section 13.3.

Both the `save()` and `load()` methods can be passed options that control their behavior. The options are passed as a `java.util.Map`, containing name-value pairs:

```
void save(Map options) throws IOException;
void load(Map options) throws IOException;
```

³This can be a costly feature, so use it with caution.

The options recognized by a resource are implementation specific. We'll look at the options supported by EMF's XML resource implementation in Section 13.3.2. A resource's default implementation is invoked by passing `null` for the `options` argument.

The `Resource` interface includes a second version of the `save()` and `load()` methods, which include a stream argument:

```
void save(OutputStream outputstream,  
         Map options) throws IOException;  
void load(InputStream inputstream, Map options) throws IOException;
```

You might think that this implies that EMF resources are inherently "stream based." Although most resources used with EMF tend to be stream based, including the XML resources provided with EMF, non-stream-based (for example, relational database) resources can also be implemented. In these cases, the stream-based versions of the `load()` and `save()` methods can be handled in one of two ways:

1. Throw `UnsupportedOperationException`. This will be the default behavior if a (non-stream-based) resource implementation class subclasses from `ResourceImpl`.
2. Treat them as "Save As" and "Open With" operations. Since EMF includes default serialization of any model to XML, by subclassing a non stream-based resource implementation class from either `XMLResourceImpl` or `XMIResourceImpl`, you can simply inherit, for free, the ability to produce an XML serialization of any resource, regardless of its actual persistent form.

EMF resources can also be unloaded by calling the `unload()` method. Unloading will turn all the objects in the resource into proxies, which will result in a demand-reload of the resource (by the `EcoreUtil.resolve()` method) the next time it is called with any of the objects. This allows you to bring a resource up to date if, for example, the underlying file has changed since it was last loaded.

URI Fragments

In Section 13.2.1 we mentioned that the fragment of a URI (that is, the part after the "#" character) is used to identify the objects in a resource. EMF's default resource implementation base class, `ResourceImpl`, uses an XPath-like fragment path to locate objects. For example, assume we have a resource that contains an instance of class `PurchaseOrder`, which in turn contains a set of items under it. The fragment path `//@items.0` would identify the first item (that is, `Item` instance under the `PurchaseOrder`). For the remaining items, we would use the path `//@items.1`, `//@items.2`, and so on. If the `PurchaseOrder` instance was itself contained by a `Supplier`, for example, then the path would include another level, for example `//@orders.0/@items.2`.

The `getEObject()` method on the `Resource` interface can be used to retrieve an object, given its fragment path:

```
Resource resource = ...  
Item item = (Item)resource.getEObject("//@orders.0/@items.2");
```

Determining the fragment path of a given object (that is, the reverse operation) can be done using the `getURIFragment()` method:

```
Item item = ...  
String fragment = resource.getURIFragment(item);
```

The `getEObject()` and `getURIFragment()` methods are implemented in class `ResourceImpl`, but delegate to the objects on the path. The implementation of `getEObject()` simply breaks the fragment path into its “/” separated components, and then iteratively calls the `eObjectForURIFragmentSegment()` method on interface `InternalEObject`. Each call is passed the path component to the next object in the path, which it is expected to return.

The `getURIFragment()` method works in a similar way to produce a fragment path for a given object. Starting from the object, it collects the set of other objects on its path by navigating to the root using the `eContainer()` method. For each object along the path it calls the `eURIFragmentSegment()` method (also on `InternalEObject`), passing it the next (contained) object along with the object's containment feature, from which it produces the required fragment component for that step along the path.

The default implementation of the `eObjectForURIFragmentSegment()` and `eURIFragmentSegment()` methods is provided in class `EObjectImpl`. Delegating these methods to the model objects this way, however, allows one to customize the fragment path for specific models. If, for example, a model has hierarchically (uniquely) named objects, the position-based default syntax can be replaced with a name-based lookup. The Ecore model is such a model; it overrides the two fragment methods in class `EModelElementImpl`. Recall that the simple purchase order model, which we introduced in Chapter 2, has a XMI serialization that looks something like this:

```
<?xml version="1.0" encoding="ASCII"?>
<ecore:EPackage ... name="simplepo" ... />
  <eClassifiers xsi:type="ecore:EClass" name="PurchaseOrder">
    <eReferences name="items"
      eType="#//Item" upperBound="-1" containment="true"/>
      <eAttributes name="shipTo" ... />
      <eAttributes name="billTo" ... />
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="Item">
      <eAttributes name="productName" ... />
      <eAttributes name="quantity" ... />
      <eAttributes name="price" ... />
    </eClassifiers>
  </ecore:EPackage>
```

Notice that the `eType` of the `items` reference is the URI `#//Item`. This name-based fragment is provided by Ecore's override of `eURIFragmentSegment()`. If Ecore had not overridden it, the default implementation in `EObjectImpl` would have produced `#//@eClassifiers.1` instead.

One thing that you may have noticed is that all of the fragment paths that we've shown start with `//`. Given our description of the algorithm, you may have expected the previous example path to be `#/simplepo/Item`, instead of just `#//Item`. The reason it isn't so is because the `getURIFragment()` method optimizes the first segment, especially when there is only a single root object in the resource, which is the typical case.

The `getURIFragment()` method handles the first segment by calling another method on the resource, `getURIFragmentRootSegment()`, which by default returns an empty string (“”) if there is only a single object in the resource's contents list, or the index of the object in the list if there are many. A corresponding method, `getEObjectForURIFragmentRootSegment()`, is used by `getEObject()`. You would override both of these methods if, for example, you wanted to change the root segment lookup to be name instead of index based.

Resource.Factory

Resource factories, as their name implies, are used to create resources. A resource factory is located using a registry, much like the package registry we described in Section 13.1.2. A resource factory, however, is registered against categories of URIs, instead of the URI itself. For example, the default registry allows you to register a resource factory based on a URI scheme or extension.

The resource factory interface, `Resource.Factory`, is nested in the `Resource` interface, as shown:

```
public interface Resource extends Notifier
{
    interface Factory
    {
        Resource createResource(URI uri);

        interface Descriptor
        {
            Factory createFactory();
        }
    }

    interface Registry
    {
        Factory getFactory(URI uri);

        Map<String, Factory> getProtocolToFactoryMap();

        String DEFAULT_EXTENSION = "*";

        Map<String, Factory>getExtensionToFactoryMap();

        Registry INSTANCE = new ResourceFactoryRegistryImpl();
    }
}
```

As you can see, the `Registry` interface is nested inside of the `Factory` interface. Just as with the package registry, resource factories can be registered using a `Descriptor`, instead of the factory itself. Extension points are used to register descriptors during plug-in initialization, delaying creation of the factories until they're needed.

A static registry instance can be accessed using the `INSTANCE` field. The default implementation is in class `ResourceFactoryRegistryImpl`. Here's the algorithm used in its `getFactory()` method:

1. Check for a factory in the `protocolToFactoryMap`, using the scheme of the URI.
2. If nothing was found, check the `extensionToFactoryMap` using the file extension of the URI.
3. If still nothing was found, check the `extensionToFactoryMap` using the `DEFAULT_EXTENSION` (that is, the wildcard character “`*`”).
4. If no extension match was found, call the `delegatedGetFactory()` method. This allows you to supply your own factory registry, with its own lookup criteria.
5. If a descriptor was found, instead of an actual factory, call the `createFactory()` method on the descriptor to create the factory.
6. Finally, return the factory if one was found, or null.

We've mentioned previously that the XMI serialization is the default format used by EMF. If no other registered factory applies for a given URI, the XMI Resource's factory, `XMIResourceFactoryImpl`, will be used. This happens because EMF adds it to the registry, using the `DEFAULT_EXTENSION ("")` extension, during plug-in initialization. You can see its registration if you look at the plug-in manifest file of EMF's default resource implementation plug-in, `org.eclipse.emf.ecore.xmi`:

```
<extension point = "org.eclipse.emf.ecore.extension_parser">
  <parser type="*"
    class="org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl"/>
</extension>
```

If you create your own resource implementation class, you would need to create a factory for it and then register it, in a similar way, in your own plug-in. For example, assume that you have created your own resource implementation class, `EPO2ResourceImpl`, which you want to use to persist `.epo2` resources. To do this, you would need to create a `Resource.Factory` implementation class, `EPO2ResourceFactoryImpl`, which simply implements the `createFactory()` method to return an instance of the resource class, `EPO2ResourceImpl`. Then, you would add the following extension to the `plugin.xml`:

```
<extension point = "org.eclipse.emf.ecore.extension_parser">
  <parser type="epo2"
    class="com.example.epo2.impl.EPO2ResourceFactoryImpl"/>
</extension>
```

When running EMF outside of Eclipse (that is, stand-alone), plug-in initialization doesn't take place and consequently, the default resource factory won't be registered. In this case, you'll need to add a line in your program, for example, early in `main()`, to explicitly register it, like this:

```
Resource.Factory.Registry.INSTANCE.
  getExtensionToFactoryMap().put("", new XMIResourceFactoryImpl());
```

Resource factories are used by resource sets, which provide the actual client interface for creating resources, as we'll see in the following section.

ResourceSet

A resource set represents a collection of resources that have been created or loaded together. It provides the main client API for resource management with the methods `createResource()`, `getResource()`, and `getEObject()`.

The `createResource()` method is used to create a new, empty resource in the set. The `getResource()` method also creates a resource, but then it attempts to load it using the resource's URI. The `getEObject()` method is used during demand-load to first load the resource of an object, if necessary, by calling `getResource()` using the specified URI (excluding the fragment). Then it calls the `getEObject()` method, described in Section 13.2.3, on the returned resource to locate the object using the fragment from the URI.

The implementation of the `ResourceSet` methods uses the resource factory and registry to create the resources and a `URIConverter` to normalize the URI, if necessary. Methods on the `ResourceSet` interface provide access to the registry and `URIConverter` that will be used.

Instead of creating resources using the Java `new` operator or even using the resource factory's `createResource()` method directly, clients should always create resources by calling the `getResource()` or `createResource()` method on the resource set. The reason that this is important is because the resource set, in addition to simply delegating to the resource factory for the creation, keeps track of which URIs have already been opened. This guarantees that we will never have two in-memory copies of the same resource.

The resource set also has the critical role of demand-loading resources that are referenced by proxies in other resources. Consider the following simple example resource, *supplier.epo2*:

```
<?xml version="1.0" encoding="ASCII"?>
<com.example.epo2:Supplier ... >
  <orders comment="PO1">
    <customer href="supplier2.epo2#/customers.0"/>
  </orders>
</com.example.epo2:Supplier>
```

Here we have a simple serialized ExtendedPO2 (epo2) model with a cross-document reference, indicated by the `href` attribute, to an object in another resource: *supplier2.epo2*. Simply loading this document by creating a resource and calling the `load` method like this:

```
URI fileURI =
  URI.createURI("platform:/resource/project/supplier.epo2");
Resource resource = new XMLResourceImpl(fileURI);
resource.load(null);
```

would appear to work fine. At this point in time, the cross-document reference will be (in the `orders` list) a Customer proxy object containing the URI of the actual object (that is, `platform:/resource/project/supplier2.epo2#/customers.0`). If we now navigate through the objects in the resource, when we reference the Customer, the `EcoreUtil.resolve()` method will be called, which will attempt to resolve the proxy. The `resolve()` method will attempt to navigate to the resource set and then call the `getEObject()` method on it. In this case, however, since we didn't use a resource set to load the first document, there is no resource set to navigate to, so the proxy will not be resolved. Had we loaded the first resource using the `getResource()` method on a resource set, like this:

```
ResourceSet resourceSet = new ResourceSetImpl();
URI fileURI =
  URI.createURI("platform:/resource/project/supplier.epo2");
Resource resource = resourceSet.getResource(fileURI, true);
```

then the `resolve` would have succeeded, and we would now have two resources in the resource set. The added benefit is that you can use the `getResources()` method to access all the resources that have been created or loaded in the set. We've already seen, in Chapter 12, how this can be useful when searching for objects using `UsageCrossReferencers`, for example.

EMF Resource Implementations

The EMF plug-in `org.eclipse.emf.ecore.xmi` provides two resource implementations supporting serialization to XML or XMI. The primary implementation is provided by the framework class `XMLResourceImpl`, which can be used to serialize any EMF model to XML. Its implementation is flexible with regard to the format of the XML it produces. You can control it in many different ways.

In Chapter 2, we said that the default resource implementation used by EMF is XMI. Class `XMIResourceImpl`, a simple subclass of `XMLResourceImpl`, provides the support for serialization and loading of XMI 2.0 documents.

In the following sections, we'll first look at the default behavior of `XMLResourceImpl` and show how it can be customized. Then we'll explain how, when an XML Schema is provided, the framework will control and customize it to produce schema-conformant output. Finally, we'll show how `XMIResourceImpl` extends from and specializes `XMLResourceImpl` to support XMI.

XML Resources

Serialization to XML is provided by the `save()` method of the framework resource implementation class `XMLResourceImpl`. To illustrate its default behavior, we'll start by creating a single instance and serializing it like this⁴:

```
USAddress address = EPO2Factory.eINSTANCE.createUSAddress();
Resource resource =
    new XMLResourceImpl(URI.createURI(".../address.xml"));
resource.getContents().add(address);
resource.save(null);
```

Here we create a single instance of class **USAddress**, add it to an XML resource, and then serialize it to the file *address.xml*. The resulting file will contain the following:

```
<com.example.epo2:USAddress
  xmlns:com.example.epo2="http://com/example/epo2.ecore"/>
```

As you can see, the **USAddress** is serialized as an XML element using an XML namespace. The namespace prefix (`com.example.epo2`) and corresponding URI (`http://com/example/epo2.ecore`) come from the `nsPrefix` and `nsURI` attributes of the **EPackage** for the model (`EPO2Package` in our example). Notice that the namespace declaration appears as an attribute of the `USAddress` element itself. This is because namespace declarations are added to the “root” element of the document. In our example, the **USAddress** is the one and only object in the resource, making it therefore the root element to which the namespace declaration is also added. In general, the serialized document may contain objects from more than one **Epackage**, in which case the root element will include the namespace declarations for them as well. We'll see how EMF uses the namespace information when deserializing XML documents, below.

The content of an **EObject**'s XML element depends on the attributes and references in its class (that is, its **EClass**). You may recall that class **USAddress** contains four simple attributes: `street`, `city`, `state`, and `zip`. If you're wondering why they didn't appear in our example file (that is, in *address.xml*), it's because they haven't been set yet. The serializer only includes attributes and references that are set (that is, for which `eIsSet()` returns `true`). This approach minimizes the size of the XML file by only including things that need to be serialized.

Multiplicity-1 Attributes

A single valued attribute is serialized as an XML attribute. For example, if we set some of the attributes of the **USAddress** in our previous example before saving it, as follows:

```
USAddress address = EPO2Factory.eINSTANCE.createUSAddress();
address.setStreet("123 Main Street");
address.setState("CA");
address.setZip(11111);
Resource resource =
    new XMLResourceImpl(URI.createURI(".../address.xml"));
resource.getContents().add(address);
resource.save(null);
```

⁴For simplicity, we're creating the resources in these examples using the Java new operator. The proper way to do it is using the resource registry and a resource set, as described in Section 13.2.5.

then the contents of *address.xml* will now look like this:

```
<com.example.epo2:USAddress
  xmlns:com.example.epo2="http://com/example/epo2.ecore"
  street="123 Main Street" state="CA" zip="11111"/>
```

As you can see, each attribute value is saved using an XML attribute. Notice that the **city** attribute, which we haven't changed from its default value, is still not included in the serialized document.

Multiplicity-Many Attributes

Each value of a multi-valued attribute is stored in a nested XML element. For example, class **GlobalAddress** has a multi-valued string attribute called **location**. Here is an example that creates a **GlobalAddress** and sets two values for the **location** attribute:

```
GlobalAddress address = EPO2Factory.eINSTANCE.createGlobalAddress();
address.getLocation().add("Timbuktu");
address.getLocation().add("Mali");
Resource resource =
  new XMLResourceImpl(URI.createURI("../global.xml"));
resource.getContents().add(address);
resource.save(null);
```

After running this, the file *global.xml* has the following contents:

```
<com.example.epo2:GlobalAddress
  xmlns:com.example.epo2="http://com/example/epo2.ecore">
<location>Timbuktu</location>
<location>Mali</location>
</com.example.epo2:GlobalAddress>
```

Notice that each of the values is serialized in the content of an XML element with the name of the attribute (that is, **location**).

Containment References

A contained object is serialized as a nested XML element of its container's element. The name of the nested XML element is that of the reference. For example, class **Supplier** includes a containment reference **customers** to class **Customer**.

Here is an example that serializes a supplier with one customer:

```
Supplier supplier = EPO2Factory.eINSTANCE.createSupplier();
supplier.setName("S1");
Customer customer = EPO2Factory.eINSTANCE.createCustomer();
customer.setCustomerID(1);
supplier.getCustomers().add(customer);
Resource resource =
  new XMLResourceImpl(URI.createURI("../supplier1.xml"));
resource.getContents().add(supplier);
resource.save(null);
```

The contents of *supplier1.xml* will include two elements, as follows:

```
<com.example.epo2:Supplier
  xmlns:com.example.epo2="http://com/example/epo2.ecore" name="S1">
<customers customerID="1"/>
</com.example.epo2:Supplier>
```

The customer has been serialized in an XML element called **customers**, which is nested in the XML element for the supplier. In this case, the class of the referenced object, **Customer**, matches the type of the **customers** reference. However, this is not always the case (see "References to Subclasses" on page 315).

Non-Containment References

EMF serializes non-containment references as attributes whose value is the URI fragment of the referenced object. Class **PurchaseOrder** has a **customer** reference and the **Customer** class has an **orders** reference. These references are not containment references. Let's create a supplier containing a cross referenced customer and purchase order, and save them:

```
Supplier supplier = EPO2Factory.eINSTANCE.createSupplier();
supplier.setName("S2");
Customer customer = EPO2Factory.eINSTANCE.createCustomer();
customer.setCustomerID(1);
supplier.getCustomers().add(customer);
PurchaseOrder order = EPO2Factory.eINSTANCE.createPurchaseOrder();
order.setComment("PO1");
order.setCustomer(customer);
supplier.getOrders().add(order);
Resource resource =
    new XMLResourceImpl(URI.createURI("..../supplier2.xml"));
resource.getContents().add(supplier);
resource.save(null);
```

The file *supplier2.xml* will have the following contents:

```
<?xml version="1.0" encoding="ASCII"?>
<com.example.epo2:Supplier
  xmlns:com.example.epo2="http://com/example/epo2.ecore"
  name="S2">
  <customers customerID="1" orders="//@orders.0"/>
  <orders comment="PO1" customer="//@customers.0"/>
</com.example.epo2:Supplier>
```

Notice the **orders** attribute of the **customers** element and the **customer** attribute of the **orders** element. Both of these references are serialized as fragment paths, as described in Section 13.2.3.

Recall from Chapter 5 that one of the **EAttributes** in an **EClass** can be designated as an **iD**. Such an attribute affects the serialization of references to objects of that class. If a referenced object's class has an **iD** attribute, then it will be referenced using its value instead of with the fragment path. For example, let's assume that the **customerID** attribute in class **Customer** had been declared to be an **iD** in the ExtendedPO2 model. If this were the case, then the **customer** attribute, above, would have serialized like this:

```
<orders comment="PO1" customer="1"/>
```

The value of the customer's **customerID** attribute would have been used for the reference.

Cross-Document References

Now let's consider the case where the reference is cross-document. The default serialization uses an XML element that has an **href** attribute whose value is a URI that identifies the object in the other resource. For example, the following saves the purchase order, from the previous example, under a different supplier in another resource:

```
Supplier supplier = EPO2Factory.eINSTANCE.createSupplier();
supplier.setName("S2");
Customer customer = EPO2Factory.eINSTANCE.createCustomer();
customer.setCustomerID(1);
supplier.getCustomers().add(customer);
Resource resource =
    new XMLResourceImpl(
        URI.createURI("platform:/resource/project/supplier2.xml"));
resource.getContents().add(supplier);

Supplier anotherSupplier = EPO2Factory.eINSTANCE.createSupplier();
```

```

anotherSupplier.setName("S3");
PurchaseOrder order = EPO2Factory.eINSTANCE.createPurchaseOrder();
order.setComment("PO1");
order.setCustomer(customer);
anotherSupplier.getOrders().add(order);
Resource resource2 =
    new XMLResourceImpl(
        URI.createURI("platform:/resource/project/supplier3.xml"));
resource2.getContents().add(anotherSupplier);

resource.save(null);
resource2.save(null);

```

The contents of *supplier2.xml* is now:

```

<?xml version="1.0" encoding="ASCII"?>
<com.example.epo2:Supplier
  xmlns:com.example.epo2="http://com/example/epo2.ecore"
  name="S2">
  <customers customerID="1">
    <orders href="supplier3.xml#//@orders.0"/>
  </customers>
</com.example.epo2:Supplier>

```

In the file *supplier2.xml*, the cross-document reference to the purchase order is now saved in an XML element called *orders* that has an *href* attribute with a value of *supplier3.xml#//orders.0*. The value is the URI of the second resource (relative to the first one) with a fragment, which is, as in the local reference case, the path to the target object in its resource.

Notice that we waited and saved both resources at the end, after all the objects had been added to their resources. If instead we had tried to save the first resource before adding the referenced object to the second resource, the serializer would not have been able to determine the value for the *href* and would have thrown an exception. This is something to keep in mind whenever saving resources with cross-document references.

A very important issue when serializing a cross-document reference is whether the *href* will be relative or absolute. Relative references are usually preferred in that they allow you to move both resources to a different absolute location, without breaking their connection. When loaded, relative URLs are resolved against the URI of the containing document, just the way it works when an HTML file is viewed by a Web browser.

In our example, the serializer did in fact produce a relative cross-document reference, *supplier3.xml#//@orders.0*. This is what we wanted, but the reason it did this may not be obvious. In the code fragment above, you may have noticed that we created the two resources, *resource* and *resource2*, using absolute URIs (that is, URIs with schemes).⁵ We did that specifically to ensure that the cross-document reference would be a relative one. The XML serializer attempts to dereference (that is, make relative) a target URI against the source document's URI, only if both URIs are absolute and hierarchical. If they are not, it will simply serialize the target's URI as is. For this reason, it's important to always use absolute URIs when creating or loading resources, if you want them to contain relative references. EMF supports the platform protocol, which we described in Section 13.2.1, both inside Eclipse and stand-alone, just for this reason.

⁵Note that a URI without a scheme/protocol is considered relative.

References to Subclasses

Recall that the target type of the `billTo` reference of class `PurchaseOrder` (in the ExtendedPO2 model) is an abstract class, `Address`. If we set a `billTo` reference to a `USAddress` (that is, a subclass of `Address`), the nested `billTo` element will include an XML attribute called `xsi:type`. The value of the `xsi:type` attribute is the namespace-qualified name of the class of the object. The following example creates a purchase order with a `USAddress` as its `billTo` address:

```
PurchaseOrder order = EPO2Factory.eINSTANCE.createPurchaseOrder();
order.setComment("Purchase order with billTo address");
USAddress address = EPO2Factory.eINSTANCE.createUSAddress();
address.setStreet("123 Main Street");
order.setBillTo(address);
Resource resource =
    new XMLResourceImpl(URI.createURI(".../billto.xml"));
resource.getContents().add(order);
resource.save(null);
```

The file `billto.xml` will contain the following:

```
<com.example.epo2:PurchaseOrder
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:com.example.epo2="http://com/example/epo2.ecore"
  comment="Purchase order with billTo address">
  <billTo xsi:type="com.example.epo2:USAddress"
    street="123 Main Street"/>
</com.example.epo2:PurchaseOrder>
```

Notice that the `billTo` XML element has an `xsi:type` attribute with value `com.example.epo2:USAddress`. The `xsi` namespace prefix corresponds to the XML Schema instance namespace defined in the XML Schema specification [8].

Deserializing (Loading) XML

Now that we've seen how the XML resource serializes various Ecore constructs, let's look at how it deserializes, or loads, an XML serialization. As you might imagine, deserialization is provided by the `load()` method of class `XMLResourceImpl`.

During load, the namespace URIs (from the namespace declarations) of the XML being loaded are used to locate the required Ecore packages. As described in Section 13.1.2, this may involve either loading serialized Ecore resources or retrieving the packages from the package registry. From the packages, the factories for creating the required EMF objects are then retrieved.

For each XML attribute and element, the loader uses the name of the attribute or element to find the corresponding construct in a core model. Once it finds the Ecore construct, it uses it to create and initialize the appropriate EMF object using the reflective `EObject` API.

For example, when an XML attribute is processed, the loader obtains the Ecore feature whose name matches that of the XML attribute. If the feature is an `EAttribute`, it sets the attribute value using the XML attribute value. If the feature is an `EReference`, it uses the XML attribute value to obtain the corresponding `EObject`. If the `EObject` has not been created yet, it stores the value until the end of the document and then attempts to obtain the `EObject` again, setting the reference value at that time.

When the default loader processes a cross-document reference, it creates a proxy for the object. This way, loading of the target document is delayed until the reference is actually accessed. EMF's lazy proxy resolution design was discussed in Chapters 2 and 9.

The loader continues working after it encounters an XML element or attribute that it cannot process correctly. This way, it loads as much of the data as possible, even if there are errors. You can look in the errors and warnings lists of the resource, after loading, to see any problems that may have occurred during a load.

The default loader uses Xerces, the Apache Software Foundation's (www.apache.org) XML parser, to perform a SAX-based load of the document, which minimizes memory usage.

Customizing XML Resources

You can customize the XML serialization by providing options to the `save()` method, instead of simply calling `resource.save(null)` as we did in the previous section. As described in Section 13.2.3, options are passed as a `Map` containing the options in the form of key-value pairs. Table 13.1 lists the options (keys) that are supported by the XML serializer, along with the types of their corresponding values.

Table 13.1. XML Save Options

<i>Option</i>	<i>Type</i>
OPTION_LINE_WIDTH	Integer
OPTION_DECLARE_XML	Boolean
OPTION_USE_ENCODED_ATTRIBUTE_STYLE	Boolean
OPTION_SKIP_ESCAPE	Boolean
OPTION_PROCESS_DANGLING_HREF	String
OPTION_SCHEMA_LOCATION	Boolean
OPTION_XML_MAP	XMLResource.XMLMap
OPTION_ENCODING	String

OPTION_LINE_WIDTH can be used to set the maximum width of a line in the XML file. You can use this option to make your XML documents more readable.

OPTION_DECLARE_XML can be set to `false`, to prevent the serializer from writing the XML declaration. However, you need to be careful when you do this, because it's illegal for the declaration to be absent if the character encoding (see below) is anything other than UTF-8. One reason you may want to suppress the declaration is if you're using the serializer to write XML in the middle of an output stream that already includes an XML declaration.

OPTION_USE_ENCODED_ATTRIBUTE_STYLE affects how EMF serializes references. If this option is set to `true`, EMF will always serialize references as URLs in attribute values, regardless of whether the references are local or cross-document. Recall from Section 13.3.1 that by default, cross-file references are serialized using an XML element with an `href` attribute, while local ones are serialized as attributes containing a fragment path. This option affects both patterns.

In the previous section, we saw that a customer's `orders` reference to a purchase order in the same resource would be serialized like this:

```
<customers customerID="1" orders="//@orders.0"/>
```

If we had set the `OPTION_USE_ENCODED_ATTRIBUTE_STYLE` option to `true`:

```
Map options = new HashMap();
options.put(XMLResource.OPTION_USE_ENCODED_ATTRIBUTE_STYLE,
           Boolean.TRUE);
resource.save(options);
```

then the same reference would have been serialized like this instead:

```
<customers customerID="1" orders="#//@orders.0"/>
```

Can you see the subtle difference? The leading `#` character indicates that we're now using a URI instead of a simple fragment path to reference the purchase order. For a cross-document reference, the difference is less subtle. Recall that the cross-document `orders` reference looked like this:

```
<customers customerID="1">
  <orders href="supplier3.xml#//@orders.0"/>
</customers>
```

With `OPTION_USE_ENCODED_ATTRIBUTE_STYLE` set to `true`, it would have looked like this:

```
<customers customerID="1" orders="supplier3.xml#//@orders.0"/>
```

Notice that with this option, the cross-document reference is now encoded in exactly the same way as the reference to an object in the same resource (that is, as an attribute).

`OPTION_SKIP_ESCAPE` can be set to `true` to tell the serializer not to encode the characters "<", ">", and "&" using predefined entities (that is, "<", ">", and "&"). You can use this option to reduce the save time, but you should only set it if you know, for sure, that your attribute values do not contain characters that need to be treated specially by XML; otherwise, you won't be able to reload the document.

`OPTION_PROCESS_DANGLING_HREF` specifies how the serializer behaves when a referenced object is not in a resource. It has one of three values: "THROW", "RECORD", or "DISCARD". If the value is "THROW" (the default), and referenced objects without resources are encountered, the serializer will discard the bad references, save the rest of the model, and then throw an exception at the end. If "RECORD", the serializer will do the same, except it will add the exception to the resource's error list instead of throwing it. If the value is "DISCARD", the serializer will again discard the bad references, but it will not inform you in any way that some references were not saved.

`OPTION_SCHEMA_LOCATION` can be set to `true` to instruct the serializer to produce an `xsi:schemaLocation` attribute in the document. This attribute will then be used, by the loader, to locate the persisted resources containing the serialized **EPackages** needed to load the document. This is an alternative to the package registry approach, which we described in Section 13.1.2. We'll describe this approach in more detail in Section 13.6.

`OPTION_XML_MAP` is used to specify the mapping from Ecore to XML. An `XMLMap` provides a mapping from Ecore constructs to `XMLInfo` objects that contain information about how to represent the element in XML. It also provides other capabilities that help when loading an XML document, which we discuss below. You can use an `XMLMap` to change the name of an XML element or attribute to something different from the name specified in a core model, or to change the way that EMF attribute values are serialized.

For example, the `comment` attribute in class `PurchaseOrder` is single-valued and, as we saw in the previous section, is normally serialized as an XML attribute. Let's use an `XMLMap` to save it instead, as an XML element and with a different name, `description`:

```
PurchaseOrder order = EPO2Factory.eINSTANCE.createPurchaseOrder();
order.setComment("Purchase order description");
Resource resource =
  new XMLResourceImpl(URI.createURI("../description.xml"));
resource.getContents().add(order);

XMLResource.XMLInfo info = new XMLInfoImpl();
info.setXMLRepresentation(XMLResource.XMLInfo.ELEMENT);
info.setName("description");
XMLResource.XMLMap xmlMap = new XMLMapImpl();
xmlMap.add(EPO2Package.eINSTANCE.getPurchaseOrder_Comment(), info);

Map options = new HashMap();
options.put(XMLResource.OPTION_XML_MAP, xmlMap);
resource.save(options);
```

As you can see, we created an `XMLMap` and added an entry for the `comment` attribute, which we accessed by calling `getPurchaseOrder_Comment()`. We've set the value to an `XMLInfo` object with XML representation set to `ELEMENT`, to specify that the attribute should be serialized using an XML element instead of an attribute. By setting the name of the `XMLInfo` object, we're also specifying that the name of the element should be `description` instead of the default, `comment`.

If we run this program, the file `description.xml` will contain the following:

```
<?xml version="1.0" encoding="ASCII"?>
<com.example.epo2:PurchaseOrder
    xmlns:com.example.epo2="http://com/example/epo2.ecore">
    <description>Purchase order description</description>
</com.example.epo2:PurchaseOrder>
```

As you can see, the value of the `comment` attribute has now been serialized in an XML element named `description`.

OPTION_ENCODING can be used to change the character encoding that is used in the serialized XML. As an alternative to using this save option, you can also change the encoding by calling the `setEncoding()` method on the `XMLResource` interface. The default XML encoding is "ASCII."

Now that we've covered the customizations that can be made when saving an XML resource, let's look at the options that can be provided to the `load()` method. Table 13.2 lists the options that are supported by the XML loader. As you can see, there are only two load options, the second of which is the same as one of the save options.

Table 13.2. XML Load Options

Option	Type
<code>OPTION_DISABLE_NOTIFY</code>	Boolean
<code>OPTION_XML_MAP</code>	<code>XMLResource.XMLMap</code>

OPTION_DISABLE_NOTIFY can be used to disable notifications during a load. If you've modified your factory or constructor to add adapters to your objects immediately on creation, notification to those adapters can significantly increase the time it takes to load a document. This option can be used to reduce the load time in this situation. Notifications will be re-enabled at the end of the load operation.

OPTION_XML_MAP is the only option that is both a save option and a load option. If you use the `OPTION_XML_MAP` option during save, you must use the option, with the same `XMLMap`, during load in order for the load to succeed. (All of the other save options require nothing special during load; the default XML loader can handle XML documents saved with any of the save options.)

There are several reasons why you might want to use `OPTION_XML_MAP` when saving a resource:

1. You want to save your data so that it conforms to a particular XML Schema.
2. You have changed your model, and you want the resource to produce XML documents that will work with the old version of the model.
3. You want to design your own XML format without changing your model.

In addition to customizing the Ecore to XML mapping during save and load, the `OPTION_XML_MAP` option can be used to identify packages for which no XML namespace is provided. We described, above, how the XML loader attempts to get a package with a namespace URI that matches that of an XML namespace in a document. We recommend that you use XML namespaces so that it is possible to save objects from more than one model in a single XML file unambiguously. However, an `XMLMap` can specify a default package to be used for XML elements that are not in a namespace.

Let's say that you want to load a **USAddress** that has been serialized without a namespace in file *nonamespace.xml* as follows:

```
<USAddress street="123 Main Street" city="SomeCity"
state="CA" zip="11111"/>
```

You can load this file with the help of an **XMLMap**, like this:

```
Resource resource =
new XMLResourceImpl(URI.createURI(".../nonamespace.xml"));
XMLResource.XMLMap xmlMap = new XMLMapImpl();
xmlMap.setNoNamespacePackage(EPO2Package.eINSTANCE);
Map options = new HashMap();
options.put(XMLResource.OPTION_XML_MAP, xmlMap);
resource.load(options);
```

Notice that the package to use is set by calling the `setNoNamespacePackage()` method, as opposed to adding anything to the map.

As we've seen, you can easily customize the default XML resource implementation a great deal by simply using its options. More significant specializations are also possible by subclassing from `XMLResourceImpl` (or `XMIResourceImpl`) and overriding the serialization code itself. If you plan to do this, you'll see that the default implementation delegates most of the serialization implementation to the framework classes `XMLSaveImpl` and `XMLHelperImpl`, while the loading implementation is (also) delegated to `XMLHelperImpl`, as well as class `XMLLoaderImpl`. You'll likely want to override some, or all, of these implementation classes as well.

Working with XML Schemas

Using the various save and load options described in the previous section, you could customize the format of an XML file to validate against an XML Schema. You don't need to do this yourself, however, if you created your core model from an XML Schema. In Chapters 2 and 7 we explained how a core model can be defined using an XML Schema, in which case the default serialization format will automatically be customized to conform to the schema.

Recall from Chapter 7 that when a core model is defined using an XML Schema, the schema importer adds **EAnnotations** to the model elements that it creates, which will later be used to control the serialization of the model. You may be wondering how the annotation-based customization relates to the `XMLMap` mechanism, described in the previous section.

The answer to this question can be seen by looking in the generated `util` package. For example, if we generate a model imported from the simple purchase order schema, *SimplePO.xsd*, which was described back in Chapter 2, we'd see, in addition to the usual adapter factory and switch classes, a resource factory class as well:

```
public class POResourceFactoryImpl extends XMLResourceFactoryImpl
{
    protected XMLResource.XMLMap xmlMap = new XMLMapImpl();

    public POResourceFactoryImpl() {
        super();
    }
    public Resource createResource(URI uri) {
        XMLResource result = new POResourceImpl(uri);
        result.getDefaultSaveOptions().put(XMLResource.OPTION_XML_MAP,
                                           xmlMap);
        result.getDefaultLoadOptions().put(XMLResource.OPTION_XML_MAP,
                                           xmlMap);
        return result;
    }
}
```

As you can see, the `createResource()` method creates a resource (`POResourceImpl`, which is a trivial subclass of `XMLResourceImpl`) as usual, but then it sets the `OPTION_XML_MAP` save and load option to an empty `XMLMap`. How is this useful?

The answer can be found in the `getInfo()` method of the `XMLMap` implementation class, `XMLMapImpl`. If an Ecore construct being looked up in the map does not have an `XMLInfo` object associated with it, then the method checks the `eAnnotations` of the Ecore construct. If it finds XSD2Ecore-type annotation (that is, one with the `source` attribute equal to “`http://org/eclipse/emf/mapping/xsd2ecore/XSD2Ecore`”), then it creates an `XMLInfo` object from the annotation, and returns it. So, although the `XMLMap` starts out empty, because the core model includes annotations, the `getInfo()` method will populate the map with `XMLInfo` objects on demand.

To summarize, you can customize the behavior of class `XMLResourceImpl` using `OPTION_XML_MAP` in either of two ways. First, as we described in the previous section, you can pass it an `XMLMap` populated with the set of `XMLInfo` objects that completely define the mapping. The alternative, which is used by default for XML Schema defined models, is to use an empty `XMLMap` and provide the mapping information using annotations on the core model. In this case, the `XMLMap` will create the `XMLInfo` objects itself.

XMI Resources

So far, we've been looking at the output produced by the XML resource implementation class, `XMLResourceImpl`. You may recall, however, that the default serialization format provided by EMF is XMI (XML Metadata Interchange), not *just* ordinary XML. What's the difference, you may ask? Actually, not a lot.

The default XMI resource implementation class, `XMIResourceImpl`, is really just a simple subclass of the `XMLResourceImpl`, which provides customizations and extensions needed to support the XMI 2.0 format.⁶ Most importantly, it adds support for referencing objects using XMI IDs.

Let's modify our `supplier2.xml` example from Section 13.3.1 to use an `XMIResourceImpl`, instead of `XMLResourceImpl` used previously:

```
Supplier supplier = EPO2Factory.eINSTANCE.createSupplier();
supplier.setName("S2");
Customer customer = EPO2Factory.eINSTANCE.createCustomer();
customer.setCustomerID(1);
supplier.getCustomers().add(customer);
PurchaseOrder order = EPO2Factory.eINSTANCE.createPurchaseOrder();
order.setComment("PO1");
order.setCustomer(customer);
supplier.getOrders().add(order);
Resource resource =
    new XMIResourceImpl(URI.createURI(".../supplier2.xmi"));
resource.getContents().add(supplier);
resource.save(null);
```

The serialized file, `supplier2.xmi`, will have the following content:

```
<?xml version="1.0" encoding="ASCII"?>
<com.example.epo2:Supplier
  xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:com.example.epo2="http://com/example/epo2.ecore"
```

⁶For example, when referencing subclasses, as we described in Section 13.3.1, it uses the standard `xmi:type` attribute, instead of `xsi:type`, to specify the target's class.

```

    name="S2">
<customers customerID="1" orders="//@orders.0"/>
<orders comment="PO1" customer="//@customers.0"/>
</com.example.epo2:Supplier>
```

If you refer back to the output we got when we used `XMLResourceImpl`, you'll notice that the only differences are the two highlighted lines; the `xmi:version` attribute indicates that this is now an XMI 2.0 file, while the `xmlns:xmi` attribute declares the XMI namespace.

Using `XMIResourceImpl` we can, however, change the serialization of the two references, `orders` and `customer`, to use XMI IDs rather than paths. We can do this by providing IDs for the referenced objects before saving them. To do this, we use a map of objects to IDs, provided by class `XMIResourceImpl`. For our example, we could provide IDs for the customer and purchase order like this:

```

Supplier supplier = EPO2Factory.eINSTANCE.createSupplier();
supplier.setName("S2");
Customer customer = EPO2Factory.eINSTANCE.createCustomer();
customer.setCustomerID(1);
supplier.getCustomers().add(customer);
PurchaseOrder order = EPO2Factory.eINSTANCE.createPurchaseOrder();
order.setComment("PO1");
order.setCustomer(customer);
supplier.getOrders().add(order);
XMIResource resource =
  new XMIResourceImpl(URI.createURI("../supplier2id.xmi"));
resource.setID(customer, "Customer_1");
resource.setID(order, "Order_1");
resource.getContents().add(supplier);
resource.save(null);
```

The `setID()` method assigns the specified ID to the specified object. References to objects with such IDs will then use it, instead of a path. The file `supplier2id.xmi` will contain the following:

```

<?xml version="1.0" encoding="ASCII"?>
<com.example.epo2:Supplier
  xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:com.example.epo2="http://com/example/epo2.ecore"
  name="S2">
<customers xmi:id="Customer_1" customerID="1" orders="Order_1"/>
<orders xmi:id="Order_1" comment="PO1" customer="Customer_1"/>
</com.example.epo2:Supplier>
```

As you can see, the IDs are serialized in `xmi:id` attributes. The `orders` and `customer` XML attributes now contain the IDs, instead of paths. Note that the XMI Resource can load files with either type of reference, or even a mixture of the two.

Adapters

EMF adapters are what other programming frameworks call observers. We call them adapters because, in addition to merely observing changes, they also allow you to extend the behavior of model objects without subclassing from them.

Object Adapting

Let's consider a trivial adapter whose job is to keep track of the number of changes being made to instances of classes in the purchase order model. To be specific, we're going to define three static counters to keep track of changes to instances of class **PurchaseOrder**, **Item**, and **Address**, and then use adapters to increment the appropriate counter in response to change notifications from objects in the model. We're going to look at three different ways of doing this:

1. Attaching an adapter to each object in the model by simply adding it to their `eAdapters` lists
2. Using a simple adapter factory `adapt()` call to attach the same adapter to the model objects
3. Using the generated purchase order adapter factory to attach type-specific adapters to the model objects

Although this example is quite trivial, it should give you a sense of how these three approaches compare, and why you might choose one over the other.

For the first approach, we'll start by defining a generic adapter class, like this:

```
public class ChangeCounterAdapter extends AdapterImpl
{
    public static int purchaseOrderCount;
    public static int itemCount;
    public static int addressCount;

    public void notifyChanged(Notification notification) {
        if (notification.getNotifier() instanceof PurchaseOrder)
            ++purchaseOrderCount;
        else if (notification.getNotifier() instanceof Item)
            ++itemCount;
        else if (notification.getNotifier() instanceof Address)
            ++addressCount;
    }

    public static ChangeCounterAdapter
        INSTANCE = new ChangeCounterAdapter();
}
```

Here we've created a simple subclass of the framework's adapter implementation base class `AdapterImpl`. We've implemented the `notifyChanged()` method to check what type of object is sending the notification and then, based on that, increment the appropriate one of three static counters. We've also created a static singleton instance of the adapter; since it carries no state, we can use a single instance for all the objects we plan to adapt.

Notice that we use the `getNotifier()` method, on the `notification` argument, to access the model object. In addition to `getNotifier()`, class `Notification` includes methods for retrieving the notification (event) type, the feature (or just the feature's ID) that has changed, old and new values of the changed feature, among other things. For example, if we wanted to ignore "touch" notifications (that is, non-state-changing events), we could have used the `isTouch()` method, like this:

```
public void notifyChanged(Notification notification) {
    if (notification.isTouch()) return;
    if (notification.getNotifier() instanceof PurchaseOrder)
        ...
}
```

Now that we have the adapter class, we need to add it to instances of the purchase order model. Here's a simple program that does that:

```

USAddress address = EPO2Factory.eINSTANCE.createUSAddress();
address.eAdapters().add(ChangeCounterAdapter.INSTANCE);
address.setName("123 Maple Street");

Item item = EPO2Factory.eINSTANCE.createItem();
item.eAdapters().add(ChangeCounterAdapter.INSTANCE);
item.setProductName("Apples");
item.setQuantity(20);

PurchaseOrder purchaseOrder =
    EPO2Factory.eINSTANCE.createPurchaseOrder();
purchaseOrder.eAdapters().add(ChangeCounterAdapter.INSTANCE);
purchaseOrder.setBillTo(address);
purchaseOrder.getItems().add(item);

System.out.println("PurchaseOrder changes: " +
    ChangeCounterAdapter.purchaseOrderCount);
System.out.println("Item changes: " +
    ChangeCounterAdapter.itemCount);
System.out.println("Address changes: " +
    ChangeCounterAdapter.addressCount);

```

As you can see, we're using the adapter to compute the changes that occur while constructing a model consisting of a single purchase order, item, and address. As we said it would, our first approach simply attaches the adapter by adding to the `eAdapters` list; that is, by calling `eAdapters().add()`. We attach the adapter singleton, `INSTANCE`, to each object in the model immediately after it's created.

If we run this program, then each time an attribute or reference of any of the three created (and adapted) objects is set, the adapter will receive a change notification and increment the appropriate counter. In case you're interested, the program will produce the following output:

```

PurchaseOrder changes: 2
Item changes: 3
Address changes: 1

```

Now let's move on to our second approach of using an adapter factory instead of simply adding to the `eAdapters` lists. For this approach, we can use the same adapter class, `ChangeCounterAdapter`, but we'll need to add one more method to it:

```

public class ChangeCounterAdapter extends AdapterImpl
{
    public static int purchaseOrderCount;
    public static int itemCount;
    public static int addressCount;

    public void notifyChanged(Notification notification) {
        if (notification.getNotifier() instanceof PurchaseOrder)
            ++purchaseOrderCount;
        else if (notification.getNotifier() instanceof Item)
            ++itemCount;
        else if (notification.getNotifier() instanceof Address)
            ++addressCount;
    }

    public boolean isAdapterForType(Object type) {
        return type == ChangeCounterAdapter.class;
    }

    public static ChangeCounterAdapter
        INSTANCE = new ChangeCounterAdapter();
}

```

The `isAdapterForType()` method will be used by adapter factories when checking if an adapter of a specified type is already attached to an object. Remember that adapter factories are used to attach adapters, which usually represent some kind, or “type,” of behavioral extensions to the object. In our example, we’re using the class `ChangeCounterAdapter.class` to represent the adapter’s type.⁷

Now that the adapter knows its type, we need an adapter factory capable of adapting objects with adapters of this type. We can create such an adapter factory like this:

```
public class ChangeCounterAdapterFactory extends AdapterFactoryImpl
{
    public boolean isFactoryForType(Object type) {
        return type == ChangeCounterAdapter.class;
    }

    protected Adapter createAdapter(Notifier target) {
        return ChangeCounterAdapter.INSTANCE;
    }

    public static ChangeCounterAdapterFactory
        INSTANCE = new ChangeCounterAdapterFactory();
}
```

By subclassing the framework base class `AdapterFactoryImpl`, all we need to implement are two methods: `isFactoryForType()` and `createAdapter()`. In our example, we’re only using this adapter factory for our single adapter type, so `isFactoryForType()` has the same implementation as the adapter’s `isAdapterForType()` method; it returns `true` only for the type `ChangeCounterAdapter.class`. For the same reason, `createAdapter()` simply returns an instance of `ChangeCounterAdapter`. Notice that, as in the previous example, we’re simply using the singleton adapter, `ChangeCounterAdapter.INSTANCE`, for every object we adapt.

Using the adapter factory approach, we would now change the test program to attach the adapters like this:

```
USAddress address = EPO2Factory.eINSTANCE.createUSAddress();
ChangeCounterAdapterFactory.INSTANCE.adapt(address,
                                             ChangeCounterAdapter.class);
address.setName("123 Maple Street");

Item item = EPO2Factory.eINSTANCE.createItem();
ChangeCounterAdapterFactory.INSTANCE.adapt(item,
                                             ChangeCounterAdapter.class);
item.setProductName("Apples");
item.setQuantity(20);

PurchaseOrder purchaseOrder =
    EPO2Factory.eINSTANCE.createPurchaseOrder();
ChangeCounterAdapterFactory.INSTANCE.adapt(purchaseOrder,
                                             ChangeCounterAdapter.class);
purchaseOrder.setBillTo(address);
purchaseOrder.getItems().add(item);

System.out.println(...)
```

Running this program will produce the exact same output as before. The only difference in doing it this way is that we need to specify the additional “type” argument to the `adapt()` method that we didn’t need when manipulating the `eAdapters` list directly. In our simple example, this approach is really of no added value since we’re only adding a single adapter to each object, and never accessing them again. If, however, we wanted to pass these (adapted) objects around and later retrieve the

⁷Notice that you can use anything you want as an adapter type; it’s represented by a `java.lang.Object`. The two most commonly used types are `java.lang.Class` and `java.lang.String`.

adapter, with this design, we could simply call `adapt(..., ChangeCounterAdapter.class)` again to do that. Instead of adding another adapter to the `eAdapters` list, the adapter factory would then simply return the existing one. With the other approach of using `eAdapters().add()` to add the adapter, we would need to iterate through the list to find the existing adapter.⁸

Now that we're using an adapter factory, let's move on to our third, and final, approach: using type-specific adapters. To do this we'll create another adapter factory, but this time it will subclass the generated `EPO2AdapterFactory`, like this:

```

public class TypedChangeCounterAdapterFactory
    extends EPO2AdapterFactory
{
    public Adapter createPurchaseOrderAdapter() {
        return new ChangeCounterAdapter()
    {
        public void notifyChanged(Notification notification) {
            ++purchaseOrderCount;
        }
    };
}

public Adapter createItemAdapter() {
    return new ChangeCounterAdapter()
    {
        public void notifyChanged(Notification notification) {
            ++itemCount;
        }
    };
}

public Adapter createAddressAdapter() {
    return new ChangeCounterAdapter()
    {
        public void notifyChanged(Notification notification) {
            ++addressCount;
        }
    };
}

public boolean isFactoryForType(Object type) {
    return type == ChangeCounterAdapter.class ||
           super.isFactoryForType(type);
}

public static TypedChangeCounterAdapterFactory
    INSTANCE = new TypedChangeCounterAdapterFactory();
}
}

```

The primary difference from the previous implementation is that instead of implementing the single `createAdapter()` method, here we implement specific create methods for each type of adapter. You may recall from Chapter 9 that a generated adapter factory, like our base class `EPO2AdapterFactory`, implements the `createAdapter()` method by delegating to type-specific `create()` methods, which by default return `null`. A derived class, like ours, can then simply override the appropriate methods to return type-specific adapters. Notice that in our case, we simply use

⁸If we had included the `isAdapterForType()` method in our initial adapter implementation, we could use the `EcoreUtil.getExistingAdapter()` convenience method to find the adapter, even if we had added it without using an adapter factory. The adapter factory, however, gives us the added benefit of on-demand adapter creation, which is often desirable.

anonymous subclasses of the generic `ChangeCounterAdapter`, each of which overrides the `notifyChanged()` method to increment the appropriate counter. This clearly illustrates the benefit of the typed adapter approach: no `instanceof` checking. We now have a pure object-oriented implementation.

As you might expect, we'd use this factory in exactly the same way as the previous example, only using `TypedChangeCounterAdapterFactory.INSTANCE` for the adapter factory:

```
USAddress address = EPO2Factory.eINSTANCE.createUSAddress();
TypedChangeCounterAdapterFactory.INSTANCE.adapt(address,
                                                ChangeCounterAdapter.class);
address.setName("123 Maple Street");

// etc ...
```

One thing that you may have noticed in class `TypedChangeCounterAdapterFactory` is that unlike the previous two approaches where we used a singleton adapter for all the objects, here we simply new up an adapter every time one of the `create()` methods is called. Actually, for a stateless adapter like this, we would normally prefer to create three singleton adapters, one for each class, and simply return them in each of the three `create()` methods. However, for our trivial program we only need three adapters in total (that is, we only have one of each type of model object in our example program), so it wouldn't have made any difference anyway.

If you've been paying close attention, you may have noticed that our test program creates an instance of class `USAddress`, but in the adapter factory we've implemented the `createAddressAdapter()` method, not `createUSAddressAdapter()`. This is a very important property of the generated adapter factory class (also of the generated switch class, as we'll see in Section 13.5.2). The generated adapter factory class includes `create()` methods for every class in the model package, even abstract base classes. It implements `createAdapter()` by calling the `create()` method corresponding to the instance's class first, but if that returns `null`, which is the default, it will then walk up the superclass chain, trying the `create()` methods for the base classes. This design gives you the option of creating type-specific adapters for a base-class instead of one for every concrete class in the model. In our example, we created a single adapter class to handle all `Address` instances, instead of having one for class `USAddress` and a different one for class `GlobalAddress`.

Behavioral Extensions

So now that we've looked at three ways of adding an adapter, we can see how they compare. The first approach was, in fact, the simplest for our trivial example of adding a single observer to an object. This approach, however, would not be particularly useful if we wanted our adapter to actually provide some kind of extended behavior to the object, instead of simply counting change events. Let's consider a slightly different example, which does involve a behavioral extension.

Let's assume that instead of having static counters to keep track of changes to "types of objects," we instead want to keep track of the changes at the instance level. Imagine as well that we want to "extend" the interface of the objects in the purchase order model (without subclassing) with the following methods:

```
public interface InstanceChangeCounter
{
    public int getCount();
    public boolean isMoreActive(EObject otherObject);
}
```

The method `getCount()` will return the number of changes that have occurred in the object, while `isMoreActive()` will return `true` if the object has changed more than the specified other object.

Here's an adapter and corresponding adapter factory that implement this extension:

```

public class InstanceChangeCounterAdapter extends AdapterImpl
    implements InstanceChangeCounter
{
    protected int count;

    public int getCount() {
        return count;
    }

    public boolean isMoreActive(EObject otherObject) {
        InstanceChangeCounter otherAdapter =
            (InstanceChangeCounter)
                InstanceChangeCounterAdapterFactory.INSTANCE.adapt(
                    otherObject, InstanceChangeCounter.class);
        return otherAdapter == null || otherAdapter.getCount() < count;
    }

    public void notifyChanged(Notification notification) {
        ++count;
    }

    public boolean isAdapterForType(Object type) {
        return type == InstanceChangeCounter.class;
    }
}

public class InstanceChangeCounterAdapterFactory
    extends AdapterFactoryImpl
{
    public boolean isFactoryForType(Object type) {
        return type == InstanceChangeCounter.class;
    }

    protected Adapter createAdapter(Notifier target) {
        return new InstanceChangeCounterAdapter();
    }

    public static InstanceChangeCounterAdapterFactory
        INSTANCE =
            new InstanceChangeCounterAdapterFactory();
}

```

As you can see, the adapter factory looks the same as class `ChangeCounterAdapterFactory` from the previous example, only it creates a new instance of the adapter each time; we can't use a singleton now that we have a state variable, `count`, in the adapter.

As for the adapter class itself, it has a fairly straightforward implementation. The `isMoreActive()` method is the most interesting. Notice how it uses the adapter factory to access the extended interface, `InstanceChangeCounter`, of the passed in object, `otherObject`, in order to do its comparison; it is, itself, acting as a client of the extended behavior in its implementation.

To test this extension, we can use the same test model as in the previous example:

```

USAddress address = EPO2Factory.eINSTANCE.createUSAddress();
InstanceChangeCounterAdapterFactory.INSTANCE.adapt(address,
    InstanceChangeCounter.class);
address.setName("123 Maple Street");

Item item = EPO2Factory.eINSTANCE.createItem();
InstanceChangeCounterAdapterFactory.INSTANCE.adapt(item,
    InstanceChangeCounter.class);
item.setProductName("Apples");
item.setQuantity(20);

PurchaseOrder purchaseOrder =
    EPO2Factory.eINSTANCE.createPurchaseOrder();

```

```

InstanceChangeCounterAdapterFactory.INSTANCE.adapt(purchaseOrder,
                                                InstanceChangeCounter.class);
purchaseOrder.setBillTo(address);
purchaseOrder.getItems().add(item);

InstanceChangeCounter poChangeCounter =
(InstanceChangeCounter)
    InstanceChangeCounterAdapterFactory.INSTANCE.adapt(
        purchaseOrder, InstanceChangeCounter.class);
System.out.println("purchase order changes: " +
                    poChangeCounter.getCount());
System.out.println(" more than item: " +
                    poChangeCounter.isMoreActive(item));
System.out.println(" more than address: " +
                    poChangeCounter.isMoreActive(address));

```

If we run this program, it will produce the following output:

```

purchase order changes: 2
more than item: false
more than address: true

```

All of the examples that we've looked at so far needed the adapters to be created at a specific time (immediately after creation), so that they would start counting events at a known point in time. In general, however, adapters are often lazily created when they're first needed. We've already mentioned that this is one of the advantages of using an adapter factory to access adapters. The first client that needs to access an adapter implementing some extended interface will also create the adapter (transparently) when calling `adapt()` on the adapter factory.

Before a client can adapt an object, it needs to have access to an adapter factory of the appropriate type, but sometimes this is not possible. To help with this situation, EMF resource sets maintain a list of registered adapter factories that can be used to adapt objects in any resource of the set. An adapter factory can be registered with a resource set using the `getAdapterFactories()` method on the `ResourceSet` interface:

```

ResourceSet resourceSet = ...
resourceSet.getAdapterFactories().
    add(InstanceChangeCounterAdapterFactory.INSTANCE);

```

The `EcoreUtil.getRegisteredAdapter()` method can then be used to adapt objects using the registered adapter factory, like this:

```

EcoreUtil.getRegisteredAdapter(purchaseOrder,
                               InstanceChangeCounter.class);

```

The `getRegisteredAdapter()` method, after first checking if the supplied object already has an adapter of the requested type, will navigate from the object to its resource and then its resource set, from which it will retrieve the list of registered adapter factories. It then iterates through the registered adapter factories, searching for one where `isFactoryForType()` returns `true`. If it finds one, it calls `adapt()` on that factory to create the adapter.

As you can see, with this approach, all the objects that we want to adapt must be in a resource; you cannot add adapters to an object without first putting it into a resource of the set in which the adapter factory is registered. Another disadvantage of this approach is that it is less flexible; we can only have one resource factory for any given type of adapter, which all clients must use.

Observing Generated Classes

When adding adapters to generated classes, there are two optimizations that you should try to take advantage of. The first has to do with determining what feature has changed, the second with accessing the old and new values of a feature with a primitive type.

Let's assume that we've created an adapter for observing instances of class `PurchaseOrder`. Assume as well that in our observer, we need to figure out what feature has changed before handling the event. To do this, we could write something like this:

```
public void notifyChanged(Notification notification) {
    EStructuralFeature feature = notification.getFeature();
    if (feature == EPO2Package.eINSTANCE.getPurchaseOrder__Comment())
        // do something ...
    else if (feature ==
        EPO2Package.eINSTANCE.getPurchaseOrder__OrderDate())
        // do something ...
    else ...
}
```

Although this will work, it's not the most efficient way to do it. You may recall from Chapter 9 that generated `set()` methods use feature IDs, as opposed to actual features, when constructing the notification object. As a result, the feature itself is never actually accessed unless, like we just did, the `getFeature()` method is called (that is, it's retrieved lazily).

A better alternative is to use the feature ID itself to determine the notification event:

```
public void notifyChanged(Notification notification) {
    switch (notification.getFeatureID(PurchaseOrder.class)) {
        case EPO2Package.PURCHASE_ORDER__COMMENT:
            // do something ...
        case EPO2Package.PURCHASE_ORDER__ORDER_DATE:
            // do something ...
        case ...
    }
}
```

This approach has the visible benefit of allowing you to replace the (possibly lengthy) string of `if...else if... clauses` with a single, efficient `switch` statement. The other hidden benefit of not calling the `getFeature()` method is that we will avoid looking up the actual structural feature entirely. The `getFeatureID()` method simply returns the feature ID that the notification was constructed with—possibly corrected, in the case of multiple inheritance—for the expected class supplied to the method (`PurchaseOrder.class` in this case). We described in Chapter 9 how this correction, if necessary, is also done quite efficiently.

Now let's look at the second opportunity for optimization available when observing an attribute of primitive type. Let's imagine that we're observing changes to the `zip` attribute of class `USAddress`, and that we're interested in comparing the old and new values whenever it changes. We could do this as follows:

```
public void notifyChanged(Notification notification) {
    switch (notification.getFeatureID(USAddress.class)) {
        case EPO2Package.US_ADDRESS__ZIP:
            if (((Integer)notification.getOldValue()).intValue() >
                ((Integer)notification.getNewValue()).intValue())
                // do something ...
    }
}
```

However, for features of primitive type, the `Notification` interface provides a set of convenient type-specific value accessors, which we can use instead. In our case, we can access the `zip` attribute's old and new `int` values like this:

```

public void notifyChanged(Notification notification) {
    switch (notification.getFeatureID(USAddress.class)) {
        case EPO2Package.US_ADDRESS__ZIP:
            if (notification.getOldIntValue() >
                notification.getNewIntValue())
                // do something ...
    }
}

```

In addition to the convenience aspect, this approach also has a significant performance implication if the class we're observing is a generated one. When a generated `set()` method for a primitive-typed attribute constructs its notification, it passes the old and new values as primitives. If we later access them using the appropriate type-specific `getValue()` method (`getOldIntValue()` for example), the notification will simply return the value that it was constructed with. If, instead, we call the generic `getValue()` method (`getOldValue()` for example), then the notification will need to first construct an appropriate Java wrapper object (`java.lang.Integer` in this case) before returning it. By using the type-specific `getValue()` method, we will completely avoid this extra overhead.

Working with EMF Objects

EMF provides an efficient generic API that allows you to work with packages, factories, and objects, regardless of the model that defines them. The EMF framework uses these APIs to implement the XML resource load and save operations described in Section 13.3, as well as to provide several utility functions that you can use.

In this section, we'll look at some examples that show how to use the generic APIs and some of the EMF utilities to perform various tasks.

Interrogating EObjects

The reflective `EObject` API allows us to access the attributes and references of any object, regardless of its type. It also provides access to the type itself, using the `eClass()` method. Combined, this gives us everything we need to generically interrogate the contents of any arbitrary `EObject`. For example, here's a simple generic method that will print the values of all the attributes of an object:

```

public static void printAttributeValues(EObject object) {
    EClass eClass = object.eClass();
    System.out.println(eClass.getName());
    for (Iterator iter =
        eClass.getEAllAttributes().iterator(); iter.hasNext(); ) {
        EAttribute attribute = (EAttribute)iter.next();
        Object value = object.eGet(attribute);

        System.out.print(" " + attribute.getName() + " : ");
        if (object.eIsSet(attribute))
            System.out.println(value);
        else
            System.out.println(value + " (default)");
    }
}

```

We start by accessing the `EClass` of the object we wish to interrogate. Notice that to retrieve the class' attributes, we use the `getEAllAttributes()` method, which returns *all* of the attributes, including the ones defined in base classes. For each attribute, we then call the reflective `eGet()` method to retrieve its value. We also check to see if it has been set by calling `eIsSet()`. If not, we append the extra informational string "(default)" to the printed value.

If we call this method using the following object:

```
Item item = EPO2Factory.eINSTANCE.createItem();
item.setProductName("Tires");
item.setUSPrice(50);
item.setQuantity(4);
item.setPartNum("ABC-1234");
```

then the output will be as follows:

```
Item
productName : Tires
quantity : 4
USPrice : 50
comment : null (default)
shipDate : null (default)
partNum : ABC-1234
```

TreeIterator and Switch as a Visitor

In Chapters 2 and 9 we've seen that EMF can optionally generate an adapter factory and corresponding switch class for a model package. We described how the adapter factory uses the switch class in its implementation to dispatch to an appropriate `create()` method based on the type of object being adapted. You may be wondering what else the switch class is good for. Let's look at a simple example.

In this example, we're going to use the generated `EPO2Switch` class to implement a simple Visitor-like [5] pattern for visiting a purchase order model instance. We'll start by creating an anonymous visitor class, like this:

```
EPO2Switch visitor = new EPO2Switch()
{
    public Object caseItem(Item object) {
        System.out.println("visiting Item: " + object.getProductName());
        return object;
    }
    public Object caseUSAddress(USAddress object) {
        System.out.println("visiting USAddress: " + object.getName());
        return object;
    }
    public Object casePurchaseOrder(PurchaseOrder object) {
        System.out.println("visiting PurchaseOrder: " +
                           object.getComment());
        return object;
    }
    public Object caseSupplier(Supplier object) {
        System.out.println("visiting Supplier: " + object.getName());
        return object;
    }
    public Object caseCustomer(Customer object) {
        System.out.println("visiting Customer: " +
                           object.getCustomerID());
        return object;
    }
    public Object caseGlobalAddress(GlobalAddress object) {
        System.out.println("visiting GlobalAddress: " +
                           object.getName());
        return object;
    }
    public Object defaultCase(EObject object) {
        System.out.println("visiting Unknown object: " + object);
        return object;
    }
};
```

Here we've created a subclass of `EPO2Switch` and implemented each `case()` method to simply print a string indicating that it has been visited. Notice that all of the methods return their supplied argument, `object`. Actually, the exact value being returned from these methods is irrelevant in our case, as long as it's not `null`. We explained in Section 13.4.1 that a generated adapter factory will call the `create()` method of an instance's base classes if the subclass' `create()` method is not overridden to return an adapter (that is, it returns `null`). In reality, it's the generated switch class that is causing it to work that way. The implementation of the generated `doSwitch()` method, which is used to invoke the switch, will call base class `case()` methods if a subclass returns `null`. So, for our visitor example, we need to return some (any) non-`null` value to indicate that the case has been handled.

Now that we've created a visitor class, we need a model to visit. Let's assume we have a purchase order instance document, `My.epo2`, that contains the following:

```
<com.example.epo2:Supplier xmi:version="2.0" ... name="S1">
  <customers orders="//@orders.0 //@orders.1"/>
  <customers customerID="1"/>
  <customers customerID="2" orders="//@orders.2"/>
  <orders comment="P0" orderDate="2003.04.01"
    customer="//@customers.0">
    <items productName="I2" quantity="5" USPrice="10"/>
    <items productName="I3" quantity="6" USPrice="20"/>
    <items productName="I4" quantity="7" USPrice="30"/>
  </orders>
  <orders comment="P1" customer="//@customers.0">
    <items productName="I1" quantity="1" USPrice="5"/>
  </orders>
  <orders comment="P2" customer="//@customers.2">
    <items productName="I1" quantity="2" USPrice="5"/>
  </orders>
</com.example.epo2:Supplier>
```

If we load this document, we can then use the `EcoreUtil.getAllContents()` method to walk the containment hierarchy to visit the model.⁹ The `getAllContents()` method returns a `TreeIterator`, an EMF subclass of `java.util.Iterator` that adds a `prune()` method to the interface. Calling `prune()`, while iterating, will cause the following `next()` call to skip over children of the current node, if there are any. For our example, we will simply treat the tree iterator like an ordinary `java.util.Iterator`, and walk the entire structure:

```
ResourceSet resourceSet = new ResourceSetImpl();
URI fileURI = URI.createURI("platform:/resource/project/My.epo2");
Resource resource = resourceSet.getResource(fileURI, true);
Supplier supplier = (Supplier)resource.getContents().get(0);
for (Iterator iter =
  EcoreUtil.getAllContents(Collections.singleton(supplier));
  iter.hasNext(); ) {
  EObject eObject = (EObject)iter.next();
  visitor.doSwitch(eObject);
}
```

As you can see, after loading the document in the usual way, we simply iterate over the contents and call `visitor.doSwitch()` for each node in the tree. The `doSwitch()` method will subsequently call out to the appropriate `case()` method, based on the type of object encountered. If we run this program, it will produce the following output:

⁹If you're wondering why we don't just use the `eAllContents()` method on the `EObject` interface, it's because we want to include the root object in the walk as well.

```

visiting Supplier: S1
visiting Customer: 0
visiting Customer: 1
visiting Customer: 2
visiting PurchaseOrder: P0
visiting Item: I2
visiting Item: I3
visiting Item: I4
visiting PurchaseOrder: P1
visiting Item: I1
visiting PurchaseOrder: P2
visiting Item: I1

```

As you can see, the `TreeIterator` returned from the `getAllContents()` method produces a depth-first traversal of the containment structure of the model.

Using Cross Referencers

When working with EMF models, one often needs to search for objects in a resource, or possibly an entire resource set, that have references that meet certain criteria. For example, when deleting an object you may want to search for other objects that have one-way references to it, so you can clean them up. Another common example is to search a resource for objects that have unresolvable references (that is, broken proxies).

EMF includes several cross referencer classes that can help with these kinds of things. You may recall that, in Chapter 12, we already used one of these classes, `EcoreUtil.UsageCrossReferencer`, to find the customer associated with a given purchase order from the `ExtendedPO3` model. You may recall that, in that model, the `orders` reference from class `Customer` to class `PurchaseOrder` is one way, so we couldn't simply navigate to it (Figure 13.1):



Figure 13.1. One-way reference.

We started by using the cross referencer's static `find()` method to retrieve all of the references to a particular purchase order, `order`, from any other object in the resource set:

```

Collection references =
    EcoreUtil.UsageCrossReferencer.find(order,
        order.eResource().getResourceSet());

```

The cross referencer returned a collection of `EStructuralFeature.Settings`, representing all of the incoming references. Since we were really only interested in the customer reference, we then needed to iterate through the collection to find it:

```

for (Iterator iter = references.iterator(); iter.hasNext(); ) {
    EStructuralFeature.Setting setting =
        (EStructuralFeature.Setting)iter.next();
    if (setting.getEstructuralFeature() ==
        SupplierPackage.eINSTANCE.getCustomer_Orders()) {
        Customer customer = (Customer)setting.getEObject();
        ...
    }
}

```

We needed to do this because we chose to use the cross referencer's default implementation. If instead we created a subclass of `UsageCrossReferencer`, we could have customized the search to match our more specific search criteria. For example, the following anonymous subclass would do the trick:

```

Collection references =
    new EcoreUtil.UsageCrossReferencer(
        order.eResource().getResourceSet())
{
    protected boolean crossReference(EObject eObject,
                                    EReference eReference,
                                    EObject crossReferencedObject) {
        return order == crossReferencedObject && eReference ==
            SupplierPackage.eINSTANCE.getCustomer_Orders();
    }

    public Collection findUsage(EObject eObject) {
        return super.findUsage(eObject);
    }

}.findUsage(order);

```

As you can see, we simply override the `crossReference()` method to return `true` if the specified `crossReferencedObject` is the purchase order we're interested in (that is, `order`), and if `eReference` is the `orders` feature from class `Customer`.

Class `UsageCrossReference`, like every other cross referencer in `EcoreUtil`, works by iterating through the content tree rooted at the object, resource, or resource set passed to its constructor (the resource set, in this case). For each object encountered, it retrieves all of its cross references (that is, non-containment references) by calling `eCrossReferences()` from the `EObject` interface. For each cross reference returned, it then calls the `protected crossReference()` method on itself, passing the object that owns the reference (`eObject`), the reference type (`eReference`), and the target of the reference (`crossReferencedObject`). Every cross reference for which the `crossReference()` method returns `true` is then added to the result set. In our example, we've only returned `true` for the customer references we're interested in.

Because we've created a subclass of `UsageCrossReferencer`, we can't simply call the static `find()` method to invoke the search as before; we need to call the `findUsage()` method instead. Notice how we also needed to override the `findUsage()` method in our subclass. This is simply to make it public. It's protected in the base class because there, calling the static `find()` method is the proper way to do it.

By subclassing a cross referencer, we also have an opportunity to speed up the search by reducing the number of objects being checked. In our example, where we are only interested in references from customers, there is no need to search for referencing objects in the containment tree below a purchase order. The only possible children of a purchase order are items and address objects, never customers. We can stop the cross referencer from going further by overriding the `containment()` method like this:

```

Collection references =
    new EcoreUtil.UsageCrossReferencer(
        order.eResource().getResourceSet())
{
    protected boolean crossReference(EObject eObject,
                                    EReference eReference,
                                    EObject crossReferencedObject) {
        return order == crossReferencedObject && eReference ==
            SupplierPackage.eINSTANCE.getCustomer_Orders();
    }

    protected boolean containment(EObject eObject)
    {
        return !(eObject instanceof PurchaseOrder);
    }

    public Collection findUsage(EObject eObject) {

```

```

        return super.findUsage(eObject);
    }

}.findUsage(order);

```

This would prevent it from continuing its algorithm with the children objects of any purchase order it encounters.

In addition to `UsageCrossReferencer`, class `EcoreUtil` includes several other cross referencer classes. They all work essentially the same way, only with different built-in search criteria. Another particularly useful one is class `UnresolvedProxyCrossReferencer`, which can be used to find broken (unresolvable) references in a resource, or resource set, to clean up after another (referenced) resource is deleted or moved, for example.

```

Map proxies =
EcoreUtil.UnresolvedProxyCrossReferencer.find(resource);
for (Iterator entries =
proxies.entrySet().iterator(); entries.hasNext(); ) {
Map.Entry entry = (Map.Entry)entries.next();
EObject proxy = (EObject)entry.getKey();
Collection settings = (Collection)entry.getValue();
...
}

```

As you can see, the `find()` method of `UnresolvedProxyCrossReferencer` returns a `Map`, instead of just the collection of settings that were returned by `UsageCrossReferencer`. Each map entry's key is a target object (proxy), and the entry value is the collection of settings referencing that proxy.

Dynamic EMF

1942365

As we've seen throughout this book, both the EMF generator and the code it generates are based on and leverage the Ecore metamodel. So far we've mainly focused on the use of EMF to generate code for an object model; we haven't really considered the situation where an application simply wants to share data without the need for a generated type-safe API. The reflective EMF API is sometimes all one really needs.

EMF provides a dynamic implementation of the reflective API (that is, the `EObject` interface) which, although slower than the one provided by the generated classes, implements the exact same behavior. If you don't need a type-safe API, then the only advantage of generating Java classes, as opposed to simply using the dynamic implementation, is that they use less memory and provide faster access to the data. The down-side is that the generated classes have to be maintained as the model evolves, and they have to be deployed along with the application. This is the normal trade-off between dynamic and static implementations.

One possible way of getting the performance benefit of code generation without paying the price of static implementation classes is by invoking the code generator at runtime and then dynamically loading the generated Java classes. This approach involves significantly more startup cost and requires access to the EMF and JDT development-time plug-ins at runtime, but does provide the performance of generated classes without the need for statically loaded classes. The approach is analogous to the way a JIT compiler can be used to boost the performance of a Java program.

For pure runtime uses, the development tool prerequisite of the dynamic generation approach is often problematic. Also, because the topic of programmatic generator invocation is beyond the scope of this book, let's instead assume that we have a scenario where we've decided that it's impractical or otherwise undesirable to generate our model and decide to use EMF in a purely dynamic way.

For this example, we'll use a completely different model so that we won't confuse it with any of the purchase order models for which we've been generating Java classes throughout the book. We'll use a model of a company's departments and employees that looks like this in UML (Figure 13.2):



Figure 13.2. A simple company model.

In this model we have two classes, **Department** and **Employee**, with a containment association between them. Both classes have two attributes. Departments have a name and department number while employees have a name and a flag to indicate whether the employee is a manager. Notice that this model is quite similar to the simple purchase order model that we used in Chapter 2, but unlike that model, we're not going to import this model into EMF and generate Java interfaces and classes for it. Instead, for this model we will simply manufacture the core model at runtime and then create instances of it dynamically.

To get started, the first thing we need is an in-memory representation of the core model. We'll create it programmatically like this:

```

EcoreFactory ecoreFactory = EcoreFactory.eINSTANCE;
EcorePackage ecorePackage = EcorePackage.eINSTANCE;

EClass employeeClass = ecoreFactory.createEClass();
employeeClass.setName("Employee");

EAttribute employeeName = ecoreFactory.createEAttribute();
employeeName.setName("name");
employeeName.setEType.ecorePackage.getEString();
employeeClass.getEAttributes().add(employeeName);
EAttribute employeeManager = ecoreFactory.createEAttribute();
employeeManager.setName("manager");
employeeManager.setEType.ecorePackage.getEBoolean();
employeeClass.getEAttributes().add(employeeManager);

EClass departmentClass = ecoreFactory.createEClass();
departmentClass.setName("Department");

EAttribute departmentName = ecoreFactory.createEAttribute();
departmentName.setName("name");
departmentName.setEType.ecorePackage.getEString();
departmentClass.getEAttributes().add(departmentName);

EAttribute departmentNumber = ecoreFactory.createEAttribute();
departmentNumber.setName("number");
departmentNumber.setEType.ecorePackage.getEInt();
departmentClass.getEAttributes().add(departmentNumber);

EReference departmentEmployees = ecoreFactory.createEReference();
departmentEmployees.setName("employees");
departmentEmployees.setEType(employeeClass);
departmentEmployees.setUpperBound(
    EStructuralFeature.UNBOUNDED_MULTIPLICITY);
departmentEmployees.setContainment(true);
departmentClass.getEReferences().add(departmentEmployees);

EPackage companyPackage = ecoreFactory.createEPackage();
companyPackage.setName("company");
companyPackage.setNsPrefix("company");
companyPackage.setNsURI("http://com.example.company.ecore");
companyPackage.getEClassifiers().add(employeeClass);
companyPackage.getEClassifiers().add(departmentClass);
  
```

As you can see, we simply use the `EcoreFactory` to instantiate the necessary Ecore objects. First we create and initialize the `EClass` for **Employee**. Next we create and initialize `EAtributes` representing the employee **name** and **manager** attributes, and add them to the `eAttributes()` list of the `EClass`.¹⁰ After that we repeat the process for class **Department** and its features, only for the **employees** feature we create an `EReference` instead of `EAtribute`. Finally we create and initialize an `EPackage` for the model and add the two classes to it.

Now that we've built up this model in memory, we can immediately use it to create instances of the model. For example, the following code creates a department containing two employees:

```
EFactor companyFactory = companyPackage.getEFactoryInstance();

EObject employee1 = companyFactory.create(employeeClass);
employee1.eSet(employeeName, "John");

EObject employee2 = companyFactory.create(employeeClass);
employee2.eSet(employeeName, "Katherine");
employee2.eSet(employeeManager, Boolean.TRUE);

EObject department = companyFactory.create(departmentClass);
department.eSet(departmentName, "ABC");
department.eSet(departmentNumber, new Integer(123));
((List)department.eGet(departmentEmployees)).add(employee1);
((List)department.eGet(departmentEmployees)).add(employee2);
```

Notice that we access the factory by calling the `getEFactoryInstance()` method on our company package. If the company package had been generated, this method would return the corresponding generated factory. Since there is no generated factory, or generated classes of any type, the `getEFactoryInstance()` method returns the dynamic factory, `EFactorImpl`, instead. Once we have the factory, we simply create and initialize the instances using the reflective API in the usual way (that is, the same as if we were using the reflective API to manipulate generated classes).

Since we didn't generate classes for the company model, the objects created by `EFactorImpl` will be instances of class `EObjectImpl`. You might recall from Chapter 9 that `EObjectImpl` is the base class of all generated classes as well. That's true, but in the purely generated case the generated implementations override the parts related to dynamic EMF. However, dynamic classes can also be created that extend from generated ones. The dynamic factory, `EFactorImpl`, implements the `create()` method something like this:

```
public EObject create(EClass eClass) {
    for (List eSuperTypes =
        eClass.getESuperTypes(); !eSuperTypes.isEmpty(); ) {
        EClass eSuperType = (EClass)eSuperTypes.get(0);
        if (eSuperType.getInstanceClass() != null) {
            EObject result =
                eSuperType.getEPackage().getEFactoryInstance().create(
                    eSuperType);
            ((InternalEObject)result).eSetClass(eClass);
            return result;
        }
        eSuperTypes = eSuperType.getESuperTypes();
    }

    EObjectImpl result = new EObjectImpl();
    result.eSetClass(eClass);
    return result;
}
```

¹⁰Notice how we use the `EcorePackage` to reference the simple data types. If we were using a more complex type, like `java.lang.Date` for example, then we would have also needed to instantiate an `EDataType` and call the `setInstanceClass()` method on it. Since our model only uses Ecore's built-in data types, we didn't need to do that.

Notice that before creating an instance of `EObjectImpl`, it first tries to find a supertype that has an instance class (that is, a corresponding generated class). If it finds one, it calls `create()` on its (generated) factory, which in turn will instantiate the generated Java implementation class of the supertype. So, if a generated base class exists, then an instance will be implemented by a combination of generated features (from the base `EClass`) and dynamic features (from the dynamic `EClass` subclass) that will be implemented in `EObjectImpl`, which in this case is the base class of the actual Java implementation class being used.

In our example, we have no generated base class, so our objects will be simple instances of `EObjectImpl`. As you can see in the `create()` method above, a dynamic instance must be provided with its corresponding `EClass`, which is used by `EObjectImpl` to implement the getting and setting of the dynamic features. Dynamic instances allocate additional storage for dynamic settings as well as the `EClass` itself. By contrast, if custom Java classes are generated, then no storage is required and the `EClass` is not used in the processing.

Now that we've created our instances, we can serialize it exactly the same way as we do for any other EMF model. Assuming `XMLResourceImpl` is registered to handle `.xml`/files, we can save it as `company.xml`/like this:

```
ResourceSet resourceSet = new ResourceSetImpl();
Resource resource = resourceSet.createResource(
    URI.createURI("platform:/resource/project/company.xml"));
resource.getContents().add(department);
resource.save(null);
```

The resulting file looks, as you'd expect, like this:

```
<company:Department
  xmlns:company="http://com.example.company.ecore"
  name="ABC" number="123">
<employees name="John"/>
<employees name="Katherine" manager="true"/>
</company:Department>
```

It's no different from what we'd get, had we been using a generated core model. You can see from this how code generation is only an in-memory concept in EMF. Once serialized, all models are equal.

So, now that we've serialized our dynamic model, you may be wondering how to read it back. The answer is really quite simple. Recall from Section 13.1.2 where we explained that, when loading a model instance document, the package namespace URI needs to be available, either by loading or in the package registry. For generated models, the generated packages are automatically added to the registry, but not in our case. We could do it manually, like this:

```
EPackage.Registry.INSTANCE.put("http://com.example.company.ecore",
    companyPackage);
```

We simply add our dynamic `companyPackage` to the package registry using the same namespace URI, `http://com.example.company.ecore`, which we set in the `EPackage` before we serialized the instance document.

Now that we have the package in the registry, we can proceed to load the instance document, and access the instances in the usual (reflective) way:

```
ResourceSet resourceSet = new ResourceSetImpl();
URI fileURI =
    URI.createURI("platform:/resource/project/company.xml");
Resource resource = resourceSet.getResource(fileURI, true);
EObject department = (EObject)resource.getContents().get(0);
```

Although the package registry approach will work, one thing you may be thinking is that a document's save and load operations don't usually happen in the same program, so where will the package that we add to the registry actually come from? Will we need to rerun the code that builds it up before we do any resource loading?

For generated packages, that's exactly how it works; the `init()` method in the generated package implementation class is invoked when the package is first accessed, which builds the package and adds it to the registry. So, for our dynamic package, we could do the same thing, although a good alternative is to serialize the dynamic core model and use an `xsi:schemaLocation` attribute to locate and load the package automatically.

To see how this works, let's change our example so that before we save the `company.xml`/resource, we first serialize the package in another resource, `company.ecore`, like this:

```
ResourceSet resourceSet = new ResourceSetImpl();
Resource metaResource = resourceSet.createResource(
    URI.createURI("platform:/resource/project/company.ecore"));
metaResource.getContents().add(companyPackage);
metaResource.save(null);
```

Now, when serializing the instance document, `company.xml`, we'll use the `OPTION_SCHEMA_LOCATION` save option, described in Section 13.3.2, to produce an `xsi:schemaLocation` attribute in the document:

```
ResourceSet resourceSet = new ResourceSetImpl();
Resource resource = resourceSet.createResource(
    URI.createURI("platform:/resource/project/company.xml"));
resource.getContents().add(department);
Map options = new HashMap();
options.put(XMLResource.OPTION_SCHEMA_LOCATION, Boolean.TRUE);
resource.save(options);
```

If we look at the serialized `company.xml`/file, it will now look like this:

```
<?xml version="1.0" encoding="ASCII"?>
<company:Department
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:company="http://com.example.company.ecore"
  xsi:schemaLocation=
    "http://com.example.company.ecore company.ecore#/"
  name="ABC" number="123">
  <employees name="John"/>
  <employees name="Katherine" manager="true"/>
</company:Department>
```

As you can see, it's exactly the same as before, except for two new attributes: the `xsi` namespace declaration and the `xsi:schemaLocation` that we requested. The value of an `xsi:schemaLocation` attribute contains one or more pairs of namespace URI to package URI mappings. In our example, it contains a single mapping from our dynamic package's namespace URI, `http://com.example.company.ecore`, to the root (fragment "/") object (that is, the `EPackage`) in our serialized core model, `company.ecore#/`. Notice that the URI is relative, because we used absolute URIs for the documents, as described in Section 13.3.1.

Now that we've serialized both the core model and the instance document with an `xsi:schemaLocation`, we don't have to do anything special before reloading. We can simply load the instance document, `company.xml`, and using the mapping information in the `xsi:schemaLocation`, the loader will automatically load the metamodel, `company.ecore`, on demand.

Chapter 14. EMF.Edit Programming

By now, you probably have a fairly good understanding of EMF and what you can do with it. You also know how EMF.Edit and its code generator let you quickly create viewers and editors for your EMF models. We've talked about how the framework is also designed to allow you to easily customize your views and editors, but we haven't really seen how that works yet.

The best way to show how easily EMF.Edit behavior can be customized is by looking at some examples. The following sections include several examples, each of which implements one or more common kinds of customization. Through the examples, we'll also introduce some of EMF.Edit's convenience classes that help make these kinds of customizations easier to implement.

Overriding Commands

In Chapter 3, we explained how modifications to models in EMF.Edit are made using commands. In an EMF.Edit-based editor, all the menu actions, property sheet changes, and even drag-and-drop, are implemented this way. By constructing commands using an editing domain, instead of simply using the Java `new` operator, the framework provides a convenient way for you to override the behavior of any editing command on your model; you can override the `createCommand()` method to return your own specialized implementation class.

We described in Section 3.3.3 how the default editing domain, `AdapterFactoryEditingDomain`, delegates the `createCommand()` method to the item provider of the primary object involved in the command (the command owner). We also mentioned how EMF.Edit's item provider base class, `IItemProviderAdapter`, implements the `createCommand()` method for all the generic EMF.Edit commands (`AddCommand`, `RemoveCommand`, and so on) described in Section 3.3.2, but we did not elaborate. Let's take it from there.

`IItemProviderAdapter` implements `createCommand()` by dispatching the creation of each of the standard (built-in) commands to command-specific convenience methods, something like this:

```
public Command createCommand(final Object object,
                            final EditingDomain domain,
                            Class commandClass,
                            CommandParameter commandParm) {
    ...
    if (commandClass == RemoveCommand.class)
        return createRemoveCommand(domain,
                                   commandParameter.getEObject(),
                                   commandParameter.getEReference(),
                                   commandParameter.getCollection());
    else if (commandClass == AddCommand.class)
        return createAddCommand(domain,
                               commandParameter.getEObject(),
                               ...);
    else ...
}
```

The default implementation of each of these convenience methods simply returns a new instance of the appropriate command. For example, the `createCommand()` method in class `ItemProviderAdapter` looks like this:

```
protected Command createRemoveCommand(EditingDomain domain,
                                     EObject owner,
                                     EReference feature,
                                     Collection collection)
{
    return new RemoveCommand(domain, owner, feature, collection);
}
```

With this design, you can customize any predefined EMF.Edit command by simply overriding its command-specific `create()` method; you don't need to override `createCommand()` itself unless you're adding a new type of command. Notice how, before calling the `create()` methods, the arguments for the various commands are first extracted from the generic `commandParameter` argument. The `create()` methods have the same signature as their corresponding command constructors, making them a most convenient override point.

The first example we will look at involves customizing the purchase order application to support volume discounting. For the purpose of this example, any of the model versions that we've introduced previously will do, so we might as well use the simplest one, SimplePO, which was first introduced back in Chapter 2.

Recall that a purchase order item included, among other things, `price` and `quantity` attributes (Figure 14.1):

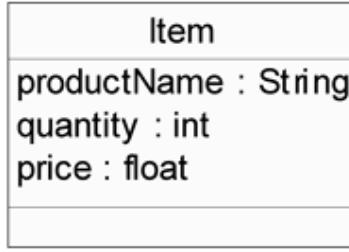


Figure 14.1. Purchase order item attributes.

In our example application we are going to provide a 10 percent price discount for item quantities of 10 or more, and we will implement this “volume discounting” by overriding the `SetCommand`. We need to implement the following special behavior:

1. When we set the quantity to a value greater than or equal to 10 and the previous quantity is less than 10, then we will also reduce the price by 10 percent.
2. When we set the quantity to a value less than 10 and the previous quantity is greater than or equal to 10, then we will increase the price (that is, remove the 10 percent discount).
3. If we set the price while the quantity is greater than or equal to 10, then we will reduce the price by 10 percent before setting it.

Because we're changing the behavior when setting either the `price` or `quantity`, we'll need to customize the `SetCommand` for both attributes. We'll do this using two command subclasses, `SetPriceCommand` and `SetQuantityCommand`.

`SetPriceCommand` is a trivial subclass of the generic `SetCommand` that looks like this:

```
public class SetPriceCommand extends SetCommand
{
    public SetPriceCommand(EditingDomain domain, EObject owner,
                           EStructuralFeature feature, Object value) {
```

```

        super(domain, owner, feature, value);
    }

    public void doExecute() {
        Item item = (Item)owner;
        if (item.getQuantity() >= 10) {
            double newPrice = ((Float)value).floatValue() * 0.9;
            value = new Float(newPrice);
        }
        super.doExecute();
    }
}

```

As you can see, all we need to do is override a single method, `doExecute()`,¹ to reduce the value (that is, the price being set) if the item's quantity is greater than or equal to 10. Because we're changing the actual `value` instance variable during `doExecute()`, the `undo()` and `redo()` methods will work without change. Notice how we downcast the `owner` and `value` instance variables (from the base class, `SetCommand`) to `Item` and `Float`, respectively. We can safely do this because we will only be using this command for setting the `price` attribute on purchase order items. We'll see how we arrange for this below, but first let's take a look at our other command subclass, `SetQuantityCommand`:

```

public class SetQuantityCommand extends CompoundCommand
{
    protected Item item;
    protected Object value;
    protected EditingDomain domain;

    public SetQuantityCommand(EditingDomain domain, Item item,
                           EStructuralFeature feature, Object value) {
        super(0);
        this.item = item;
        this.value = value;
        this.domain = domain;
        append(new SetCommand(domain, item, feature, value));
    }

    public void execute() {
        int oldQuantity = item.getQuantity();
        super.execute();
        int quantity = ((Integer)value).intValue();
        if (oldQuantity < 10 && quantity >= 10 || 
            oldQuantity >= 10 && quantity < 10) {
            double newPrice = quantity >= 10 ?
                item.getPrice() * 0.9 : item.getPrice() / 0.9;
            appendAndExecute(new SetCommand(domain, item,
                POPackage.eINSTANCE.getItem_Price(),
                new Float(newPrice)));
        }
    }
}

```

The `SetQuantityCommand` is also quite simple, but notice how it's a subclass of `CompoundCommand` instead of `SetCommand`. The EMF.Edit framework never expects concrete command classes; it only ever deals with commands using the base interface `Command`. Even the return type of the `createSetCommand()` method is `Command`; not `SetCommand`. This gives us a great deal of flexibility when overriding commands. As long as we implement the correct command semantics, we can use any implementation class we wish.

¹If you're wondering why we override the `doExecute()` method, instead of just `execute()`, it's because the generic `SetCommand`, like all the EMF.Edit generic commands, is a subclass of `AbstractOverrideableCommand`, which introduces this pattern as described in Chapter 3.

As you can see, the purpose of the `CompoundCommand` is to aggregate a `SetCommand` for the quantity, with possibly another one for setting the price. In the constructor, we create a “real” `SetCommand` for the quantity and append it to the empty compound command (that is, `this`) by calling `append()`. Notice that we also record the `item`, `value`, and `domain` in instance variables, so that we can use them during `execute()` where we will need to determine if a change in price is also needed.

The `execute()` method works by first calling `super.execute()` to execute the actual `SetCommand` for the quantity. Next it performs the necessary checks to determine if a price increase or decrease is also required and if so, it calls the `CompoundCommand.appendAndExecute()` method to append, and immediately execute, another `SetCommand` for the price.

Notice that with this compound command approach, we didn’t need to override any other methods. By optionally appending the second `SetCommand` to the compound command during `execute` this way, the default `undo()` and `redo()` methods (from class `CompoundCommand`) will simply call the corresponding methods on the one or two actual `SetCommands` that were executed.

Now that we have both our command override classes, all we need to do is plug them into the framework. Assuming we are using generated item providers, we can simply override the `createSetCommand()` method in class `ItemItemProvider` (that is, the item provider for class `Item`) like this:

```
protected Command createSetCommand(EditingDomain domain,
                                    EObject owner,
                                    EStructuralFeature feature,
                                    Object value) {
    if (feature == POPackage.eINSTANCE.getItem_Price())
        return new SetPriceCommand(domain, owner, feature, value);
    else if (feature == POPackage.eINSTANCE.getItem_Quantity())
        return new SetQuantityCommand(domain, (Item)owner,
                                      feature, value);
    return super.createSetCommand(domain, owner, feature, value);
}
```

The first check ensures that whenever a `SetCommand` is created for the `price` feature of a purchase order item (class `Item`), our `SetPriceCommand` will be used instead of the default implementation. The second check ensures that our `SetQuantityCommand` will be used whenever setting the `quantity` attribute. Any other feature will be set using the default `SetCommand`, which is constructed by calling `super.createSetCommand()`.

So now that we’ve made these changes, you might be wondering what the visible effect will be. As we mentioned previously, property sheet changes in an EMF.Edit editor are implemented using `SetCommand`, so you can use the property sheet of the EMF.Edit generated editor to try out the new commands.

For example, if you run the purchase order editor and select an item with a quantity less than 10, the property sheet will look something like Figure 14.2.

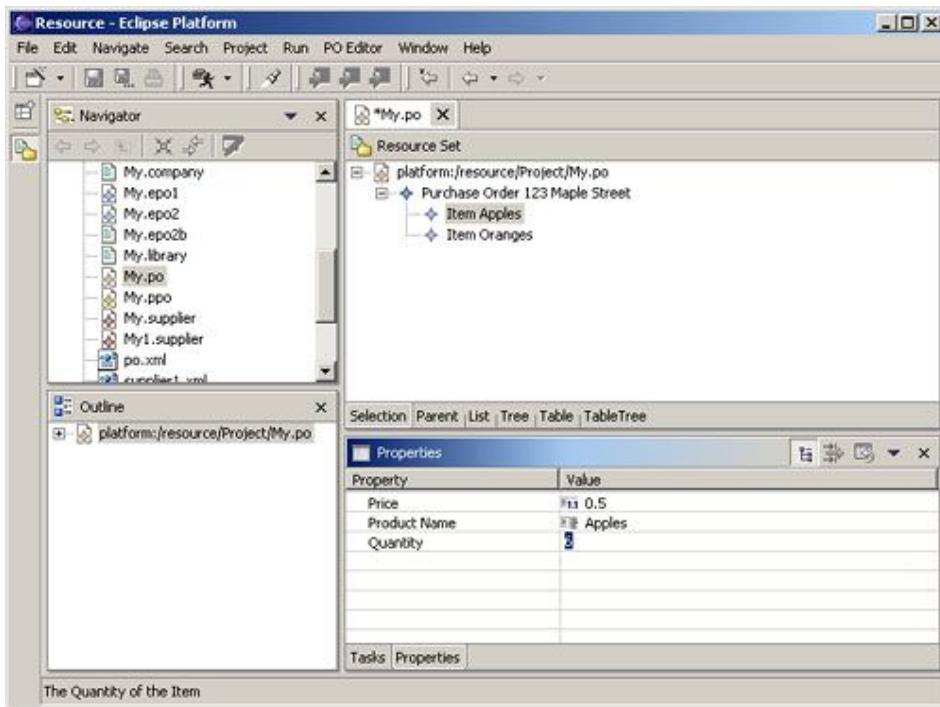


Figure 14.2. Property Sheet view of a purchase order item.

If you now change the “Quantity” to a value greater than 10 and press Enter, you’ll see that the “Price” will now be reduced by 10 percent, as shown in Figure 14.3. You can also invoke the undo and redo actions, and you’ll see that both attributes will be updated for them as well.

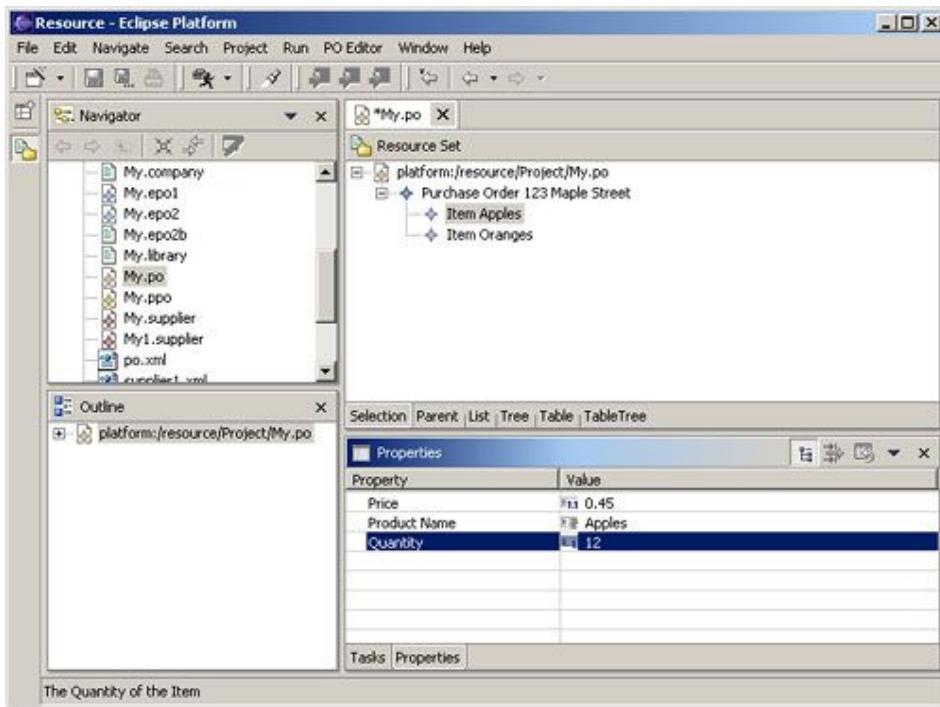


Figure 14.3. Volume discounting in action.

As you can see, overriding standard commands in EMF.Edit is really quite simple. The design we chose, using `CompoundCommand` as the base of `SetQuantityCommand`, is not the only way of doing this, but it is the simplest; it involves the minimum number of method overrides. If, instead, you try to do it by subclassing `SetCommand` (and maintaining the second `SetCommand` in an instance variable), for example, you'll see that's also simple enough, but it will require trivial overrides of the `undo()` and `redo()` methods as well.

Customizing Views

Whether you use a reflective item provider or generated ones, the views that EMF.Edit provides by default correspond very closely to the structure of the model. However, the structure that we want to display in a view is often quite different from that of the model. We've seen in Chapters 11 and 12 how you can use options in the generator model to control the EMF.Edit behavior (for example, the references considered to be children, the features displayed in the property sheet, and so on). Now, we'll consider some more significant changes that require hand coding to implement.

In the following sections we'll look at three examples that change the view structure. In the first example, we'll modify the purchase order editor to suppress some of the model objects from the view. In the second example, we'll show how to modify the generated editor to use the list and table viewers to display more interesting data than they do by default. The final example will enhance the generated views by introducing additional (non-modeled) intermediate objects into the view structure.

Suppressing Model Objects

One of the most common customizations to an EMF.Edit generated editor involves suppressing some of the model objects from the view. In Chapters 10 and 11, we saw how, by setting the "Children" property to `false` in the generator model, we can avoid displaying some or all of the children of an object. Unfortunately, that's usually not the end of the story. Most of the time, information from the suppressed objects still needs to be displayed somewhere. For example, sometimes the attributes from the suppressed child should be used in the label of the parent object, or perhaps displayed as properties of the parent. Another possibility is that the suppressed object's children need to be pushed up a level and displayed as children of the parent (that is, their grandparent).

This example is based on the PrimerPO model from Chapter 4. One of the most undesirable characteristics of this model (from the perspective of EMF.Edit's default behavior) is that class `PurchaseOrder` includes containment references for items as well as addresses (`billTo` and `shipTo`), as shown in Figure 14.4.

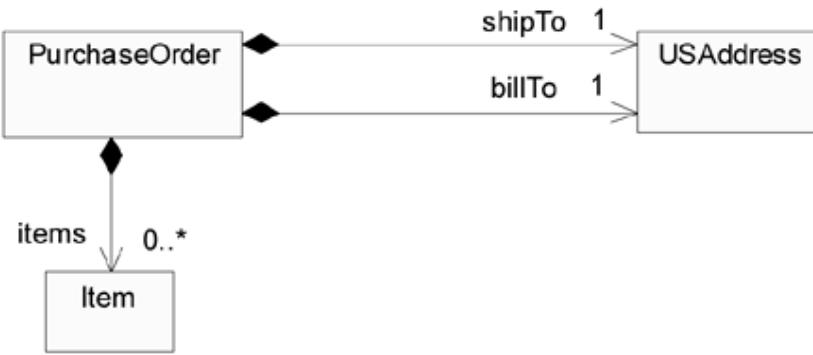


Figure 14.4. Model with multiple containment references.

As a result, the default purchase order editor displays both items and address as the children of a purchase order, as shown in Figure 14.5. To set the actual **shipTo** (or **billTo**) attributes (that is, **name**, **street**, **city**, and so on) in the property sheet, you would first need to select the appropriate address in the tree view, for example. Notice that by default it isn't even clear which of the two addresses is the **shipTo** and which is the **billTo**.

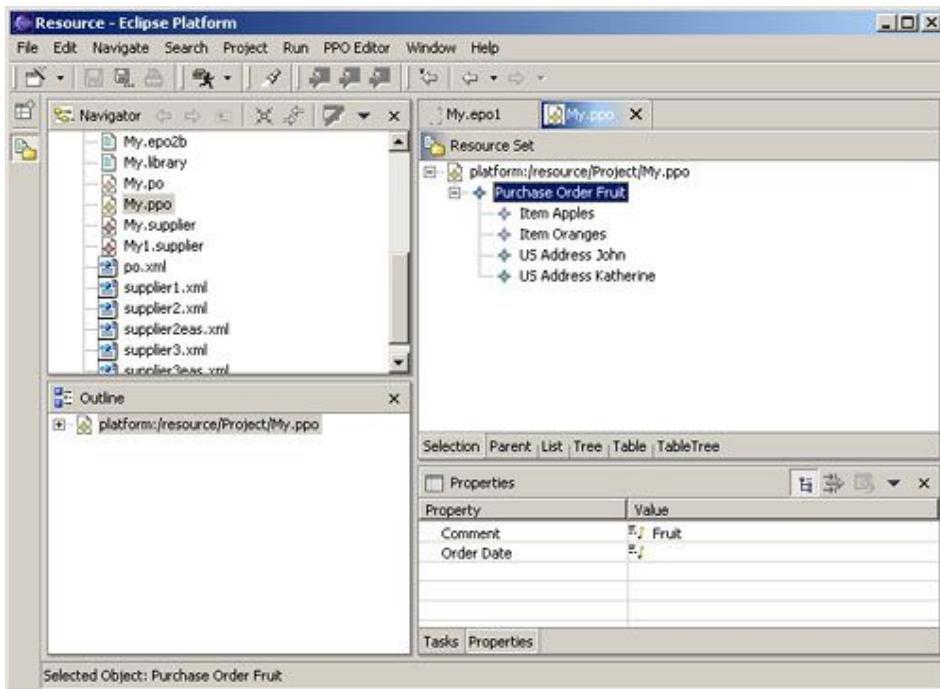


Figure 14.5. Default tree view of the PrimerPO model.

Although useable, this is clearly not the best way to display the model. A better approach, which we will implement in this example, is to suppress the addresses and only show the items as children, as shown in Figure 14.6. Notice that this view also has another change: we suppressed the root object (that is, the resource *My.ppo*) and made the purchase order itself the root. This is another common example where suppression makes sense, since this editor will only ever be editing a single resource at a time.

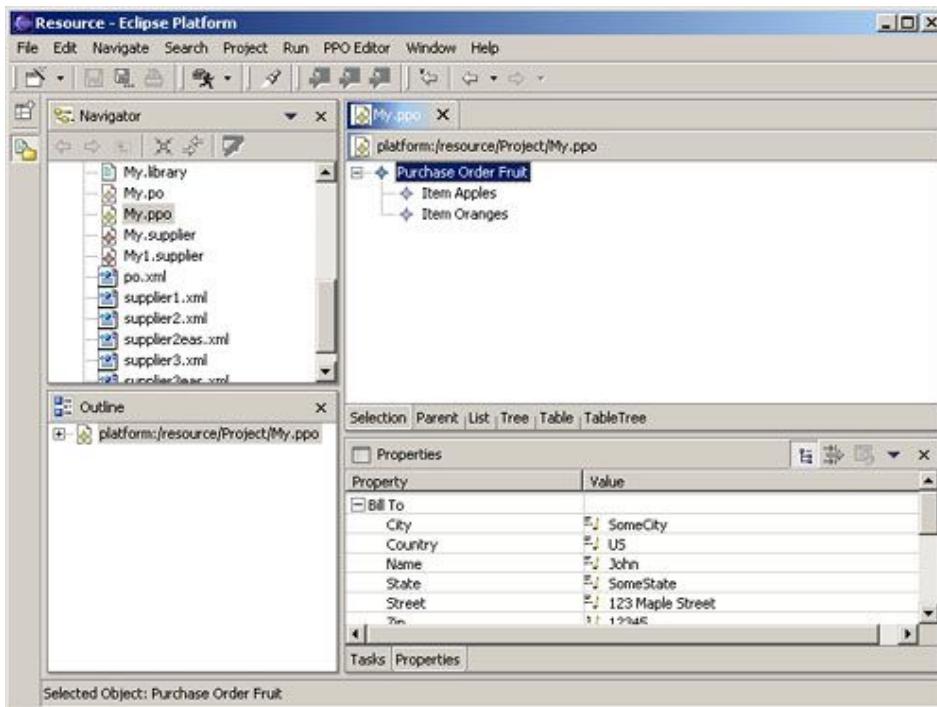


Figure 14.6. Improved purchase order tree view.

Now that we're suppressing the `billTo` and `shipTo` addresses, we need to figure out how the user will set their attributes. Making them additional properties of the purchase order would make sense, since they're really just the components of two compound attributes of it, anyway. In this example, we'll make the property sheet display them as categorized attributes of the purchase order as shown in Figure 14.7.

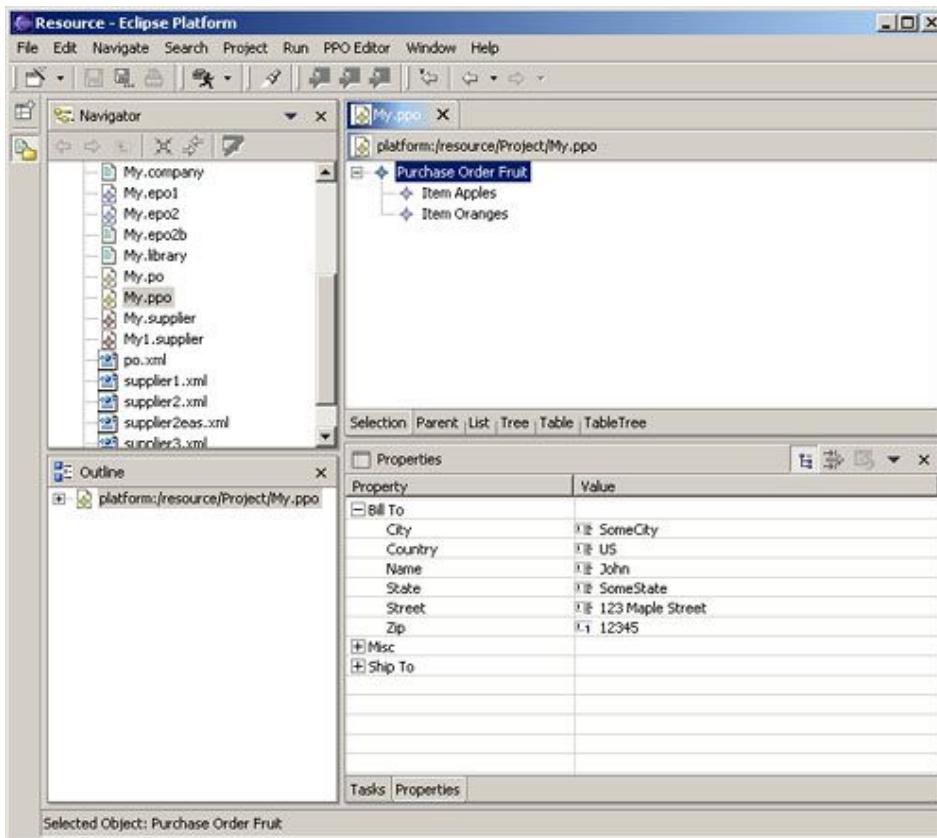


Figure 14.7. Customized purchase order property sheet.

Now that we've figured out how we want the purchase order editor to look, let's get to it. The first thing we will change is actually the simplest; suppressing the root object (resource) from the tree view. The root object of the tree viewer (**Selection** tab) is set by a call to the `setInput()` method from the `createPages()` method of the generated editor like this:

```
public void createPages() {
    ...
    selectionViewer.setInput(editingDomain.getResourceSet());
    viewerPane.setTitle(editingDomain.getResourceSet());
    ...
}
```

Notice that it sets the input to be the resource set, not the resource. You may recall from Chapter 10 that this is because the tree viewer's input is an invisible root, whose children (the single resource, *My.ppo*, in this case) are the actual displayed roots. So, to change the displayed root to be the purchase order, instead of the resource, all we need to do is set the input and title (since we'll want the pane title to also reflect this change) to the resource, like this:

```
/**
 * @generated NOT
 */
public void createPages() {
    ...
    selectionViewer.setInput(editingDomain.getResourceSet());
    viewerPane.setTitle(editingDomain.getResourceSet());
    ResourceSet resourceSet = editingDomain.getResourceSet();
    Resource resource = (Resource)resourceSet.getResources().get(0);
```

```

selectionViewer.setInput(resource);
viewerPane.setTitle(resource);
...
}

```

Since we know there is only one resource in the resource set, we simply get the first resource from the set by calling `get(0)`. Then we use it as the input and title for the view.

Because we're changing a generated method here, we also need to remove the `@generated` tag from the method comment. Notice that instead of actually deleting it, we did this by adding `NOT` after the `@generated` tag, as described in Chapter 12, so that we can easily keep track of which generated methods we've modified.

Now that we've eliminated the resource, the next job is to suppress the two addresses, `shipTo` and `billTo`. The best way to do this is by simply setting the "Children" property to `false` on the two features in the generator model, and then regenerating the `PurchaseOrderItemProvider`.² Once we do that, the generated `getChildrenReferences()` method should only include the `items` reference:

```

public Collection getChildrenReferences(Object object) {
    if (childrenReferences == null) {
        super.getChildrenReferences(object);
        childrenReferences.add(
            PPOPackage.eINSTANCE.getPurchaseOrder_Items());
        childrenReferences.add(
            PPOPackage.eINSTANCE.getPurchaseOrder_BillTo());
        childrenReferences.add(
            PPOPackage.eINSTANCE.getPurchaseOrder_ShipTo());
    }
    return childrenReferences;
}

```

So far, so good. We've eliminated the two addresses from the view, but now we need to add their properties to the purchase order. We can do that by overriding the generated `getPropertyDescriptors()` method in class `PurchaseOrderItemProvider` like this:

```

public List getPropertyDescriptors(Object object) {
    if (itemPropertyDescriptors == null) {
        getPropertyDescriptorsGen(object);

        addAddressPropertyDescriptors(
            ((PurchaseOrder)object).getShipTo(),
            getString("_UI_PurchaseOrder_shipTo_feature"));
        addAddressPropertyDescriptors(
            ((PurchaseOrder)object).getBillTo(),
            getString("_UI_PurchaseOrder_billTo_feature"));
    }
    return itemPropertyDescriptors;
}

```

Here we first call the generated version, which we've renamed to `getPropertyDescriptorsGen()`, to add the normal purchase order properties, and then we use a convenience method, `addAddressPropertyDescriptors()`, to also add the `shipTo` and `billTo` properties. The `addAddressPropertyDescriptors()` method looks like this:

```

public void addAddressPropertyDescriptors(USAddress address,
                                         final String addressFeature) {
    USAddressItemProvider addressItemProvider =
        (USAddressItemProvider)adapterFactory.adapt(
            address, IItemPropertySource.class);
    List addressDescriptors =

```

²You also could have just commented out the two lines if you prefer, but then you'd need to remove the `@generated` tag as well.

```
addressItemProvider.getPropertyDescriptors(address);
for (Iterator iter = addressDescriptors.iterator();
     iter.hasNext(); ) {
    ItemPropertyDescriptor addressDescriptor =
        (ItemPropertyDescriptor)iter.next();
    itemPropertyDescriptors.add(
        new ItemPropertyDescriptorDecorator(address,
                                         addressDescriptor)
    {
        public String getCategory(Object thisObject) {
            return addressFeature;
        }
        public String getId(Object thisObject) {
            return addressFeature + getDisplayName(thisObject);
        }
    });
}
}
```

As you can see, all this method does is delegate to the appropriate address's item provider to get the required property descriptors. Notice how instead of directly adding the descriptors to its list, it uses the EMF.Edit helper class `ItemPropertyDescriptorDecorator` to override the `getCategory()` and `getId()` methods. It needs to override `getCategory()` to achieve the grouping shown in Figure 14.7. The `getId()` override is needed because every property must be uniquely identified, and since we're using the `USAAddressItemProvider`'s properties twice (for `shipTo` and then again for `billTo`), they will otherwise conflict.

So that's all there is to it; the two addresses have been suppressed and their properties merged with those of the purchase order. There are, however, a couple of other minor changes that we should make in support of this change. First, when one of the `USAAddress`'s properties is changed, using the `SetCommand`, we would like to make it appear (in the UI) as if the purchase order, which is pretending to own it, has changed instead of the non-visible `billTo` or `shipTo` address. We can easily do this by overriding the `createSetCommand()` method in `USAAddressItemProvider` like this:

```
protected Command createSetCommand(EditingDomain domain,
                                    EObject owner,
                                    EStructuralFeature feature,
                                    Object value) {
    return
        new SetCommand(domain, owner, feature, value)
    {
        public Collection doGetAffectedObjects() {
            return Collections.singleton(this.owner.eContainer());
        }
    };
}
```

Here we use a simple anonymous subclass of `SetCommand` to override the `doGetAffectedObjects()` method to return the `eContainer()` (that is, the purchase order) of the command owner, which will be either the `shipTo` or `billTo` address. This way the purchase order will be selected whenever an address property is set. Without this change, nothing would appear to be selected because the affected object (the address) is nowhere visible.

The final change we will make for this example relates to the creation of the `USAAddress` objects. Notice how the changes we made in `createSetCommand()` and in `addAddressPropertyDescriptors()` blindly assume that addresses are associated with purchase orders; the associations between them are assumed to never return `null`. If we wanted to do lazy creation of the

addresses, we could add guards in our methods to handle the `null` case appropriately, but to keep the example as simple as possible we decided instead to simply guarantee that a purchase order and its two addresses will always exist as a group. We accomplished this by simply modifying the `createInitialModel()` method in `PPOModelWizard` like this:

```
/**
 * @generated NOT
 */
EObject createInitialModel() {
    PurchaseOrder rootObject = ppoFactory.createPurchaseOrder();
    rootObject.setShipTo(ppoFactory.createUSAddress());
    rootObject.setBillTo(ppoFactory.createUSAddress());
    return rootObject;
}
```

By modifying the `createInitialModel()` method this way, we've removed the ability for the user to select a root object in the wizard. Because of this, we should also remove the initial object creation page from the wizard by modifying the `addPages()` method as we did in the final example of Chapter 12.

Using List and Table Viewers

After having familiarized yourself with EMF.Edit and the kind of editor that it generates, you've probably noticed that the standard tree viewer (**Selection** tab) is the only one that is more than marginally interesting. The other tabs (**Parent**, **List**, and so on) are not providing any additional information. They're all displaying the same relationships, but in a slightly different way.

This is fine for the generated views, which are only intended to illustrate how other viewers work with EMF.Edit, but for a real model editor, like the purchase order editor, there are other more interesting uses for these viewers.

In this example, we are going to modify the generated purchase order editor's **List** and **Table** views to display more interesting information. This example is based on the ExtendedPO1 model, which includes an association between customers and purchase orders (Figure 14.8):

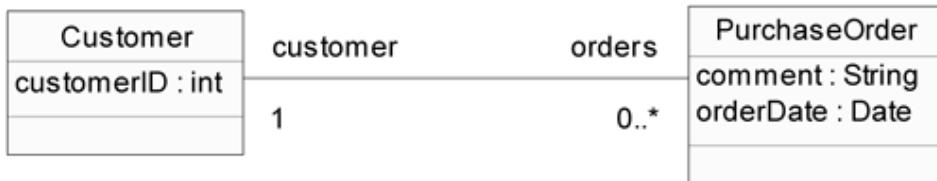


Figure 14.8. Model showing association between customers and purchase orders.

We're going to change the generated editor's list view so that whenever a customer is selected (for example, in the outline view), the list will display that customer's purchase orders, as shown in Figure 14.9.

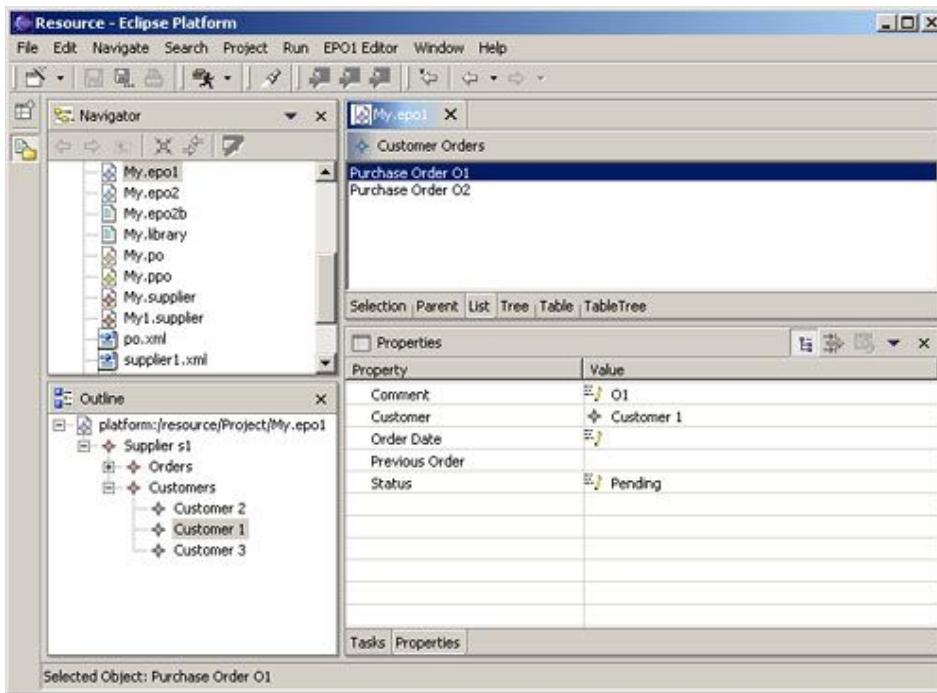


Figure 14.9. Customer orders list view.

To accomplish this, we'll use EMF.Edit's `IItemProvider` convenience class: a customizable item provider implementation for providers that are not also EMF adapters. You may recall from Chapter 10 (and the previous example) that the object passed to a tree viewer (that is, to the `setInput()` method) is not actually displayed. Instead, the `getElements()` method on the content provider is called to retrieve the root objects for the viewer. This is actually true for all JFace viewers, including our list viewer. The object which we pass to the `setInput()` method of the list viewer will only be used to retrieve the actual list of objects to display; it won't be displayed itself. So, to produce our desired list of purchase orders, all we need to do is create an instance of class `IItemProvider` that will return the required purchase orders when the `getElements()` method, from interface `IStructuredContentProvider`, is called. We do it like this in class `EPO1Editor`:

```
/*
 * @generated NOT
 */
public void handleContentOutlineSelection(ISelection selection) {
    if (currentViewerPane != null &&
        !selection.isEmpty() &&
        selection instanceof IStructuredSelection) {
        Iterator selectedElements =
            ((IStructuredSelection)selection).iterator();
        if (selectedElements.hasNext()) {
            Object selectedElement = selectedElements.next();
            if (currentViewerPane.getViewer() == selectionViewer ...) {
            }
            else if (currentViewerPane.getViewer() == listViewer) {
                Collection customerOrders =
                    selectedElement instanceof Customer ?
                        ((Customer)selectedElement).getOrders() :
                        Collections.EMPTY_LIST;

                IItemProvider listRoot =
                    new IItemProvider(
                        "Customer Orders",
                        ExtendedPO1EditPlugin.INSTANCE.getImage(

```

```
    "full/obj16/PurchaseOrder") ,  
    customerOrders);  
listViewer.setInput(listRoot);  
currentViewerPane.setTitle(listRoot);  
}  
else { ...  
}  
}  
}  
}
```

As you can see, we modify the `handleContentOutlineSelection()` method, which is called whenever the selection changes. If the current selection is a customer, then we get its associated purchase orders by calling the `getOrders()` method, create an instance of class `IItemProvider` using them, and then pass it to the `setInput()` and `setTitle()` method of the list viewer. Now, whenever we select a customer in the ExtendedPO1 editor, the list viewer will display the customer's orders as we intended.

Now that we have a more useful list viewer, let's look at enhancing the table viewer. You may have noticed that the table viewer, as generated, isn't particularly interesting; it displays the same thing in each of its two columns. We're going to change it to instead display the complete list of purchase orders in column 0 and each order's associated customer in column 1. Our modified view is shown in Figure 14.10.

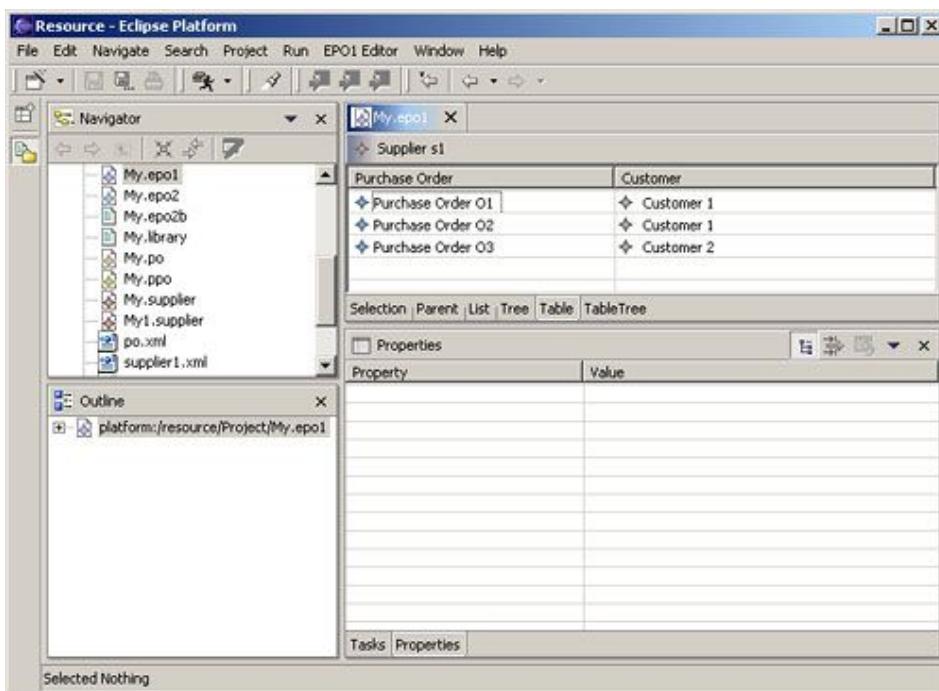


Figure 14.10. Purchase order table view.

In Chapter 3, we mentioned how EMF.Edit's generic content and label provider classes, `AdapterFactoryContentProvider` and `AdapterFactoryLabelProvider`, implement the JFace provider interfaces by delegating to item providers. We described how their implementations delegate to corresponding interfaces on an item provider. As we saw in Chapter 10, the generated item providers (and the reflective one, for that matter) implement all of the required item provider interfaces except one: `ITableItemLabelProvider`. To properly support a table viewer, we will need to mix it in and implement it ourselves.

We explained in Chapter 10 that the generated table viewer works because `AdapterFactoryLabelProvider`'s implementation of the JFace interface `ITableLabelProvider` first checks if the `ITableItemLabelProvider` interface is supported by the item provider and if not, it delegates to the `IItemLabelProvider` interface instead. If you look at the two interfaces, you'll notice that the only difference is that the `getText()` and `getImage()` methods in the table version include a column index argument, while the ordinary label provider interface does not. As a result, any table viewer based on the default item providers will only be able to retrieve one label per row, so if there is more than one column in the table, they will all display the same thing.

So, now that we want to put something more interesting in the table viewer, what we need to do is add the `ITableItemLabelProvider` interface to the implements list of class `PurchaseOrderItemProvider`, like this:

```
public class PurchaseOrderItemProvider
    extends ItemProviderAdapter
    implements
        IEditingDomainItemProvider,
        IStructuredItemContentProvider,
        ITreelistContentProvider,
        IItemLabelProvider,
        ITableItemLabelProvider, // Added to support table view
        IItemPropertySource { ... }
```

Then, we simply implement the `getColumnText()` method to return the associated customer's text for column 1, like this:

```
public String getColumnText(Object object, int columnIndex) {
    if (columnIndex == 0) return getText(object);
    Customer customer = ((PurchaseOrder)object).getCustomer();
    if (customer == null) return "";

    IItemLabelProvider itemLabelProvider =
        (IItemLabelProvider)adapterFactory.adapt(
            customer, IItemLabelProvider.class);
    return itemLabelProvider.getText(customer);
}
```

Notice that for column 0, we retrieve the label by simply calling the `IItemLabelProvider.getText()` method on the same item provider (that is, the method that previously was called for all columns). For column 1, we first need to adapt the customer before we call `getText()` on its label provider.

We implement the `getColumnImage()` method similarly:

```
public Object getColumnImage(Object object, int columnIndex) {
    if (columnIndex == 0) return getImage(object);
    Customer customer = ((PurchaseOrder)object).getCustomer();
    if (customer == null)
        return getResourceLocator().getImage("full/obj16/Customer");
    IItemLabelProvider itemLabelProvider =
        (IItemLabelProvider)adapterFactory.adapt(
            customer, IItemLabelProvider.class);
    return itemLabelProvider.getImage(customer);
}
```

Finally, we need to add one more supported type to the constructor of the item provider adapter factory, like this:

```
/**
 * This constructs an instance.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated NOT
```

```

/*
public EPO1ItemProviderAdapterFactory() {
    supportedTypes.add(IItemContentProvider.class);
    supportedTypes.add(ITreeItemContentProvider.class);
    supportedTypes.add(IItemPropertySource.class);
    supportedTypes.add(IErasingDomainItemProvider.class);
    supportedTypes.add(IItemLabelProvider.class);
    supportedTypes.add(ITableItemLabelProvider.class);
}

```

With these changes in place, a table viewer, when populated with a list of purchase orders, can display each purchase order with its associated customer as desired. So, to finish our enhancement, we now need to populate our generated editor's table viewer with a set of purchase orders.

By default, the contents of the generated table viewer is selection driven; it displays the children of the current selection in the outline view. We're going to change it so that the content is the complete set of purchase orders, regardless of the current selection. We'll also change *its* selection to match the outline view's selection. Notice that this is how the generated tree viewer under the **Selection** tab works.

To make our change, we first need to override the `createPages()` method in class `EPO1Editor` to populate the table viewer with the list of purchase orders. There are several ways that we could do this (including using class `ItemProvider` as we did for the `List` view) but the simplest approach is to subclass the `AdapterFactoryContentProvider` that we use for the table viewer in the generated editor. While we're doing this, we'll also change the viewer's column headings. Here's our modified version of the generated `createPages()` method:

```

/*
 * @generated NOT
 */
public void createPages() {
    ...
    TableColumn objectColumn = new TableColumn(table, SWT.NONE);
    layout.addColumnData(new ColumnWeightData(3, 100, true));
    objectColumn.setText(getString("_UI_ObjectColumn_label"));
    objectColumn.setText("Purchase Order");
    objectColumn.setResizable(true);

    TableColumn selfColumn = new TableColumn(table, SWT.NONE);
    layout.addColumnData(new ColumnWeightData(2, 100, true));
    selfColumn.setText(getString("_UI_SelfColumn_label"));
    selfColumn.setText("Customer");
    selfColumn.setResizable(true);

    tableViewer.setColumnProperties(new String [] {"a", "b"});
    tableViewer.setContentProvider(
        new AdapterFactoryContentProvider(adapterFactory)
    {
        public Object [] getElements(Object object) {
            return ((Supplier)object).getOrders().toArray();
        }
        public void notifyChanged(Notification notification)
        {
            switch (notification.getEventType()) {
                case Notification.ADD:
                case Notification.ADD_MANY:
                    if (notification.getFeature() !=
                        EPO1Package.eINSTANCE.getSupplier_Orders()) return;
            }
            super.notifyChanged(notification);
        }
    });
    tableViewer.setLabelProvider(
        new AdapterFactoryLabelProvider(adapterFactory));

    Resource resource =

```

```

(Resource)editingDomain.getResourceSet().getResources().get(0);
Object rootObject = resource.getContents().get(0);
if (rootObject instanceof Supplier) {
    tableViewer.setInput(rootObject);
    viewerPane.setTitle(rootObject);
}

createContextMenuFor(tableViewer);
...
}

```

As you can see, in addition to changing the two table column headings (text), we've overridden the `AdapterFactoryContentProvider getElements()` method to only return purchase orders (that is, the `orders` feature on class `Supplier`) and the `notifyChanged()` method to only pass on changes to the `orders` feature. Notice how we assume that the only object that will ever be passed to the `getElements()` method will be an instance of class `Supplier`. We can do this because we've also added the call to `setInput()` to guarantee it. We've made sure that the root object is an instance of class `Supplier`, before calling `setInput()`. If it's not, we'll simply leave the table view empty.

We need to make one more change to prevent the table viewer's input from being changed to reflect the current selection. We do this by changing the `handleContentOutlineSelection()` method like this:

```

public void handleContentOutlineSelection(ISelection selection) {
    ...
    if (currentViewerPane.getViewer() == selectionViewer ||
        currentViewerPane.getViewer() == tableViewer) {
        ArrayList selectionList = new ArrayList();
        selectionList.add(selectedElement);
        while (selectedElements.hasNext()) {
            selectionList.add(selectedElements.next());
        }

        // Set the selection to the widget.
        //
        selectionViewer.setSelection(
            new StructuredSelection(selectionList));
        currentViewerPane.getViewer().setSelection(
            new StructuredSelection(selectionList));
    }
    ...
}

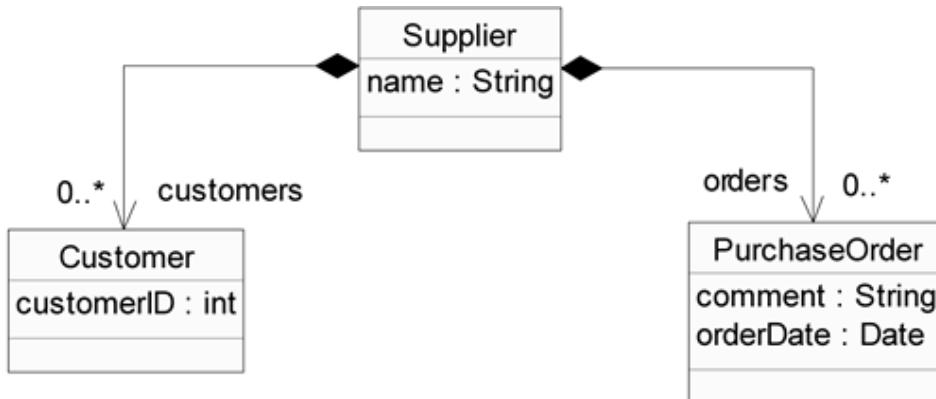
```

Notice that, as we said earlier, we're changing the table viewers selection behavior to be the same way as the `Selection` view (that is, `selectionViewer`).

Adding Non-Model Intermediary View Objects

Another common EMF.Edit customization involves introducing non-model view objects between an object and its children. Model classes often include more than one containment reference, but sometimes the default item provider implementation, which displays them as one flat heterogeneous list of children, is not desirable. Unlike our solution for the `shipTo` and `billTo` references in Section 14.2.1, suppressing some of them is often not an option. Grouping them, using intermediary objects in the view, is a good alternative in this situation. The ExtendedPO1 model is a good example.

Recall that in the ExtendedPO1 model, class `Supplier` includes two kinds of children, customers and purchase orders (Figure 14.11):

Figure 14.11. Children of class **Supplier**.

Instead of suppressing one or the other, we'd like both associations to be displayed as children, but in this example, we will introduce “Orders” and “Customers” nodes into the view to separate the two categories of children, as shown in Figure 14.12.

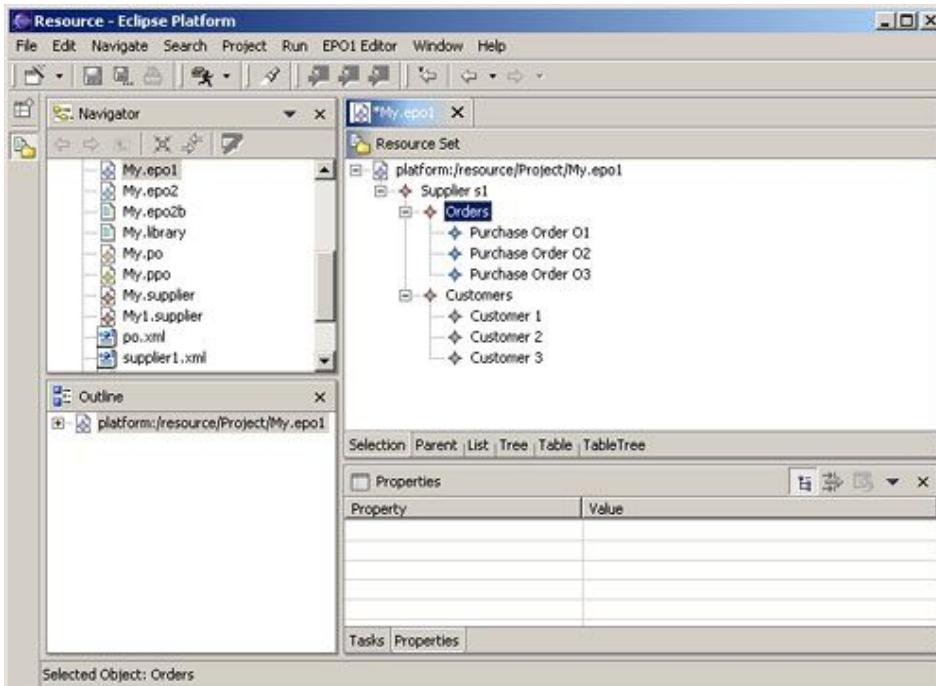


Figure 14.12. Adding non-modeled nodes in a view.

Adding such objects in a view is really quite simple, although there is some extra work involved to keep all the commands (especially drag-and-drop) working properly. Even so, it's still not that hard, as we'll see below.

As you might have guessed, the first change we need to make is to override the `getChildren()` method in the item provider for class **Supplier**. Assuming generated item providers, we can simply add the following to class **SupplierItemProvider**:

```

protected List children = null;

public Collection getChildren(Object object) {
    if (children == null) {
  
```

```

Supplier supplier = (Supplier)object;
children = new ArrayList();
children.add(new OrdersItemProvider(adapterFactory, supplier));
children.add(new CustomersItemProvider(adapterFactory,
                                         supplier));
}
return children;
}

```

Here we're overriding the `getChildren()` method to return instances of two item provider classes, `OrdersItemProvider` and `CustomersItemProvider`, which we will implement by hand. Recall from Chapter 3 that the `getChildren()` method is supposed to return objects from the model, not item providers. The item providers are usually retrieved from these returned objects when the content and label providers call the `adapt()` method on the adapter factory. In our example, however, there are no "Orders" or "Customers" model classes, so we simply return item providers instead. This all hangs together because of the default implementation of `adapt()` (in class `AdapterFactoryImpl`):

```

public Object adapt(Object object, Object type){
    if (object instanceof Notifier)
        return this.adapt((Notifier)object, type);
    else
        return object;
}

```

Notice that if the object being adapted is not an EMF (`Notifier`) object, it simply returns the object itself. With this design, an item provider—like `Supplier`—that wants to add non-model objects to a view can simply return instances of any class that implements the required item provider interfaces itself. This is the secret to displaying a mixture of EMF (modeled) and non-modeled objects in the same view.

Before we move on to look at the implementation of our two new item provider classes, there is one more thing worth mentioning about `SupplierItemProvider`. Notice in the above code fragment that we also added an instance variable, `children`, to maintain the children collection. As a result, we should regenerate the supplier item provider using the Stateful pattern (instead of the usually preferred Singleton pattern), which we described in Chapter 10. Actually, it's not strictly necessary in this particular example because in the ExtendedPO1 editor there can only ever be one instance of class `Supplier` (at the root) anyway. In the general case, however, you should always make sure to use the Stateful pattern whenever you add instance variables to an item provider class.

Let's move on and look at our new item providers. Actually, they both use exactly the same pattern so we really just need to look at one of them to understand the design. We'll choose `OrdersItemProvider` for no particular reason:

```

public class OrdersItemProvider
    extends TransientSupplierItemProvider
{
    public OrdersItemProvider(AdapterFactory adapterFactory,
                             Supplier supplier) {
        super(adapterFactory, supplier);
    }

    public Collection getChildrenReferences(Object object) {
        if (childrenReferences == null) {
            super.getChildrenReferences(object);
            childrenReferences.add(
                EPO1Package.eINSTANCE.getSupplier_Orders());
        }
        return childrenReferences;
    }

    public String getText(Object object) {

```

```

        return "Orders";
    }

    public void notifyChanged(Notification notification) {
        switch (notification.getFeatureID(Supplier.class)) {
            case EPO1Package.SUPPLIER__ORDERS:
                fireNotifyChanged(
                    new NotificationWrapper(this, notification));
                return;
        }
    }

    protected void collectNewChildDescriptors(
        Collection newChildDescriptors,
        Object object) {
        super.collectNewChildDescriptors(newChildDescriptors, object);
        newChildDescriptors.add(createChildParameter(
            EPO1Package.eINSTANCE.getSupplier_Orders(),
            EPO1Factory.eINSTANCE.createPurchaseOrder()));
    }
}

```

Notice how it looks very similar to a generated item provider. In fact, if you compare these methods with the generated ones in class `SupplierItemProvider`, you'll notice that they're implementing the `orders`-specific subset of each method. For example, `getChildrenReferences()` looks like this in `SupplierItemProvider`:

```

public Collection getChildrenReferences(Object object) {
    if (childrenReferences == null) {
        super.getChildrenReferences(object);
        childrenReferences.add(
            EPO1Package.eINSTANCE.getSupplier_Customers());
        childrenReferences.add(
            EPO1Package.eINSTANCE.getSupplier_Orders());
    }
    return childrenReferences;
}

```

As you can see, the difference is the elimination of the `customers` part, which makes perfect sense since this is the "Orders" node, which shouldn't know anything about the `customers` reference.

You may have noticed that some of the methods in class `OrdersItemProvider`, like `getChildrenReferences()`, are template methods from class `ItemProviderAdapter`, as described in Chapter 10. You may recall from Chapters 3 and 10 how class `ItemProviderAdapter` is used as the convenient base class for item providers that are adapters for EMF model objects. This may seem surprising, since we already pointed out how there are no modeled classes for "Orders" or "Customers". So what model objects are they adapting?

If you think about it, although not the "official" item providers for class `Supplier`, `CustomersItemProvider` and `OrdersItemProvider` can really be thought of as providers of a subset of a supplier. Much of their implementation works in the usual way using the supplier model object. Also, registering them as observers of the supplier is necessary to provide their change notification. So, the bottom line is that we simply have three item providers attached to the supplier model object; the first (that is, `SupplierItemProvider`) is the real item provider for the supplier, while the other two are simply attached as an implementation convenience.

You may have noticed that the base class of `OrdersItemProvider` is not actually `ItemProviderAdapter`, but instead another class, `TransientSupplierItemProvider`. `TransientSupplierItemProvider` is another hand-coded convenient base class, which contains a few methods that are shared by `OrdersItemProvider` and `CustomersItemProvider`. It is, itself, a typical subclass of `ItemProviderAdapter` that mixes in the standard item provider interfaces:

```
public class TransientSupplierItemProvider
    extends ItemProviderAdapter
    implements
        IEditingDomainItemProvider,
        IStructuredItemContentProvider,
        ITreeItemContentProvider,
        IItemLabelProvider,
        IItemPropertySource
{
    ...
}
```

Because the “Orders” and “Customers” item providers are not created in the usual way (that is, by calling `adapt()` for the supplier), their constructors explicitly add them to the `eAdapters` list like this:

```
public TransientSupplierItemProvider(AdapterFactory adapterFactory,
                                     Supplier supplier) {
    super(adapterFactory);
    supplier.eAdapters().add(this);
}
```

Most of the other methods in `TransientSupplierItemProvider` are simple method overrides that substitute arguments before calling the super version of the same method. For example, `getChildren()` looks like this:

```
public Collection getChildren(Object object) {
    return super.getChildren(target);
}
```

Notice how the `object` argument is replaced with the target of the adapter. When `getChildren()` is called, the `object` argument will be the item provider itself, because the transient item providers are masquerading as model objects as well. The `target` instance variable comes from the adapter base class `AdapterImpl`, and in this case will be the supplier that the adapter was attached to; it's the real EMF model object from which the children will be retrieved.

The `getParent()` method is simply overridden to return the `target`, like this:

```
public Object getParent(Object object) {
    return target;
}
```

Now that we've changed `SupplierItemProvider` to return our transient item providers as its children, and our transient providers are implemented to return the supplier as their parent and the **orders** or **customers** as their children, we need to make one more parent change. The `getParent()` methods in `PurchaseOrderItemProvider` and `CustomerItemProvider` need to return the appropriate transient item provider. We use exactly the same pattern in both classes. In `PurchaseOrderItemProvider`, for example, it looks like this:

```
public Object getParent(Object object) {
    Object supplier = super.getParent(object);
    SupplierItemProvider supplierItemProvider =
        (SupplierItemProvider)adapterFactory.adapt(
            supplier, IEditingDomainItemProvider.class);

    return supplierItemProvider != null ?
        supplierItemProvider.getOrders() : null;
}
```

Notice how in the non-null case it calls the `getOrders()` method on the supplier item provider. By adding two convenience methods, `getOrders()` and `getCustomers()`, on the supplier item provider we avoid adding state to `PurchaseOrderItemProvider` and `CustomerItemProvider` and can continue to use the Singleton pattern for them. The `getOrders()` method in class `SupplierItemProvider` is very simple and looks like this:

```
public Object getOrders() {
    return children.get(0);
}
```

As you might expect, `getCutomers()` is the same only it returns `children.get(1)` and it is used for the implementation of `getParent()` in class `CustomerItemProvider`.

Now that we've overridden all the applicable `getChildren()` and `getParent()` methods, we can run the ExtendedPO1 editor and the tree view will look as shown in Figure 14.12. There are however, still a few details that we'll need to address unless we want to disable command execution entirely (by simply overriding the `createCommand()` method to return `UnexecutableCommand.INSTANCE`, for example). Since fixing them isn't really that hard, and we're trying to learn about the framework anyway, we'll just bite the bullet and fix all the problems.

The first problem that we need to address is that the (generated) `SupplierItemProvider` adds new child descriptors for purchase orders and customers to itself. Since we don't want the `CreateChildCommand` to be enabled on the supplier node (that is, we should only be adding new purchase orders to the "Orders" node and new customers to the "Customers" node), we need to take over the `collectNewChildDescriptors()` method in `SupplierItemProvider`:

```
/** 
 * @generated NOT
 */
protected void collectNewChildDescriptors(...) {
    super.collectNewChildDescriptors(newChildDescriptors, object);
    newChildDescriptors.add(createChildParameter(
        EPO1Package.eINSTANCE.getSupplier_Customers(),
        EPO1Factory.eINSTANCE.createCustomer()));
    newChildDescriptors.add(createChildParameter(
        EPO1Package.eINSTANCE.getSupplier_Orders(),
        EPO1Factory.eINSTANCE.createPurchaseOrder()));
}
```

The second problem that we need to address is the same problem that we handled in Section 14.2.1 when we suppressed the **USAddress** objects from the view, but then needed to override the `getAffectedObjects()` method on commands involving them, to return the purchase order instead. In this case, it's not the `SetCommand`, but rather `AddCommand` and `RemoveCommand` that have this problem.

After removing a purchase order or customer from a supplier, we want the affected object (that is, the selection) to be the appropriate transient node, instead of the supplier. We also want this to be the case if we undo an add command. To make it work in all cases (remember that `AddCommand` and `RemoveCommand` are primitive commands used for a lot of things, including drag-and-drop) we need to override `AddCommand` and `RemoveCommand` in both the `TransientItemProvider` and `SupplierItemProvider` to switch the affected objects. In `SupplierItemProvider`, we do it like this:

```
protected Command createRemoveCommand(EditingDomain domain,
                                     EObject owner,
                                     EReference feature,
                                     Collection collection) {
    return createWrappedCommand(
        super.createRemoveCommand(domain, owner, feature, collection)),
```

```

        owner, feature);
    }

protected Command createAddCommand(EditingDomain domain,
                                  EObject owner,
                                  EReference feature,
                                  Collection collection,
                                  int index) {
    return createWrappedCommand(
        super.createAddCommand(domain, owner, feature, collection,
                               index),
        owner, feature);
}

protected Command createWrappedCommand(Command command,
                                       final EObject owner,
                                       final EReference feature) {
    if (feature == EPO1Package.eINSTANCE.getSupplier_Orders() ||
        feature == EPO1Package.eINSTANCE.getSupplier_Customers())
        return
            new CommandWrapper(command)
        {
            public Collection getAffectedObjects() {
                Collection affected = super.getAffectedObjects();
                if (affected.contains(owner))
                    affected = Collections.singleton(
                        feature == EPO1Package.eINSTANCE.getSupplier_Orders()
                            ? getOrders() : getCustomers());
                return affected;
            }
        };
    else
        return command;
}

```

As you can see, for the **orders** and **customers** features, we create an anonymous command subclass that overrides the `getAffectedObjects()` method to return the appropriate transient item provider whenever the “real” affected object is the `owner` (that is, the supplier). Notice how we used the EMF.Edit convenience class `CommandWrapper` so that we could handle both the `AddCommand` and `RemoveCommand` cases with a single implementation.

As mentioned above, we need to also include this affected object correction code in class `TransientItemProvider`. There, however, the `createWrappedCommand()` method can be a little simpler:

```

protected Command createWrappedCommand(Command command,
                                       final EObject owner) {
    return
        new CommandWrapper(command)
    {
        public Collection getAffectedObjects() {
            Collection affected = super.getAffectedObjects();
            if (affected.contains(owner))
                affected = Collections.singleton(
                    TransientSupplierItemProvider.this);
            return affected;
        }
    };
}

```

Notice that here we don't need to check which feature we're dealing with, because each of our two transient classes only supports one feature (that is, `OrdersItemProvider` only supports the **orders** feature while `CustomersItemProvider` only supports **customers**).

We're on the final stretch now. With the changes we've made so far, only one more questionable behavior will remain, involving drag-and-drop. With our current implementation, you can drag a purchase order and drop it on the "Customers" node, or drag a customer and drop it on the "Orders" node. The effect will be to add the object to the end of the other list. It won't corrupt the model, but it certainly doesn't seem right from the user's perspective. This happens because, as we've seen above, class `TransientItemProvider` implements most of its methods by effectively redirecting them to the adapter's target (supplier). This is also the case for the `createCommand()` method, so add commands on either the "Orders" or "Customers" node are simply converted to add commands on the supplier, which then enable the drop, regardless of the user's actual chosen drop target.

To prevent this final problem from occurring, we need to override the `createDragAndDropCommand()` method in our two transient item providers to disallow dropping of the wrong kind of object. In `OrdersItemProvider`, we need to make sure that only purchase orders are being dropped, while in `CustomersItemProvider`, there should only be customers. Here's how we do it in `OrdersItemProvider`:

```
protected Command createDragAndDropCommand(EditDomain domain,
                                         Object owner, float location, int operations,
                                         int operation, Collection collection) {
    if (new AddCommand(domain, (EObject)owner,
                       EPO1Package.eINSTANCE.getSupplier_Orders(),
                       collection).canExecute())
        return super.createDragAndDropCommand(domain, owner, location,
                                             operations, operation,
                                             collection);
    else
        return UnexecutableCommand.INSTANCE;
}
```

Notice how we decided to use an `AddCommand` as a convenient way to verify that all the objects in the collection of drop items are purchase orders (that is, the correct target type for the `orders` feature); the `AddCommand`'s enablement check is exactly this. If the `canExecute()` method returns `false`, then some or all of the objects being dropped are not purchase orders, so we simply disable the drag-and-drop command by returning `UnexecutableCommand.INSTANCE`. Now all we need to do is add the same override in `CustomersItemProvider`, only using the `customers` feature instead of the `orders` feature in the `AddCommand`, and we're done. The ExtendedPO1 editor will now correctly implement all of the commands.

As you can see, there were quite a few details that we needed to address to get this example working, but each particular problem had a fairly simple solution. The total amount of code that we needed to add was only about a hundred and fifty lines, but only because we knew how to use the framework effectively. Now that we've painstakingly walked through this example, you should be all set to attack your own customization problems and come up with short and elegant implementations for them as well.

Part V. EMF API

Chapter 15. The org.eclipse.emf.common Plug-In

The `org.eclipse.emf.common` plug-in provides the command and notification frameworks, as well as a collection of utility classes, that are used throughout the other EMF plug-ins.

Although this plug-in declares a dependency on the `org.eclipse.core.runtime` plug-in, this is, in fact, a soft dependency. Only one class uses it, if available, to access certain facilities provided by the Eclipse platform; otherwise, alternate implementations are used. The dependency must be declared, however, as `org.eclipse.core.runtime` is required to build this plug-in.

Because it has no other dependencies, this plug-in can be reused easily in non-EMF applications.

Requires: `org.eclipse.core.runtime(org.apache.xerces)`

The `org.eclipse.emf.common` Package

The `org.eclipse.emf.common` package provides `EMFPlugin`, the base class for EMF plug-in classes that act as the central providers of text and image resources and logging capabilities to all classes in their respective plug-ins. It also includes one of these, `CommonPlugin`, for the `org.eclipse.emf.common` plug-in itself.

`CommonPlugin`

`org.eclipse.emf.common`

`CommonPlugin` is the plug-in class for the `org.eclipse.emf.common` plug-in. Like all other EMF plug-in classes, it extends from `EMFPlugin` and uses the Singleton design pattern, with its single instance available as the constant `INSTANCE`.

When running within Eclipse, the Eclipse plug-in class, of type `CommonPlugin.Implementation`, is available from the static `getPlugin()` method. The same object is returned by `getPluginResourceLocator()` and is used as a delegate by the `EMFPlugin` implementations of `getBaseURL()`, `getImage()`, `getString()`, and `log()`, to make use of the platform's plug-in support facilities. When running stand-alone, `getPluginResourceLocator()` returns `null`, and the base class uses its own implementations.

```
public final class CommonPlugin extends EMFPlugin {
    // No Constructor
    // Public Constants
    public static final CommonPlugin INSTANCE;
    // Public Inner Classes
    public static class Implementation;
    // Public Class Methods
```

```

    public static CommonPlugin.Implementation getPlugin();
    // Property Accessor Methods (by property name)
    public ResourceLocator getPluginResourceLocator();                                Overrides:EMFPlugin
}


```

Hierarchy: Object → EMFPlugin(ResourceLocator, Logger) → CommonPlugin

CommonPlugin.Implementation

org.eclipse.emf.common

CommonPlugin.Implementation extends org.eclipse.core.runtime.Plugin, the base class that represents a plug-in within Eclipse, via EMFPlugin.EclipsePlugin. It is instantiated by the platform, and then used to provide platform-based implementations for most of the methods of the CommonPlugin plug-in class. It is also through this class that other Eclipse facilities, including the preference store and plug-in descriptor, can be accessed.

```

public static class CommonPlugin.Implementation extends EMFPlugin.EclipsePlugin {
    // Public Constructors
    public Implementation(IPluginDescriptor descriptor);
}

```

Hierarchy: Object → Plugin → EMFPlugin.EclipsePlugin(ResourceLocator, Logger) → CommonPlugin.Implementation

EMFPlugin

org.eclipse.emf.common

EMFPlugin is the abstract base class for all EMF plug-in classes, which provide text and image resources and logging capabilities to the other classes in an EMF plug-in. These capabilities are provided by the getString(), getImage(), and log() methods, respectively. When running within Eclipse, these methods delegate to an instance of EMFPlugin.EclipsePlugin, which defines platform-based implementations. When running stand-alone, it provides default implementations that log to standard error and look for a *plugin.properties* file and *icons*/directory on the class path, under the directory corresponding to the package to which a given subclass belongs—for example, *org/eclipse/emf/common/for* CommonPlugin. If a resource is not found, getString() and getImage() then attempt to delegate to any other ResourceLocators passed to the constructor.

The pattern used by a typical EMFPlugin subclass is described in Section 10.6.

```

public abstract class EMFPlugin implements ResourceLocator, Logger {
    // Public Constructors
    public EMFPlugin(ResourceLocator[] delegateResourceLocators);
    // Public Inner Classes
    public static abstract class EclipsePlugin;
    // Property Accessor Methods (by property name)
    public URL getBaseUrl();                                              Implements:ResourceLocator
    public Logger getPluginLogger();
    public abstract ResourceLocator getPluginResourceLocator();
    // Methods Implementing ResourceLocator
    public Object getImage(String key);
    public String getString(String key);
    public String getString(String key, Object[] substitutions);
    // Methods Implementing Logger
    public void log(Object logEntry);
    // Protected Instance Methods
    protected Object delegatedGetImage(String key);
    protected String delegatedGetString(String key);
}

```

```

protected Object doGetImage(String key);
// Protected Instance Fields
protected URL baseURL;
protected ResourceLocator[] delegateResourceLocators;
protected Map images;
protected ResourceBundle resourceBundle;
protected Map strings;
}

```

Hierarchy: Object → EMFPlugin(ResourceLocator, Logger)

EMFPlugin.EclipsePlugin

org.eclipse.emf.common

EMFPlugin.EclipsePlugin is the abstract base class for EMF plug-in classes' Eclipse-based delegates. As a subclass of org.eclipse.core.runtime.Plugin, which Eclipse uses to represent a plug-in, it has access to the plug-in descriptor and log. It uses these standard Eclipse facilities to implement getBaseURL(), getString(), getImage(), and log().

```

public static abstract class EMFPlugin.EclipsePlugin extends Plugin implements ←
ResourceLocator, Logger {
// Public Constructors
public EclipsePlugin(IPluginDescriptor descriptor);
// Property Accessor Methods (by property name)
public URL getBaseUrl();                                Implements:ResourceLocator
// Methods Implementing ResourceLocator
public Object getImage(String key);
public String getString(String key);
public String getString(String key, Object[] substitutions);
// Methods Implementing Logger
public void log(Object logEntry);
// Protected Instance Methods
protected Object doGetImage(String key);
}

```

Hierarchy: Object → Plugin → EMFPlugin.EclipsePlugin(ResourceLocator, Logger)

The org.eclipse.emf.common.command Package

The org.eclipse.emf.common.command package provides the common command framework that is extended and used by EMF.Edit, as described in Chapter 3. It provides the basic command interface and several convenience implementations of it. It also includes interfaces for a command stack and for an object that listens for changes to one.

Although its primary use is in EMF.Edit, this framework is completely generic, in that it deals in collections of arbitrary objects.

AbstractCommand

org.eclipse.emf.common.command

AbstractCommand is the abstract base from which all other EMF commands, including those defined in this package and by EMF.Edit, derive. When extending this class, overrides should be provided for most methods. However, a fully functional CompoundCommand-based implementation is provided for chain(). Also, the implementation of canExecute() calls out only once to prepare(), caching the result for its future invocations. This is useful when testing enablement requires significant computation. Thus, derived classes should override prepare() instead.

```

public abstract class AbstractCommand implements Command {
    // Protected Constructors
    protected AbstractCommand();
    protected AbstractCommand(String label);
    protected AbstractCommand(String label, String description);
    // Property Accessor Methods (by property name)
    public Collection getAffectedObjects();
    Implements:Command
    public String getDescription();
    public void setDescription(String description);                                empty
    public String getLabel();
    public void setLabel(String label);
    public Collection getResult();
    Implements:Command
    // Public Methods Overriding Object
    public String toString();
    // Methods Implementing Command
    public boolean canExecute();
    public boolean canUndo();
    public Command chain(Command command);
    public void dispose();empty
    public void undo();
    // Protected Instance Methods
    protected boolean prepare();
    // Protected Instance Fields
    protected String description;
    protected boolean isExecutable;
    protected boolean isPrepared;
    protected String label;
}

```

Hierarchy: Object → AbstractCommand (Command)

BasicCommandStack

org.eclipse.emf.common.command

The **BasicCommandStack** class is a simple implementation of **CommandStack**. It also adds two methods that are not specified by the interface: **saveIsDone()** and **isSaveNeeded()**. The former is meant to be called after the model being edited is saved; it records the current position in the stack. Then, the latter method indicates whether the model has changed, requiring it to be saved again. This assumes that the model is only modified via commands on the stack.

```

public class BasicCommandStack implements CommandStack {
    // Public Constructors
    public BasicCommandStack();
    // Property Accessor Methods (by property name)
    public Command getMostRecentCommand();
    public Command getRedoCommand();
    public boolean isSaveNeeded();
    public Command getUndoCommand();
    // Public Instance Methods
    public void saveIsDone();
    // Methods Implementing CommandStack
    public void addCommandStackListener(CommandStackListener listener);
    public boolean canRedo();
    public boolean canUndo();
    public void execute(Command command);
    public void flush();
    public void redo();
    public void removeCommandStackListener(CommandStackListener listener);
    public void undo();
    // Protected Instance Methods
    protected void notifyListeners();
    // Protected Instance Fields
    protected List commandList;
    protected Collection listeners;
}

```

```

protected Command mostRecentCommand;
protected int saveIndex;
protected int top;
}

```

Hierarchy: Object → BasicCommandStack (CommandStack)

Command

org.eclipse.emf.common.command

The `Command` interface defines the behavior of an EMF command. A command should carry out its function in `execute()`, which can only be called if `canExecute()` returns `true`. A command can also offer undo and redo functionality via `undo()` and `redo()`. These can only be called if `canUndo()` and `canExecute()`, respectively, return `true` and must be called in the expected order; that is, `undo()` after `execute()` or `redo()`, and `redo()` after `undo()`.

The `getResult()` method returns whatever the command considers to be its result, and can be used in command composition to supply input to a following command. The `getAffectedObjects()` method returns the set of affected objects that best illustrate the effect of the last execution, redo or undo. This might or might not be the same as the result. The `chain()` method returns a command representing the composition of the command with another given command.

```

public interface Command {
    // Property Accessor Methods (by property name)
    public abstract Collection getAffectedObjects();
    public abstract String getDescription();
    public abstract String getLabel();
    public abstract Collection getResult();
    // Public Instance Methods
    public abstract boolean canExecute();
    public abstract boolean canUndo();
    public abstract Command chain(Command command);
    public abstract void dispose();
    public abstract void execute();
    public abstract void redo();
    public abstract void undo();
}

```

CommandStack

org.eclipse.emf.common.command

`CommandStack` defines the interface for executing and maintaining commands in an undoable command stack. A command is executed and added to the top of the stack by `execute()`. The `undo()` and `redo()` methods move down and up the stack, respectively, undoing and redoing commands. The next commands to be undone and redone are returned by calling `getUndoCommand()` and `getRedoCommand()`, and `getMostRecentCommand()` returns the last command to be executed, undone, or redone.

A listener can be notified whenever a command has been processed. The `addCommandStackListener()` method adds such a listener to the stack.

```

public interface CommandStack {
    // Property Accessor Methods (by property name)
    public abstract Command getMostRecentCommand();
    public abstract Command getRedoCommand();
    public abstract Command getUndoCommand();
    // Public Instance Methods
}

```

```

public abstract void addCommandStackListener(CommandStackListener listener);
public abstract boolean canRedo();
public abstract boolean canUndo();
public abstract void execute(Command command);
public abstract void flush();
public abstract void redo();
public abstract void removeCommandStackListener(CommandStackListener listener);
public abstract void undo();
}

```

CommandStackListener

org.eclipse.emf.common.command

The `CommandStackListener` interface must be implemented by any class that is to be notified when a command has been processed by a stack. Notification is performed by calling the `commandStackChanged()` method, passing a standard Java `EventObject`, whose source is the command stack.

```

public interface CommandStackListener {
    // Public Instance Methods
    public abstract void commandStackChanged(EventObject event);
}

```

CommandWrapper

org.eclipse.emf.common.command

`CommandWrapper` is a convenience class that delegates the implementation of `Command` to another wrapped command. Typically, an instance of `CommandWrapper` is used as a decorator, to modify the behavior of a command without directly subclassing it, or as a proxy for a command whose creation is delayed. In the former case, the wrapped command is specified as an argument to the constructor, and behavior is modified by overriding the delegating `CommandWrapper` methods. In the latter, `createCommand()` is overridden to create the desired command; this method is called by `prepare()`.

```

public class CommandWrapper extends AbstractCommand {
    // Public Constructors
    public CommandWrapper(Command command);
    public CommandWrapper(String label, String description, Command command);
    // Protected Constructors
    protected CommandWrapper(String label, Command command);
    protected CommandWrapper();
    protected CommandWrapper(String label);
    protected CommandWrapper(String label, String description);
    // Property Accessor Methods (by property name)
    public Collection getAffectedObjects();                                Overrides:AbstractCommand
    public Command getCommand();                                              Overrides:AbstractCommand
    public String getDescription();                                            Overrides:AbstractCommand
    public String getLabel();                                                 Overrides:AbstractCommand
    public Collection getResult();                                             Overrides:AbstractCommand
    // Public Methods Overriding AbstractCommand
    public boolean canUndo();                                                 constant
    public void dispose();
    public String toString();
    public void undo();
    // Methods Implementing Command
    public void execute();
    public void redo();
    // Protected Instance Methods
    protected Command createCommand();                                         constant
}

```

```

protected boolean prepare();
// Protected Instance Fields
protected Command command;
}

```

Overrides:AbstractCommand

Hierarchy: Object → AbstractCommand (Command) → CommandWrapper

CompoundCommand

org.eclipse.emf.common.command

CompoundCommand is a convenience class for building higher level commands as compositions of more basic commands. A list of component commands can be passed to the constructor, and individual commands can be added with `append()`. Calling `execute()` or `redo()` on the compound command calls the same method on each of its component commands in order; similarly, `undo()` is delegated to each of the component commands in reverse order. A compound command `canExecute()` or `canUndo()` only if all of its component commands can.

The `getResult()` and `getAffectedObjects()` methods delegate to the component command indexed by the `resultIndex` constructor argument. `LAST_COMMAND_ALL` can be specified to delegate to the last command; by default, or if `MERGE_COMMAND_ALL` is specified, the results from delegating to every component command are merged into one collection.

For convenience, `appendAndExecute()` appends a command to the compound command and immediately executes it, which is a good way of recording a set of executed commands that can later be undone as a unit. The `unwrap()` method can be used to optimize a compound command built up conditionally. If a compound command contains no component commands, `unwrap()` returns the singleton instance of `UnexecutableCommand`; if it contains a single component command, `unwrap()` returns that component; otherwise, `unwrap()` returns the compound command itself.

```

public class CompoundCommand extends AbstractCommand {
// Public Constructors
public CompoundCommand();
public CompoundCommand(String label);
public CompoundCommand(String label, String description);
public CompoundCommand(List commandList);
public CompoundCommand(String label, List commandList);
public CompoundCommand(String label, String description, List commandList);
public CompoundCommand(int resultIndex);
public CompoundCommand(int resultIndex, String label);
public CompoundCommand(int resultIndex, String label, String description);
public CompoundCommand(int resultIndex, List commandList);
public CompoundCommand(int resultIndex, String label, List commandList);
public CompoundCommand(int resultIndex, String label, String description, List commandList);
// Public Constants
public static final int LAST_COMMAND_ALL;                                     ==-2147483648
public static final int MERGE_COMMAND_ALL;                                    ==2147483647
// Property Accessor Methods (by property name)
public Collection getAffectedObjects();                                         Overrides:AbstractCommand
public String getDescription();                                              Overrides:AbstractCommand
public boolean isEmpty();                                                 Overrides:AbstractCommand
public String getLabel();                                                 Overrides:AbstractCommand
public Collection getResults();                                             Overrides:AbstractCommand
// Public Instance Methods
public void append(Command command);
public boolean appendAndExecute(Command command);
public boolean appendIfCanExecute(Command command);
public Command unwrap();
// Public Methods Overriding AbstractCommand
public boolean canUndo();
public void dispose();

```

```

public String toString();
public void undo();
// Methods Implementing Command
public void execute();
public void redo();
// Protected Instance Methods
protected Collection getMergedAffectedObjectsCollection();
protected Collection getMergedResultCollection();
protected boolean prepare();
// Protected Instance Fields
protected List commandList;
protected int resultIndex;
}

Overrides:AbstractCommand

```

Hierarchy: Object → AbstractCommand(Command) → CompoundCommand

IdentityCommand

org.eclipse.emf.common.command

IdentityCommand implements a Command that does nothing. Its canExecute() and canUndo() methods always return true, and its execute(), undo(), and redo() methods are empty. A result can be specified as a constructor argument and is then returned by getResult().

A default instance of IdentityCommand, with no result, is available as INSTANCE.

```

public class IdentityCommand extends AbstractCommand {
// Public Constructors
public IdentityCommand();
public IdentityCommand(Object result);
public IdentityCommand(Collection result);
public IdentityCommand(String label);
public IdentityCommand(String label, Object result);
public IdentityCommand(String label, Collection result);
public IdentityCommand(String label, String description);
public IdentityCommand(String label, String description, Object result);
public IdentityCommand(String label, String description, Collection result);
// Public Constants
public static final IdentityCommand INSTANCE;
// Property Accessor Methods (by property name)
public String getDescription();
public String getLabel();
public Collection getResult();
Overrides:AbstractCommand
Overrides:AbstractCommand
Overrides:AbstractCommand
// Public Methods Overriding AbstractCommand
public boolean canExecute();
public void undo();
constant
empty
// Methods Implementing Command
public void execute();
public void redo();
empty
empty
// Protected Instance Fields
protected Collection result;
}

Overrides:AbstractCommand

```

Hierarchy: Object → AbstractCommand(Command) → IdentityCommand

StrictCompoundCommand

org.eclipse.emf.common.command

The `StrictCompoundCommand` class is used to compose commands when some component commands depend on the results of preceding component commands. To handle this case, `canExecute()` is implemented more carefully, actually executing each command before testing the next. The `isPessimistic` field is `false` by default, but if it is `true`, `canExecute()` finishes by undoing all component commands that it has executed.

Note that because commands are actually executed by `canExecute()`, it is important that all but the last command have no visible side effects.

```
public class StrictCompoundCommand extends CompoundCommand {
    // Public Constructors
    public StrictCompoundCommand();
    public StrictCompoundCommand(String label);
    public StrictCompoundCommand(String label, String description);
    public StrictCompoundCommand(List commandList);
    public StrictCompoundCommand(String label, List commandList);
    public StrictCompoundCommand(String label, String description, List commandList);
    // Public Methods Overriding CompoundCommand
    public boolean appendAndExecute(Command command);
    public void execute();
    public void redo();
    public String toString();
    public void undo();
    // Protected Instance Methods
    protected boolean prepare();Overrides:CompoundCommand
    // Protected Instance Fields
    protected boolean isPessimistic;
    protected boolean isUndoable;
    protected int rightMostExecutedCommandIndex;
}
```

Hierarchy: Object → AbstractCommand (Command) → CompoundCommand → StrictCompoundCommand

UnexecutableCommand

org.eclipse.emf.common.command

`UnexecutableCommand` implements a Command that can never be executed. Its `canExecute()` and `canUndo()` methods always return `false`, and `execute()`, `undo()`, and `redo()` simply throw exceptions.

This class is implemented using the Singleton design pattern; its single instance is available as the `INSTANCE` field.

```
public class UnexecutableCommand extends AbstractCommand {
    // No Constructor
    // Public Constants
    public static final UnexecutableCommand INSTANCE;
    // Public Methods Overriding AbstractCommand
    public boolean canExecute();constant
    public boolean canUndo();constant
    // Methods Implementing Command
    public void execute();
    public void redo();
}
```

Hierarchy: Object → AbstractCommand (Command) → UnexecutableCommand

The org.eclipse.emf.common.notify Package

The `org.eclipse.emf.common.notify` package provides the common notification framework that is used throughout EMF, as described in Chapter 2. It includes interfaces for notifications themselves, for the notifiers and adapters that send and receive them, and for the factories that create adapters.

Adapter

`org.eclipse.emf.common.notify`

`Adapter` specifies the interface used throughout EMF for receiving notifications of changes to model objects and for extending the behavior of those objects.

The object that an adapter receives notifications for, which must implement the `Notifier` interface, is called its target. The `setTarget()` accessor is called by the notifier when the adapter is added to its adapter list.

A notifier notifies its adapters of changes by calling their `notifyChanged()` methods.

Because notifiers can have multiple adapters, the `isAdapterForType()` method is used by an adapter factory to determine which, if any, adapter to return for a given notifier. The `type` can be any object that is meaningful for the particular adapter. When an adapter is used to support a particular interface, the corresponding instance of `java.lang.Class` might be an appropriate choice.

```
public interface Adapter {
    // Property Accessor Methods (by property name)
    public abstract Notifier getTarget();
    public abstract void setTarget(Notifier newTarget);
    // Public Instance Methods
    public abstract boolean isAdapterForType(Object type);
    public abstract void notifyChanged(Notification notification);
}
```

AdapterFactory

`org.eclipse.emf.common.notify`

The `AdapterFactory` interface defines the behavior of factories that create adapters and associate them with notifiers. To obtain an adapter for a given notifier and of a given type, `adapt()` is called. If the notifier already has the correct type of adapter associated with it, that adapter is returned; otherwise, a new adapter is created by calling `adaptNew()`, which also adds the adapter to the notifier's adapter list and calls `setTarget()` on the adapter.

Like adapters themselves, adapter factories may specify if they support a particular type, via the `isFactoryForType()` method.

```
public interface AdapterFactory {
    // Public Instance Methods
    public abstract Object adapt(Object object, Object type);
    public abstract Adapter adapt(Notifier target, Object type);
    public abstract void adaptAllNew(Notifier notifier);
    public abstract Adapter adaptNew(Notifier target, Object type);
    public abstract boolean isFactoryForType(Object type);
}
```

Notification

`org.eclipse.emf.common.notify`

The `Notification` interface defines the type of object that describes a change to a notifier. It is passed to an adapter's `notifyChanged()` method.

A notification can specify the notifier object, the type of change event, the feature that changed, the position within a list-based feature at which the change occurred, and the old and new values of the feature. `Notification` defines a number of constants for event types, such as `CREATE`, `SET`, `UNSET`, and `ADD`.

For efficiency, special primitive-typed methods are included for accessing the old and new values, avoiding the need to wrap these values in objects. Also, `getFeatureID()` can be used to return a numeric feature ID to enable use of efficient `switch` statements in adapters. Otherwise, the `NO_FEATURE_ID` constant should be returned. Section 9.6.10 describes how this method is used in notifications from EMF-modeled objects.

The `isTouch()` method specifies whether the notification represents an event in which the state of the notifier did not change; `isReset()` specifies whether a feature was set to its default value.

The `merge()` method supports merging of compatible notifications. If possible, it modifies a notification to include the information contained in the other notification, specified as a parameter, and returns `true`.

```
public interface Notification {
    // Public Constants
    public static final int ADD; =3
    public static final int ADD_MANY; =5
    public static final int EVENT_TYPE_COUNT; =10
    public static final int MOVE; =7
    public static final int NO_FEATURE_ID; =-1
    public static final int NO_INDEX; =-1
    public static final int REMOVE; =4
    public static final int REMOVE_MANY; =6
    public static final int REMOVING_ADAPTER; =8
    public static final int RESOLVE; =9
    public static final int SET; =1
    public static final int UNSET; =2

    // Property Accessor Methods (by property name)
    public abstract int getEventType();
    public abstract Object getFeature();
    public abstract boolean getNewBooleanValue();
    public abstract byte getNewByteValue();
    public abstract char getNewCharValue();
    public abstract double getNewDoubleValue();
    public abstract float getNewFloatValue();
    public abstract int getNewIntValue();
    public abstract long getNewLongValue();
    public abstract short getNewShortValue();
    public abstract String getNewStringValue();
    public abstract Object getNewValue();
    public abstract Object getNotifier();
    public abstract boolean getOldBooleanValue();
    public abstract byte getOldByteValue();
    public abstract char getOldCharValue();
    public abstract double getOldDoubleValue();
    public abstract float getOldFloatValue();
    public abstract int getOldIntValue();
    public abstract long getOldLongValue();
    public abstract short getOldShortValue();
    public abstract String getOldStringValue();
    public abstract Object getOldValue();
    public abstract int getPosition();
    public abstract boolean isReset();
}
```

```

public abstract boolean isTouch();
// Public Instance Methods
public abstract int getFeatureID(Class expectedClass);
public abstract boolean merge(Notification notification);
}

```

NotificationChain

org.eclipse.emf.common.notify

The `NotificationChain` class permits accumulation of notifications. As notifications are produced, they can be added to the chain via `add()`, before being sent by their appropriate notifiers via `dispatch()`.

```

public interface NotificationChain {
// Public Instance Methods
    public abstract boolean add(Notification notification);
    public abstract void dispatch();
}

```

NotificationWrapper

org.eclipse.emf.common.notify

`NotificationWrapper` is a convenience class, used to create a notification that delegates to another, wrapped notification. All methods are delegated, with the possible exception of `getNotifier()`, if a different notifier is specified as a constructor argument.

```

public class NotificationWrapper implements Notification {
// Public Constructors
    public NotificationWrapper(Notification notification);
    public NotificationWrapper(Object notifier, Notification notification);
// Property Accessor Methods (by property name)
    public int getEventType();                                Implements:Notification
    public Object getFeature();                               Implements:Notification
    public boolean getNewBooleanValue();                      Implements:Notification
    public byte getNewByteValue();                            Implements:Notification
    public char getNewCharValue();                           Implements:Notification
    public double getNewDoubleValue();                        Implements:Notification
    public float getNewFloatValue();                          Implements:Notification
    public int getNewIntValue();                            Implements:Notification
    public long getNewLongValue();                           Implements:Notification
    public short getNewShortValue();                         Implements:Notification
    public String getNewStringValue();                       Implements:Notification
    public Object getNewValue();                            Implements:Notification
    public Object getNotifier();                           Implements:Notification
    public boolean getOldBooleanValue();                     Implements:Notification
// Implementations
    public byte getOldByteValue();                          Implements:Notification
    public char getOldCharValue();                         Implements:Notification
    public double getOldDoubleValue();                      Implements:Notification
    public float getOldFloatValue();                        Implements:Notification
    public int getOldIntValue();                           Implements:Notification
    public long getOldLongValue();                          Implements:Notification
    public short getOldShortValue();                        Implements:Notification
    public String getOldStringValue();                     Implements:Notification
    public Object getOldValue();                           Implements:Notification
    public int getPosition();                            Implements:Notification
    public boolean isReset();                             Implements:Notification
    public boolean isTouch();                            Implements:Notification
}

```

```
// Methods Implementing Notification
public int getFeatureID(Class expectedClass);
public boolean merge(Notification notification);
// Protected Instance Fields
protected Notification notification;
protected Object notifier;
}
```

Hierarchy: Object → NotificationWrapper (Notification)

Notifier

org.eclipse.emf.common.notify

Notifier is the interface for an object that can send notifications to an adapter. EObject, the base interface for all modeled objects in EMF, extends this interface.

The notifier's list of adapters is available from eAdapters(). When eNotify() is called, each adapter is notified via its notifyChanged() method, provided delivery is enabled. The eDeliver() and eSetDeliver() methods form a pair of accessors to control and test whether this is the case.

```
public interface Notifier {
// Public Instance Methods
public abstract EList eAdapters();
public abstract boolean eDeliver();
public abstract void eNotify(Notification notification);
public abstract void eSetDeliver(boolean deliver);
}
```

NotifyingList

org.eclipse.emf.common.notify collection

The NotifyingList interface is a base for lists that deliver notifications when they are modified. Such a list may have a feature, a feature ID, and a managing notifier, which are returned by the three NotifyingList methods and which the list is expected to include in its notifications. The other details of the notification are to be filled in as is appropriate for the particular modification.

This interface is implemented by EcoreEList, the base class for all the lists that implement multivalued Ecore features.

```
public interface NotifyingList extends EList {
// Property Accessor Methods (by property name)
public abstract Object getFeature();
public abstract int getFeatureID();
public abstract Object getNotifier();
}
```

Hierarchy: (NotifyingList(EList(List(Collection))))

The org.eclipse.emf.common.util Package

The org.eclipse.emf.common.util package contains a number of miscellaneous utility interfaces and classes that are used throughout EMF. Most notably, EMF-specific list and map interfaces are defined, and extensible basic implementations are provided. Also included are an interface and implementation class that form a basis for implementing enumerated types and a class for working with Uniform Resource Identifiers (URIs).

AbstractEnumerator

org.eclipse.emf.common.util

The `AbstractEnumerator` class provides a simple implementation of the `Enumerator` interface and is extended by all generated classes that represent enumerated types.

```
public abstract class AbstractEnumerator implements Enumerator {
    // Protected Constructors
    protected AbstractEnumerator(int value, String name);
    // Property Accessor Methods (by property name)
    public final String getName();
    public final int getValue();
    // Public Methods Overriding Object
    public final String toString();
}
```

Implements:Enumerator
Implements:Enumerator

Hierarchy: Object → AbstractEnumerator (Enumerator)

AbstractTreeIterator

org.eclipse.emf.common.util collection

`AbstractTreeIterator` provides an abstract base for correct implementation of `TreeIterator`. To obtain a fully functional tree iterator, derived classes need only implement `getChildren()` to return an ordinary iterator over the children of a given object.

```
public abstract class AbstractTreeIterator extends BasicEList implements TreeIterator {
    // Public Constructors
    public AbstractTreeIterator(Object object);
    public AbstractTreeIterator(Object object, boolean includeRoot);
    // Methods Implementing TreeIterator
    public void prune();
    // Methods Implementing Iterator
    public boolean hasNext();
    public Object next();
    public void remove();
    // Protected Instance Methods
    protected abstract Iterator getChildren(Object object);
    // Protected Instance Fields
    protected boolean includeRoot;
    protected Iterator nextPruneIterator;
    protected Iterator nextRemoveIterator;
    protected Object object;
}
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → AbstractTreeIterator(TreeIterator(Iterator))

BasicEList

org.eclipse.emf.common.util collection

The `BasicEList` class provides a highly extensible, array-backed implementation of the `EList` interface.

List items are stored in the protected `data` field, an object array that is allocated by `newData()`. This method can be overridden to create typed storage, providing dynamic type checking at negligible cost.

Before returning an object, `get()` calls the `resolve()` method, which simply returns the object itself. However, this gives subclasses that override `resolve()` an opportunity to transform objects as they are fetched.

This class includes simple boolean tests that determine what kind of values can be stored in the list. Null items are permitted if `canContainNull()` returns `true`. Duplicates are forbidden if `isUnique()` returns `true`. Equality is tested using the `equals()` method, rather than the more efficient `==` operator, if `useEquals()` returns `true`. This class' implementations return `true`, `false`, and `true`, respectively, but subclasses should override these to make the list more restrictive when appropriate. More generally, the `validate()` method, which is implemented here to test the `null` constraint based on `canContainNull()`, can be overridden to provide further validation.

After adding, removing, setting, or moving a list item, or clearing the list, `didAdd()`, `didRemove()`, `didSet()`, `didMove()`, or `didClear()` is called. In addition, `didChange()` is called after any of these changes. These callbacks enable a subclass to observe or react to changes to the list. Only `didClear()` has a nonempty default implementation; it just calls `didRemove()` for each object that was in the list.

The iterators returned by `iterator()` and `listIterator()` are instances of the inner classes `EIterator` and `EListIterator`, respectively. The `basicIterator()` and `basicListIterator()` methods return instances of `NonResolvingEIterator` and `NonResolvingEListIterator`, which skip the usual call to `resolve()` before returning a list item. The `basicList()` method returns a nonresolving view of the list's data. This result is an instance of the `UnmodifiableEList` inner class, to prevent accidental concurrent modifications.

```
public class BasicEList extends AbstractList implements EList, Cloneable, Serializable {
    // Public Constructors
    public BasicEList();                                         empty
    public BasicEList(int initialCapacity);
    public BasicEList(Collection collection);
    // Protected Constructors
    protected BasicEList(int size, Object[] data);
    // Public Inner Classes
    public static class UnmodifiableEList;
    // Protected Inner Classes
    protected class EIterator;
    protected class EListIterator;
    protected class NonResolvingEIterator;
    protected class NonResolvingEListIterator;
    // Property Accessor Methods (by property name)
    public boolean isEmpty();                                     Overrides:AbstractCollection
    // Public Instance Methods
    public boolean addAllUnique(Collection collection);
    public boolean addAllUnique(int index, Collection collection);
    public void addUnique(Object object);
    public void addUnique(int index, Object object);
    public Object[] data();
    public void grow(int minimumCapacity);
    public void setData(int size, Object[] data);
    public Object setUnique(int index, Object object);
    public void shrink();
    // Public Methods Overriding AbstractList
    public boolean add(Object object);
    public void add(int index, Object object);
    public boolean addAll(int index, Collection collection);
    public void clear();
    public boolean equals(Object object);
    public Object get(int index);
```

```

public int hashCode();
public int indexOf(Object object);
public Iterator iterator();
public int lastIndexOf(Object object);
public ListIterator listIterator();
public ListIterator listIterator(int index);
public Object remove(int index);
public Object set(int index, Object object);
// Public Methods Overriding AbstractCollection
public boolean addAll(Collection collection);
public boolean contains(Object object);
public boolean containsAll(Collection collection);
public boolean remove(Object object);
public boolean removeAll(Collection collection);
public boolean retainAll(Collection collection);
public int size();
public Object[] toArray();
public Object[] toArray(Object[] array);
public String toString();
// Public Methods Overriding Object
public Object clone();
// Methods Implementing EList
public void move(int index, Object object);
public Object move(int targetIndex, int sourceIndex);
// Protected Instance Methods
protected Object assign(int index, Object object);
protected Iterator basicIterator();
protected List basicList();
protected ListIterator basicListIterator();
protected ListIterator basicListIterator(int index);
protected boolean canContainNull();
protected void didAdd(int index, Object newObject);
protected void didChange();
protected void didClear(int size, Object[] oldObjects);
protected void didMove(int index, Object movedObject, int oldIndex);
empty
protected void didRemove(int index, Object oldObject);                                empty
protected void didSet(int index, Object newObject, Object oldObject);
empty
protected boolean equalObjects(Object firstObject, Object secondObject);
protected Collection getDuplicates(Collection collection);
protected Collection getNonDuplicates(Collection collection);
protected boolean isUnique();                                                        constant
protected Object[] newData(int capacity);
protected Object resolve(int index, Object object);
protected boolean useEquals();                                                       constant
protected Object validate(int index, Object object);
// Protected Instance Fields
protected Object[] data;
protected int size;
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable)

BasicEList.EIterator

org.eclipse.emf.common.util

BasicEList.EIterator is an extensible implementation of java.util.Iterator for a BasicEList.

```

protected class BasicEList.EIterator implements Iterator {
// Public Constructors
public EIterator();
// Methods Implementing Iterator
public boolean hasNext();                                empty

```

```

    public Object next();
    public void remove();
    // Protected Instance Methods
    protected void checkModCount();
    // Protected Instance Fields
    protected int cursor;
    protected int expectedModCount;
    protected int lastCursor;
}

```

Hierarchy: Object → BasicEList.EIterator(Iterator)

BasicEList.EListIterator

org.eclipse.emf.common.util

\BasicEList.EListIterator is an extensible implementation of java.util.ListIterator for a BasicEList.

```

protected class BasicEList.EListIterator extends BasicEList.EIterator implements ListIterator {
    // Public Constructors
    public EListIterator();
    // Methods Implementing ListIterator
    public void add(Object object);
    public boolean hasPrevious();
    public int nextIndex();
    public Object previous();
    public int previousIndex();
    public void set(Object object);
}

```

Hierarchy: Object → BasicEList.EIterator(Iterator) → BasicEList.EListIterator(ListIterator(Iterator))

BasicEList.NonResolvingEIterator

org.eclipse.emf.common.util

BasicEList.NonResolvingEIterator is an extensible implementation of java.util.Iterator for a BasicEList. It accesses the underlying list items directly, rather than calling get(), to skip the call to resolve().

```

protected class BasicEList.NonResolvingEIterator extends BasicEList.EIterator {
    // Public Constructors
    public NonResolvingEIterator();
    // Public Methods Overriding BasicEList.EIterator
    public Object next();
    public void remove();
}

```

Hierarchy: Object → BasicEList.EIterator(Iterator) → BasicEList.NonResolvingEIterator

BasicEList.NonResolvingEListIterator

org.eclipse.emf.common.util

BasicEList.NonResolvingEListIterator is an extensible implementation of java.util.ListIterator for a BasicEList. It accesses the underlying list items directly, rather than calling get(), to skip the call to resolve().

```
protected class BasicEList.NonResolvingEListIterator extends BasicEList.EListIterator {
    // Public Constructors
    public NonResolvingEListIterator();
    public NonResolvingEListIterator(int index);                                empty
    // Public Methods Overriding BasicEList.EListIterator
    public void add(Object object);
    public Object previous();
    public void set(Object object);
    // Public Methods Overriding BasicEList.EIterator
    public Object next();
    public void remove();
}
```

Hierarchy: Object → BasicEList.EIterator(Iterator) → BasicEList.EListIterator(ListIterator(Iterator)) → BasicEList.NonResolvingEListIterator

BasicEList.UnmodifiableEList

org.eclipse.emf.common.util collection

BasicEList.UnmodifiableEList is an unmodifiable version of BasicEList. Each of its set(), add(), addAll(), remove(), removeAll(), retainAll(), clear(), move(), shrink(), and grow() methods throw an UnsupportedOperationException.

```
public static class BasicEList.UnmodifiableEList extends BasicEList {
    // Public Constructors
    public UnmodifiableEList(int size, Object[] data);
    // Public Methods Overriding BasicElist
    public boolean add(Object object);
    public void add(int index, Object object);
    public boolean addAll(Collection collection);
    public boolean addAll(int index, Collection collection);
    public void clear();
    public void grow(int minimumCapacity);
    public Iterator iterator();
    public ListIterator listIterator();
    public ListIterator listIterator(int index);
    public void move(int index, Object object);
    public Object move(int targetIndex, int sourceIndex);
    public boolean remove(Object object);
    public Object remove(int index);
    public boolean removeAll(Collection collection);
    public boolean retainAll(Collection collection);
    public Object set(int index, Object object);
    public void shrink();
}
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → BasicEList.UnmodifiableEList

BasicEMap

`org.eclipse.emf.common.util
collection`

The `BasicEMap` class provides an extensible implementation of the `EMap` interface. This implementation primarily maintains a bucket hash table, `entryData`, and a list, `delegateEList`. Both are populated with the same objects, which implement `Map.Entry`. Not surprisingly, the `EList` methods delegate to `delegateEList`, and the `Map` methods are implemented using `entryData`. The `initializeDelegateEList()` method creates the list, a `BasicEList` that overrides the callbacks to provide synchronization with the hash table. `BasicEMap` also includes its own callbacks, `didAdd()`, `didRemove()`, `didModify()`, and `didClear()`. Only `didClear()` has a nonempty default implementation; it just calls `didRemove()` for each entry that was in the map.

The bucket hash table is implemented as an array of `BasicEList` instances. The array is allocated by `newEntryData()` and, more important, the lists themselves are created by `newList()`. This method returns a new `BasicEList` with `newData()` overridden to ensure that the type of the underlying array is `EntryImpl`, a simple `Entry` implementation defined as an inner class. It is `newList()` that should be overridden if a subclass is to further restrict the type of the map entries. In that case, `newEntry()` should also be overridden to ensure that entries created by `put()` and `clone()` are of the correct type.

`BasicEMap` provides validation methods, `validateKey()` and `validateValue()`, that are called by `newEntry()` and, by default, do nothing. Also, `useEqualsForKey()` and `useEqualsForValue()` determine whether to test for equality of keys and values, respectively, using the `equals()` method, rather than the `==` operator. This class' implementations return `true` for both, but can be overridden to realize an efficiency gain, where appropriate.

Like `BasicEList`, `BasicEMap` also includes a `resolve()` method, which is called before the map version of `get()` returns an object. Its implementation simply returns the object itself, but subclasses can override it to transform objects as they are fetched.

```
public class BasicEMap implements EMap, Cloneable, Serializable {
    // Public Constructors
    public BasicEMap();
    public BasicEMap(int initialCapacity);
    public BasicEMap(Map map);
    // Public Inner Classes
    public static interface Entry;
    // Protected Inner Classes
    protected class BasicEMapIterator;
    protected class BasicEMapKeyIterator;
    protected class BasicEMapValueIterator;
    protected class DelegatingMap;
    protected class EntryImpl;
    protected static class View;
    // Property Accessor Methods (by property name)
    public boolean isEmpty();                                            Implements:List
    // Public Methods Overriding Object
    public Object clone();
    public boolean equals(Object object);
    public int hashCode();
    public String toString();
    // Methods Implementing EMap
    public boolean containsKey(Object key);
    public boolean containsValue(Object value);
    public Set entrySet();
    public Object get(Object key);
    public int indexOfKey(Object key);
    public Set keySet();
}
```

```

public Map map();
public Object put(Object key, Object value);
public void putAll(Map map);
public Object removeKey(Object key);
public Collection values();
// Methods Implementing EList
public void move(int index, Object object);
public Object move(int targetIndex, int sourceIndex);
// Methods Implementing List
public boolean add(Object object);
public void add(int index, Object object);
public boolean addAll(Collection collection);
public boolean addAll(int index, Collection collection);
public void clear();
public boolean contains(Object object);
public boolean containsAll(Collection collection);
public Object get(int index);
public int indexOf(Object object);
public Iterator iterator();
public int lastIndexOf(Object object);
public ListIterator listIterator();
public ListIterator listIterator(int index);
public boolean remove(Object object);
public Object remove(int index);
public boolean removeAll(Collection collection);
public boolean retainAll(Collection collection);
public Object set(int index, Object object);
public int size();
public List subList(int start, int end);
public Object[] toArray();
public Object[] toArray(Object[] array);
// Protected Instance Methods
protected void didAdd(BasicEMap.Entry entry);                                empty
protected void didClear(BasicEList[] oldEntryData);
protected void didModify(BasicEMap.Entry entry, Object oldValue);
empty
protected void didRemove(BasicEMap.Entry entry);                                empty
protected void doClear();                                                     empty
protected void doMove(BasicEMap.Entry entry);
protected void doPut(BasicEMap.Entry entry);
protected void doRemove(BasicEMap.Entry entry);
protected void ensureEntryDataExists();
protected BasicEMap.Entry entryForKey(int index, int hash, Object key);
protected int entryIndexForKey(int index, int hash, Object key);
protected boolean grow(int minimumCapacity);
protected int hashOf(Object key);
protected int indexOf(int hash);
protected void initializeDelegateEList();
protected BasicEMap.Entry newEntry(int hash, Object key, Object value);
protected BasicEList[] newEntryData(int capacity);
protected BasicEList newList();
protected Object putEntry(BasicEMap.Entry entry, Object value);
protected Object removeEntry(int index, int entryIndex);
protected Object resolve(Object key, Object value);
protected boolean useEqualsForKey();
protected boolean useEqualsForValue();                                         constant
protected void validateKey(Object key);                                         constant
protected void validateValue(Object value);                                     empty
// Protected Instance Fields
protected EList delegateEList;
protected BasicEList[] entryData;
protected int modCount;
protected int size;
protected BasicEMap.View view;
}

```

Hierarchy: Object → BasicEMap (EMap (EList (List (Collection)))), Cloneable, Serializable)

BasicEMap.BasicEMapIterator

org.eclipse.emf.common.util

BasicEMap.BasicEMapIterator is an implementation of java.util.Iterator that yields the entries in a BasicEMap.

```
protected class BasicEMap.BasicEMapIterator implements Iterator {
// No Constructor
// Methods Implementing Iterator
    public boolean hasNext();
    public Object next();
    public void remove();
// Protected Instance Methods
    protected void scan();
    protected Object yield(BasicEMap.Entry entry);
// Protected Instance Fields
    protected int cursor;
    protected int entryCursor;
    protected int expectedModCount;
    protected int lastCursor;
    protected int lastEntryCursor;
}
```

Hierarchy: Object → BasicEMap.BasicEMapIterator(Iterator)

BasicEMap.BasicEMapKeyIterator

org.eclipse.emf.common.util

BasicEMap.BasicEMapKeyIterator is an implementation of java.util.Iterator that yields the keys of a BasicEMap.

```
protected class BasicEMap.BasicEMapKeyIterator extends BasicEMap.BasicEMapIterator {
// No Constructor
// Protected Instance Methods
    protected Object yield(BasicEMap.Entry entry);
    Overrides:BasicEMap.BasicEMapIterator
}
```

Hierarchy: Object → BasicEMap.BasicEMapIterator(Iterator) → BasicEMap.BasicEMapKeyIterator

BasicEMap.BasicEMapValueIterator

org.eclipse.emf.common.util

BasicEMap.BasicEMapValueIterator is an implementation of java.util.Iterator that yields the values of a BasicEMap.

```
protected class BasicEMap.BasicEMapValueIterator extends BasicEMap.BasicEMapIterator {
// No Constructor
// Protected Instance Methods
    protected Object yield(BasicEMap.Entry entry);
    Overrides:BasicEMap.BasicEMapIterator
}
```

Hierarchy: Object → BasicEMap.BasicEMapIterator(Iterator) → BasicEMap.BasicEMapValueIterator

BasicEMap.DelegatingMap

org.eclipse.emf.common.util

BasicEMap.DelegatingMap is an implementation of the `java.util.Map` interface that simply delegates to its associated BasicEMap.

```
protected class BasicEMap.DelegatingMap implements Map {
    // Public Constructors
    public DelegatingMap();                                empty
    // Property Accessor Methods (by property name)
    public boolean isEmpty();                             Implements:Map
    // Public Methods Overriding Object
    public boolean equals(Object object);
    public int hashCode();
    // Methods Implementing Map
    public void clear();
    public boolean containsKey(Object key);
    public boolean containsValue(Object value);
    public Set entrySet();
    public Object get(Object key);
    public Set keySet();
    public Object put(Object key, Object value);
    public void putAll(Map map);
    public Object remove(Object key);
    public int size();
    public Collection values();
}
```

Hierarchy: Object → BasicEMap.DelegatingMap (Map)

BasicEMap.Entry

org.eclipse.emf.common.util

BasicEMap.Entry is a simple extension of the `java.util.Map.Entry` interface that allows the entry itself to provide the value by which it is hashed, and allows the map to set this value and the key value. Although BasicEMap only ever needs to set these values via a constructor, subclasses might require the accessors if they create their entry instances using a factory.

```
public static interface BasicEMap.Entry extends Map.Entry {
    // Property Accessor Methods (by property name)
    public abstract int getHash();
    public abstract void setHash(int hash);
    public abstract void setKey(Object key);
}
```

Hierarchy: (BasicEMap.Entry (Map.Entry))

BasicEMap.View

org.eclipse.emf.common.util

The BasicEMap.View class is simply a container for the entry, key, value, and map views offered by BasicEMap.

```
protected static class BasicEMap.View {
    // Public Constructors
    public View();                                         empty
    // Public Instance Fields
    public Set entrySet;
```

```

    public Set keySet;
    public Map map;
    public Collection values;
}

```

DelegatingEList

```

org.eclipse.emf.common.util
collection

```

The `DelegatingEList` class provides a highly extensible, abstract implementation of the `EList` interface, backed by the delegate list that a subclass provides.

The subclass should implement `delegateList()` to return the list to which the basic list operations are delegated. Alternately, to use a backing that does not implement the `List` interface, it can implement `delegateList()` to just return `null`, and then override the implementations of the other methods whose names begin with “delegate,” which by default all perform corresponding list operations. These methods isolate access to the delegate list.

Before returning an object, `get()` calls the `resolve()` method, which simply returns the object itself. However, this gives subclasses that override `resolve()` an opportunity to transform objects as they are fetched.

This class includes simple boolean tests that determine what kinds of values can be stored in the list. Null items are permitted if `canContainNull()` returns `true`. Duplicates are forbidden if `isUnique()` returns `true`. Equality is tested using the `equals()` method, rather than the more efficient `==` operator, if `useEquals()` returns `true`. This class’ implementations return `true`, `false`, and `true`, respectively, but subclasses should override these to make the list more restrictive when appropriate. More generally, the `validate()` method, which is implemented here to test the `null` constraint based on `canContainNull()`, can be overridden to provide further validation.

After adding, removing, setting, or moving a list item, or clearing the list, `didAdd()`, `didRemove()`, `didSet()`, `didMove()`, or `didClear()` is called. In addition, `didChange()` is called after any of these changes. These callbacks enable a subclass to observe or react to changes to the list. Only `didClear()` has a nonempty default implementation; it just calls `didRemove()` for each object that was in the list.

The iterators returned by `iterator()` and `listIterator()` are instances of the inner classes `EIterator` and `EListIterator`, respectively. The `basicIterator()` and `basicListIterator()` methods return instances of `NonResolvingEIterator` and `NonResolvingEListIterator`, which skip the usual call to `resolve()` before returning a list item. The `basicList()` method returns a nonresolving view of the list’s data: the backing list itself, obtained via `delegeateBasicList()`, which by default just returns the list supplied by `delegateList()`.

```

public abstract class DelegatingEList extends AbstractList implements EList, Cloneable, Serializable {
    // Public Constructors
    public DelegatingEList();
    public DelegatingEList(Collection collection);                                empty
    // Public Inner Classes
    public static class UnmodifiableEList;
    // Protected Inner Classes
    protected class EIterator;
    protected class EListIterator;
    protected class NonResolvingEIterator;
    protected class NonResolvingEListIterator;
    // Property Accessor Methods (by property name)
    public boolean isEmpty();
    Overrides:AbstractCollection

```

```

// Public Instance Methods
public boolean addAllUnique(Collection collection);
public boolean addAllUnique(int index, Collection collection);
public void addUnique(Object object);
public void addUnique(int index, Object object);
public Object setUnique(int index, Object object);
// Public Methods Overriding AbstractList
public boolean add(Object object);
public void add(int index, Object object);
public boolean addAll(int index, Collection collection);
public void clear();
public boolean equals(Object object);
public Object get(int index);
public int hashCode();
public int indexOf(Object object);
public Iterator iterator();
public int lastIndexOf(Object object);
public ListIterator listIterator();
public ListIterator listIterator(int index);
public Object remove(int index);
public Object set(int index, Object object);
// Public Methods Overriding AbstractCollection
public boolean addAll(Collection collection);
public boolean contains(Object object);
public boolean containsAll(Collection collection);
public boolean remove(Object object);
public boolean removeAll(Collection collection);
public boolean retainAll(Collection collection);
public int size();
public Object[] toArray();
public Object[] toArray(Object[] array);
public String toString();
// Methods Implementing EList
public void move(int index, Object object);
public Object move(int targetIndex, int sourceIndex);
// Protected Instance Methods
protected Iterator basicIterator();
protected List basicList();
protected ListIterator basicListIterator();
protected ListIterator basicListIterator(int index);
protected boolean canContainNull();
constant
protected void delegateAdd(Object object);
protected void delegateAdd(int index, Object object);
protected List delegateBasicList();
protected void delegateClear();
protected boolean delegateContains(Object object);
protected boolean delegateContainsAll(Collection collection);
protected boolean delegateEquals(Object object);
protected Object delegateGet(int index);
protected int delegateHashCode();
protected int delegateIndexOf(Object object);
protected boolean delegateIsEmpty();
protected Iterator delegateIterator();
protected int delegateLastIndexOf(Object object);
protected abstract List delegateList();
protected ListIterator delegateListIterator();
protected Object delegateRemove(int index);
protected Object delegateSet(int index, Object object);
protected int delegateSize();
protected Object[] delegateToArray();
protected Object[] delegateToArray(Object[] array);
protected String delegateToString();
protected void didAdd(int index, Object newObject);
empty
protected void didChange();
empty
protected void didClear(int size, Object[] oldObjects);
protected void didMove(int index, Object movedObject, int ←
oldIndex);                                empty
protected void didRemove(int index, Object oldObject);

```

```

empty
protected void didSet(int index, Object newObject, Object ←
oldObject);
                           empty
protected void doClear(int oldSize, Object[] oldData);
protected boolean equalObjects(Object firstObject, Object secondObject);
protected Collection getDuplicates(Collection collection);
protected Collection getNonDuplicates(Collection collection);
protected boolean isUnique();
constant
protected Object resolve(int index, Object object);
protected boolean useEquals();
constant
protected Object validate(int index, Object object);
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → DelegatingEList(EList(List)), Cloneable, Serializable

DelegatingEList.EIterator

org.eclipse.emf.common.util

DelegatingEList.EIterator is an extensible implementation of java.util.Iterator for a DelegatingEList.

```

protected class DelegatingEList.EIterator implements Iterator {
// Public Constructors
public EIterator();                                                 empty
// Methods Implementing Iterator
public boolean hasNext();
public Object next();
public void remove();
// Protected Instance Methods
protected void checkModCount();
// Protected Instance Fields
protected int cursor;
protected int expectedModCount;
protected int lastCursor;
}

```

Hierarchy: Object → DelegatingEList.EIterator(Iterator)

DelegatingEList.EListIterator

org.eclipse.emf.common.util

DelegatingEList.EListIterator is an extensible implementation of java.util.ListIterator for a DelegatingEList.

```

protected class DelegatingEList.EListIterator extends DelegatingEList.EIterator implements ↪
ListIterator {
// Public Constructors
public EListIterator();
public EListIterator(int index);                                         empty
//Methods Implementing ListIterator
public void add(Object object);
public boolean hasPrevious();
public int nextIndex();
public Object previous();
public int previousIndex();
public void set(Object object);
}

```

Hierarchy: Object → DelegatingEList.EIterator(Iterator) → DelegatingEList.EListIterator(ListIterator(Iterator))

DelegatingEList.NonResolvingEIterator

org.eclipse.emf.common.util

DelegatingEList.NonResolvingEIterator is an extensible implementation of java.util.Iterator for a DelegatingEList. It accesses the underlying list items via delegateGet(), rather than get(), to skip the call to resolve().

```
protected class DelegatingEList.NonResolvingEIterator extends DelegatingEList.EIterator {
    // Public Constructors
    public NonResolvingEIterator();
    // Public Methods Overriding DelegatingEList.EIterator
    public Object next();
    public void remove();
}
```

Hierarchy: Object → DelegatingEList.EIterator(Iterator) → DelegatingEList.NonResolvingEIterator

DelegatingEList.NonResolvingEListIterator

org.eclipse.emf.common.util

DelegatingEList.NonResolvingEListIterator is an extensible implementation of java.util.ListIterator for a DelegatingEList. It accesses the underlying list items via delegateGet(), rather than get(), to skip the call to resolve().

```
protected class DelegatingEList.NonResolvingEListIterator extends ↵
    DelegatingEList.EListIterator {
    // Public Constructors
    public NonResolvingEListIterator();
    public NonResolvingEListIterator(int index);
    // Public Methods Overriding DelegatingEList.EListIterator
    public void add(Object object);
    public Object previous();
    public void set(Object object);
    // Public Methods Overriding DelegatingEList.EIterator
    public Object next();
    public void remove();
}
```

Hierarchy: Object → DelegatingEList.EIterator(Iterator) → DelegatingEList.EListIterator(ListIterator(Iterator)) → DelegatingEList.NonResolvingEListIterator

DelegatingEList.UnmodifiableEList

org.eclipse.emf.common.util collection

DelegatingEList.UnmodifiableEList is an unmodifiable version of DelegatingEList. Each of its set(), add(), addAll(), remove(), removeAll(), retainAll(), clear(), and move() methods throw an UnsupportedOperationException.

```

public static class DelegatingEList.UnmodifiableEList extends DelegatingEList {
    // Public Constructors
    public UnmodifiableEList(List underlyingList);
    // Public Methods Overriding DelegatingEList
    public boolean add(Object object);
    public void add(int index, Object object);
    public boolean addAll(Collection collection);
    public boolean addAll(int index, Collection collection);
    public void clear();
    public Iterator iterator();
    public ListIterator listIterator();
    public ListIterator listIterator(int index);
    public void move(int index, Object object);
    public Object move(int targetIndex, int sourceIndex);
    public boolean remove(Object object);
    public Object remove(int index);
    public boolean removeAll(Collection collection);
    public boolean retainAll(Collection collection);
    public Object set(int index, Object object);
    // Protected Instance Methods
    protected List delegateList();
    Overrides:DelegatingEList
    // Protected Instance Fields
    protected List underlyingList;
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → DelegatingEList(EList(List), Cloneable, Serializable) → DelegatingEList.UnmodifiableEList

ECollections

org.eclipse.emf.common.util

Like java.util.Collections, ECollections only contains static utility members and cannot be instantiated. The unmodifiableEList() method returns an unmodifiable view of the specified EList. The EMPTY_ELIST field holds an efficiently implemented instance of an unmodifiable, empty EList.

```

public class ECollections {
    // No Constructor
    // Public Constants
    public static final EList EMPTY_ELIST;
    // Public Class Methods
    public static EList unmodifiableEList(EList list);
}

```

EList

org.eclipse.emf.common.util collection

EList is the base interface for lists used throughout EMF. It is a slight extension of the standard Java list interface, java.util.List, adding two move() methods. These methods move a given object or an object at a given position to a new position.

```

public interface EList extends List {
    // Public Instance Methods
    public abstract void move(int newPosition, Object object);
    public abstract Object move(int newPosition, int oldPosition);
}

```

Hierarchy: (EList(List(Collection)))

EMap

org.eclipse.emf.common.util
collection

EMap defines the interface for key-value mappings used in EMF. Notice that EMap is derived from EList, not from Java's java.util.Map. Thus, EMF considers a map to be a list of key-value pairs, each implementing Map.Entry, that also maintains an index for fast retrieval by key.

Ideally, EMap would also extend Java's Map interface, but that is not possible because of the conflicting return values of List.remove() and Map.remove(): boolean and Object, respectively. So, EMap instead includes removeKey(), along with the rest of the methods defined by Map. It also provides a map() method, which returns a modifiable Map view. Finally, it adds an indexOfKey() into the mix, which simply returns the index of the Map.Entry with the given key.

```
public interface EMap extends EList {
    // Public Instance Methods
    public abstract boolean containsKey(Object key);
    public abstract boolean containsValue(Object value);
    public abstract Set entrySet();
    public abstract Object get(Object key);
    public abstract int indexOfKey(Object key);
    public abstract Set keySet();
    public abstract Map map();
    public abstract Object put(Object key, Object value);
    public abstract void putAll(Map map);
    public abstract Object removeKey(Object key);
    public abstract Collection values();
}
```

Hierarchy: (EMap(EList(List(Collection))))

Enumerator

org.eclipse.emf.common.util

The Enumerator interface is implemented by classes that represent enumerated types. As described in Section 9.2.3, EMF implements enumerated types using the Type-safe Enum pattern, where a class with no public constructor provides constant instance fields representing the type's legal values. It is these classes that implement Enumerator, with each instance corresponding to one enumerated type literal. Thus, this interface simply declares accessors for the name and value of the literal.

```
public interface Enumerator {
    // Property Accessor Methods (by property name)
    public abstract String getName();
    public abstract int getValue();
}
```

Logger

org.eclipse.emf.common.util

`Logger` defines a simple interface for handling log entries. In EMF, it is generally implemented by a plug-in class derived from `EMFPlugin`, which uses the Eclipse platform logging facility when running within Eclipse. Alternate logging mechanisms can be used instead, if available and more appropriate.

The interface includes a single `log()` method, which has an argument that is an arbitrary object. Depending on the logging mechanism, that object can be a `Throwable` or something else entirely.

```
public interface Logger {
    // Public Instance Methods
    public abstract void log(Object logEntry);
}
```

ResourceLocator

org.eclipse.emf.common.util

`ResourceLocator` defines a simple interface for retrieving text and image resources. In EMF, it is generally implemented by a plug-in class derived from `EMFPlugin`.

The `getBaseURL()` method returns the URL on which location of resources is based. It can be used by the other methods, `getImage()` and `getString()`, which look up and return an image and a string, respectively, according to a specified key. The second form of `getString()` substitutes the values in the `substitutions` parameter into the retrieved string.

```
public interface ResourceLocator {
    // Property Accessor Methods (by property name)
    public abstract URL getBaseURL();
    // Public Instance Methods
    public abstract Object getImage(String key);
    public abstract String getString(String key);
    public abstract String getString(String key, Object[] substitutions);
}
```

TreeIterator

org.eclipse.emf.common.util

`TreeIterator` provides an interface for iterating over all the nodes of a tree. Each call to `next()` returns the next item in a preorder traversal. However, if `prune()` is called, any children of the current object are skipped by the following call to `next()`.

```
public interface TreeIterator extends Iterator {
    // Public Instance Methods
    public abstract void prune();
}
```

Hierarchy. (`TreeIterator(Iterator)`)

UniqueEList

org.eclipse.emf.common.util collection

The UniqueEList class provides an extensible implementation of EList that does not permit duplicate items. It is a simple extension of BasicEList that overrides isUnique() to return true.

```
public class UniqueEList extends BasicEList {
    // Public Constructors
    public UniqueEList();
    public UniqueEList(int initialCapacity);
    public UniqueEList(Collection collection);
    // Protected Instance Methods
    protected boolean isUnique();                                Overrides:BasicEList ←
    constant
}
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → UniqueEList

URI

org.eclipse.emf.common.util

URI is a utility class for working with Uniform Resource Identifiers. Instances of it are used throughout EMF to identify resources, or specific portions of resources. They are generally created by calling one of the following static factory methods to parse a string into the URI it represents: createURI(), createFileURI() or createPlatformResourceURI().

A URI can be absolute or relative, depending on whether or not it includes a scheme. It might also be hierarchical or not. If it is nonhierarchical, it must include an opaque part. If not, it can include an authority, a device, a path made up of segments, and a query. Any URI can also include a fragment. Each of these portions of a URI can be accessed via the method that carries the same name.

Like java.lang.String, URI is an immutable class; a URI instance offers several methods that return a new URI object based on its current state. Most useful, a relative URI can be resolved against a base absolute URI, which typically identifies the document in which it appears, by calling resolve(). The inverse to this is dereolve(), which answers the question, "What relative URI will resolve, against the given base, to this absolute URI?"

```
public final class URI {
    // No Constructor
    // Public Class Methods
    public static URI createFileURI(String pathName);
    public static URI createGenericURI(String scheme, String opaquePart, String fragment);
    public static URI createHierarchicalURI(String scheme, String authority, String device, ←
    String query, String fragment);
    public static URI createHierarchicalURI(String scheme, String authority, String device, ←
    String[] segments, String query, String fragment);
    public static URI createHierarchicalURI(String[] segments, String query, String fragment);
    public static URI createPlatformResourceURI(String pathName);
    public static URI createURI(String uri);
    public static boolean validAuthority(String value);
    public static boolean validDevice(String value);
    public static boolean validFragment(String value);
    constant
        public static boolean validOpaquePart(String value);
        public static boolean validQuery(String value);
        public static boolean validScheme(String value);
        public static boolean validSegment(String value);
        public static boolean validSegments(String[] value);
```

```

// Property Accessor Methods (by property name)
public boolean isCurrentDocumentReference();
public boolean isEmpty();
public boolean isFile();
public boolean isHierarchical();
public boolean isPrefix();
public boolean isRelative();
// Public Instance Methods
public URI appendFileExtension(String fileExtension);
public URI appendFragment(String fragment);
public URI appendQuery(String query);
public URI appendSegment(String segment);
public URI appendSegments(String[] segments);
public String authority();
public URI dereResolve(URI base);
public URI dereResolve(URI base, boolean preserveRootParents, boolean anyRelPath, boolean ←
shorterRelPath);
public String device();
public String devicePath();
public String fileExtension();
public String fragment();
public boolean hasAbsolutePath();
public boolean hasAuthority();
public boolean hasDevice();
public boolean hasEmptyPath();
public boolean hasFragment();
public boolean hasOpaquePart();
public boolean hasPath();
public boolean hasQuery();
public boolean hasRelativePath();
public boolean hasTrailingPathSeparator();
public String host();
public String lastSegment();
public String opaquePart();
public String path();
public String port();
public String query();
public URI replacePrefix(URI oldPrefix, URI newPrefix);
public URI resolve(URI base);
public URI resolve(URI base, boolean preserveRootParents);
public String scheme();
public String segment(int i);
public int segmentCount();
public String[] segments();
public List segmentsList();
public String toFileString();
public URI trimFileExtension();
public URI trimFragment();
public URI trimQuery();
public URI trimSegments(int i);
public String userInfo();
// Public Methods Overriding Object
public boolean equals(Object obj);
public int hashCode();
public String toString();
}

```

WrappedException

org.eclipse.emf.common.util
unchecked

The `WrappedException` class is used to wrap exceptions that are not subclasses of `RuntimeException`, allowing them to be rethrown by methods that do not declare them in an explicit `throws` clause. The original, wrapped exception is available via the `exception()` method.

```
public class WrappedException extends RuntimeException {  
    // Public Constructors  
    public WrappedException(Exception exception);  
    public WrappedException(String message, Exception exception);  
    // Public Instance Methods  
    public Exception exception();  
    // Public Methods Overriding Throwable  
    public void printStackTrace();  
    public void printStackTrace(PrintStream printStream);  
    public void printStackTrace(PrintWriter printWriter);  
    // Protected Instance Fields  
    protected Exception wrappedException;  
}
```

Hierarchy: Object → Throwable(Serializable) → Exception → RuntimeException → WrappedException

Chapter 16. The org.eclipse.emf.common.ui Plug-In

The `org.eclipse.emf.common.ui` plug-in provides extensions to and utilities for several SWT and JFace classes that are used in other EMF UI plug-ins. Like `org.eclipse.emf.common`, this plug-in is simply a prerequisite for other parts of EMF. Thus, it can also be reused easily in non-EMF applications.

Requires: `org.eclipse.ui` → `org.eclipse.core.resources` →,
`org.eclipse.emf.common` (`org.eclipse.core.runtime` (`org.apache.xerces`))

The `org.eclipse.emf.common.ui` Package

The `org.eclipse.emf.common.ui` package includes the plug-in class for the `org.eclipse.emf.common.ui` plug-in, as well as one utility class for changing the appearance of viewers.

`CommonUIPlugin`

`org.eclipse.emf.common.ui`

`CommonUIPlugin` is the plug-in class, the central provider of resources and logging capabilities, for the `org.eclipse.emf.common.ui` plug-in. Like all other EMF plug-in classes, it extends from `EMFPlugin` and uses the Singleton design pattern, with its single instance available as the constant `INSTANCE` field.

When running within Eclipse, the Eclipse plug-in class, of type `CommonUIPlugin.Implementation`, is available from the static `getPlugin()` method. The same object is returned by `getPluginResourceLocator()` and is used as a delegate by the `EMFPlugin` implementations of `getBaseURL()`, `getImage()`, `getString()`, and `log()` to make use of the platform's plug-in support facilities. When running stand-alone, `getPluginResourceLocator()` returns null, and the base class uses its own implementations.

```
public final class CommonUIPlugin extends EMFPlugin {
    // No Constructor
    // Public Constants
    public static final CommonUIPlugin INSTANCE;
    // Public Inner Classes
    public static class Implementation;
    // Public Class Methods
    public static CommonUIPlugin.Implementation getPlugin();
    // Property Accessor Methods (by property name)
    public ResourceLocator getPluginResourceLocator();                                Overrides:EMFPlugin
}
```

Hierarchy: Object → `EMFPlugin` (`ResourceLocator`, `Logger`) → `CommonUIPlugin`

CommonUIPlugin.Implementation

org.eclipse.emf.common.ui

`CommonUIPlugin.Implementation` is derived from `org.eclipse.core.runtime.Plugin`, the base class that represents a plug-in within Eclipse. It is instantiated by the platform, and then used to provide platform-based implementations for most of the methods of the `CommonUIPlugin` plug-in class. It is also through this class that other Eclipse facilities, including the preference store and plug-in descriptor, can be accessed.

```
public static class CommonUIPlugin.Implementation extends EMFPlugin.EclipsePlugin {
    // Public Constructors
    public Implementation(IPluginDescriptor descriptor);
}
```

Hierarchy: Object → Plugin → EMFPlugin.EclipsePlugin(ResourceLocator, Logger) → CommonUIPlugin.Implementation

ViewerPane

org.eclipse.emf.common.ui

The `ViewerPane` class adds a title bar, with an optional toolbar and menu, to a `Viewer`. It must be subclassed to provide an implementation for the abstract `createViewer()` method that creates the desired type of viewer.

After the `ViewerPane` has been created, the container to which it should be added is passed to `createControl()`, which creates the widget itself, calling `createViewer()` to supply the viewer. That viewer is then available via `getViewer()`.

The text and image that appears on the title bar can be set by passing both to `setTitle()`. An alternate form of this method accepts a single object; if the viewer has a label provider, it is used to obtain the text and image for that object.

The `getToolBarManager()` and `getMenuManager()` methods return empty managers for the toolbar and menu, which display in the title bar only if items are added to them.

```
public abstract class ViewerPane implements IPropertyListener, Listener {
    // Public Constructors
    public ViewerPane(IWorkbenchPage page, IWorkbenchPart part);
    // Property Accessor Methods (by property name)
    public Collection getBuddies();
    public Control getControl();
    public Font getFont();
    public MenuManager getMenuManager();
    public void setPinVisible(boolean visible);
    public void setTitle(Object object);
    public ToolBarManager getToolBarManager();
    public Viewer getViewer();
    // Public Instance Methods
    public void createControl(Composite parent);
    public abstract Viewer createViewer(Composite parent);
    public void dispose();
    public void hookFocus(Control ctrl);
    public void setFocus();
    public void setTitle(String title, Image image);
    public void showFocus(boolean inFocus);
    public void updateActionBars();
    public void updateTitles();
    // Public Methods Overriding Object
    public String toString();
```

```

// Methods Implementing IPropertyListener
public void propertyChanged(Object source, int propId);
// Methods Implementing Listener
public void handleEvent(Event event);
// Protected Instance Methods
protected void createTitleBar();
protected void doMaximize();
protected ViewForm getViewForm();
protected void requestActivation();
// Protected Instance Fields
protected ToolBar actionBar;
protected Collection buddies;
protected Composite container;
protected ViewForm control;
protected MenuManager menuManager;
protected MouseListener mouseListener;
protected IWorkbenchPage page;
protected IWorkbenchPart part;
protected IPartListener partListener;
protected ToolItem pinButton;
protected Image pinButtonImage;
protected boolean pinVisible;
protected Image pullDownImage;
protected ToolBar systemBar;
protected CLabel titleLabel;
protected ToolBarManager toolBarManager;
protected Viewer viewer;
}

```

Hierarchy: Object → ViewerPane (IPropertyListener, Listener)

The org.eclipse.emf.common.ui.celleditor Package

The org.eclipse.emf.common.ui.celleditor package provides extensions to JFace CellEditor classes and to SWT ControlEditor classes.

ExtendedComboBoxCellEditor

org.eclipse.emf.common.ui.celleditor

The ExtendedComboBoxCellEditor class extends JFace's ComboBoxCellEditor to make it object-based, rather than index-based. ComboBoxCellEditor is a cell editor that presents a list of items in a combo box. The items are specified as an array of strings, and the selected item is set and retrieved by doSetValue() and doGetValue() as the index of the string in the array. The items array can be specified via either the constructor or the setItems() method.

Instead of an array of strings, ExtendedComboBoxCellEditor's constructors take a list of objects and a label provider. It uses the label provider to obtain strings for the object, which it then supplies as the string array. It also overrides doSetValue() and doGetValue() to take and return the objects themselves, rather than their indices.

Note that the optional sorted constructor parameter is currently ignored.

```

public class ExtendedComboBoxCellEditor extends ComboBoxCellEditor {
// Public Constructors
    public ExtendedComboBoxCellEditor(Composite composite, List list, ILabelProvider ←
labelProvider);
    public ExtendedComboBoxCellEditor(Composite composite, List list, ILabelProvider ←
labelProvider, boolean sorted);
    public ExtendedComboBoxCellEditor(Composite composite, List list, ILabelProvider ←
labelProvider, int style);
    public ExtendedComboBoxCellEditor(Composite composite, List list, ILabelProvider ←
labelProvider, boolean sorted, int style);

```

```

// Public Class Methods
public static String[] createItems(List list, ILabelProvider labelProvider, boolean sorted);
// Public Methods Overriding ComboBoxCellEditor
public Object doGetValue();
public void doSetValue(Object value);
// Protected Instance Fields
protected List list;
}

```

Hierarchy: ObjectCellEditorComboBoxCellEditor → ExtendedComboBoxCellEditor

ExtendedDialogCellEditor

org.eclipse.emf.common.ui.celleditor

The `ExtendedDialogCellEditor` class extends JFace's `DialogCellEditor` to use a label provider to obtain the label text for a given value. `DialogCellEditor` is an abstract cell editor that includes a label and a button. The label shows the current value, and pressing the button opens a dialog box that allows the user to change that value. The label text is obtained by simply calling `toString()` on the value object.

Instead of calling `toString()`, `ExtendedDialogCellEditor` uses the label provider passed to its constructor to obtain a label for the object.

```

public abstract class ExtendedDialogCellEditor extends DialogCellEditor {
// Public Constructors
    public ExtendedDialogCellEditor(Composite composite, ILabelProvider labelProvider);
// Protected Instance Methods
    protected void updateContents(Object object);
Overrides:DialogCellEditor
// Protected Instance Fields
    protected ILabelProvider labelProvider;
}

```

Hierarchy: Object → CellEditor → DialogCellEditor → ExtendedDialogCellEditor

ExtendedTableEditor

org.eclipse.emf.common.ui.celleditor

The `ExtendedTableEditor` class extends SWT's `TableEditor` to handle its own activation. `TableEditor` is a manager for a control that appears over a cell in a `Table`, allowing in-place editing of table values. To use it, a listener must be attached to the `Table` to activate the editor at the appropriate time, hooking up the control with the correct column of the selected row.

`ExtendedTableEditor` does its own listening for keyboard, mouse, and selection events from the table, and calls its abstract `editItem()` method when the editor potentially needs to be activated. This method is passed the appropriate `TableItem` and column index.

```

public abstract class ExtendedTableEditor extends TableEditor implements KeyListener, ←
MouseListener, SelectionListener {
// Public Constructors
    public ExtendedTableEditor(Table table);
// Public Instance Methods
    public void dismiss();
// Public Methods Overriding TableEditor
    public void setEditor(Control canvas, TableItem tableItem, int column);
// Methods Implementing KeyListener
    public void keyPressed(KeyEvent event);                                empty
    public void keyReleased(KeyEvent event);
// Methods Implementing MouseListener
}

```

```

public void mouseDoubleClick(MouseEvent event);
public void mouseDown(MouseEvent event);
public void mouseUp(MouseEvent event);
// Methods Implementing SelectionListener
public void widgetDefaultSelected(SelectionEvent event);
public void widgetSelected(SelectionEvent event);
// Protected Instance Methods
protected abstract void editItem(TableItem tableItem, int column);
// Protected Instance Fields
protected TableItem editTableItem;
protected int editTableItemColumn;
protected TableItem selectedTableItem;
protected Table table;
}

```

Hierarchy. Object → ControlEditor → TableEditor → ExtendedTableEditor (KeyListener(SWTEventListener(EventListener)), MouseListener(SWTEventListener), SelectionListener(SWTEventListener))

ExtendedTableTreeEditor

org.eclipse.emf.common.ui.celleditor

The ExtendedTableTreeEditor class extends SWT's TableTreeEditor to handle its own activation. TableTreeEditor is a manager for a control that appears over a cell in a TableTree, allowing in-place editing of values. To use it, a listener must be attached to the TableTree to activate the editor at the appropriate time, hooking up the control with the correct column of the selected row.

ExtendedTableTreeEditor does its own listening for keyboard, mouse, and selection events from the TableTree, and calls its abstract editItem() method when the editor potentially needs to be activated. This method is passed the appropriate TableTreeItem and column index.

```

public abstract class ExtendedTableTreeEditor extends TableTreeEditor implements KeyListener, ←
MouseListener, SelectionListener {
// Public Constructors
    public ExtendedTableTreeEditor(TableTree tableTree);
// Public Instance Methods
    public void dismiss();
// Public Methods Overriding TableTreeEditor
    public void setEditor(Control canvas, TableTreeItem tableTreeItem, int column);
// Methods Implementing KeyListener
    public void keyPressed(KeyEvent event);
    public void keyReleased(KeyEvent event);                                empty
// Methods Implementing MouseListener
    public void mouseDoubleClick(MouseEvent event);
    public void mouseDown(MouseEvent event);
    public void mouseUp(MouseEvent event);
// Methods Implementing SelectionListener
    public void widgetDefaultSelected(SelectionEvent event);
    public void widgetSelected(SelectionEvent event);
// Protected Instance Methods
    protected abstract void editItem(TableItem tableItem, TableTreeItem tableTreeItem, int ←
column);
// Protected Instance Fields
    protected TableTreeItem editTableTreeItem;
    protected int editTableTreeItemColumn;
    protected TableItem selectedTableItem;
    protected Table table;
    protected TableTree tableTree;
}

```

Hierarchy: Object → ControlEditor → TableTreeEditor → ExtendedTableTreeEditor (KeyListener(SWTEventListener(EventListener)), MouseListener(SWTEventListener), SelectionListener(SWTEventListener))

ExtendedTreeEditor

org.eclipse.emf.common.ui.celleditor

The ExtendedTreeEditor class extends SWT's TreeEditor to handle its own activation. TreeEditor is a manager for a control that appears over a cell in a Tree, allowing in-place editing of tree values. To use it, a listener must be attached to the Tree to activate the editor at the appropriate time, hooking up the control with the selected item.

ExtendedTreeEditor does its own listening for keyboard, mouse, and selection events from the tree, and calls its abstract editItem() method when the editor potentially needs to be activated. This method is passed the appropriate TreeItem.

```
public abstract class ExtendedTreeEditor extends TreeEditor implements SelectionListener, ←
MouseListener, KeyListener {
// Public Constructors
    public ExtendedTreeEditor(Tree tree);
// Methods Implementing SelectionListener
    public void widgetDefaultSelected(SelectionEvent event);
    public void widgetSelected(SelectionEvent event);
// Methods Implementing MouseListener
    public void mouseDoubleClick(MouseEvent event);                                empty
    public void mouseDown(MouseEvent event);
    public void mouseUp(MouseEvent event);
// Methods Implementing KeyListener
    public void keyPressed(KeyEvent event);
    public void keyReleased(KeyEvent event);                                         empty
// Protected Instance Methods
    protected abstract void editItem(TreeItem treeItem);
// Protected Instance Fields
    protected TreeItem editTreeItem;
    protected TreeItem selectedTreeItem;
    protected Tree tree;
}
```

Hierarchy: Object → ControlEditor → TreeEditor → ExtendedTreeEditor (SelectionListener(SWTEventListener(EventListener)), MouseListener(SWTEventListener), KeyListener(SWTEventListener))

The org.eclipse.emf.common.ui.viewer Package

The org.eclipse.emf.common.ui.viewer package provides an extension to one of the JFace Viewer classes.

ExtendedTableTreeViewer

org.eclipse.emf.common.ui.viewer

The ExtendedTableTreeViewer class extends JFace's TableTreeViewer to draw tree lines and allow images in the first column. TableTreeViewer displays a structured model in a TableTree widget. Unfortunately, TableTree does not allow an image to be set in the first column, reserving that space for the "+" and "-" icons used to expand and collapse the tree.

`ExtendedTableTreeViewer` remedies this behavior, allowing an additional image to be inserted immediately before the text of the first column. It also draws tree lines, which are omitted by `TableTreeViewer`. The `newItem()` method is overridden to return an instance of the `ExtendedTableTreeItem` inner class, which, unlike the ordinary `TableTreeItem` class, supports a user-specified image in the first column.

```
public class ExtendedTableTreeViewer extends TableTreeViewer {
    // Public Constructors
    public ExtendedTableTreeViewer(TableTree tableTree);
    public ExtendedTableTreeViewer(Composite parent);
    public ExtendedTableTreeViewer(Composite parent, int style);
    // Public Constants
    public static final String ITEM_ID;                                ="TableTreeItemID"
    // Public Inner Classes
    public class ExtendedTableTreeItem;
    // Protected Instance Methods
    protected void createImagePadding(int width);
    protected void hookControl(Control control);
    Overrides:TableTreeViewer
    protected Item newItem(Widget parent, int flags, int index);
    Overrides:TableTreeViewer
    // Protected Instance Fields
    protected String imagePadding;
    protected int imagePaddingWidth;
    protected int indent;
    protected int offset;
}
```

Hierarchy: Object → Viewer(IInputSelectionProvider(IInputProvider, ISelectionProvider)) → ContentViewer → StructuredViewer → AbstractTreeViewer → TableTreeViewer → ExtendedTableTreeViewer

ExtendedTableTreeViewer.ExtendedTableTreeItem

org.eclipse.emf.common.ui.viewer

`ExtendedTableTreeViewer.ExtendedTableTreeItem` extends SWT's `TableTreeItem`, adding support for a user-specified image in the first column. Unlike its base class, `ExtendedTableTreeItem` defines `setImage()` such that it does have an effect when its first parameter is 0: It sets an additional image that is inserted between the ordinary expand/collapse tree icon and the text. The `getImage()` method, however, is unchanged: When passed 0, it still returns the expand/collapse tree icon. The user-set icon is available via `getFirstImage()`.

```
public class ExtendedTableTreeViewer.ExtendedTableTreeItem extends TableTreeItem {
    // Public Constructors
    public ExtendedTableTreeItem(TableTree parent, int style);
    public ExtendedTableTreeItem(TableTree parent, int style, int index);
    public ExtendedTableTreeItem(TableTreeItem parent, int style);
    public ExtendedTableTreeItem(TableTreeItem parent, int style, int index);
    // Property Accessor Methods (by property name)
    public Image getFirstImage();
    public int getImagePaddingWidth();
    // Public Methods Overriding TableTreeItem
    public String getText(int index);
    public void setImage(int index, Image image);
    public void setText(int index, String text);
    // Protected Instance Fields
    protected Image firstImage;
}
```

Hierarchy: Object → Widget → Item → `TableTreeItem` → `ExtendedTableTreeViewer.ExtendedTableTreeItem`

Chapter 17. The org.eclipse.emf.ecore Plug-In

The `org.eclipse.emf.ecore` plug-in provides an API for Ecore, the metamodel for EMF, that is largely generated from the model itself. In addition, it defines an interface for modeling abstract persistent resources, for which concrete implementations can be created to manage specific persistent forms. The default implementation, provided by the `org.eclipse.emf.ecore.xmi` plug-in, is the subject of Chapter 18.

Although this plug-in declares dependencies on `org.eclipse.core.resources` and `org.eclipse.core.runtime`, these are, in fact, soft dependencies. Their use is isolated, with equivalent, non-Eclipse mechanisms being used when they are not available. The dependencies must be declared, however, as they are required to build this plug-in.

The plug-in offers four extension points, which can be extended by plug-ins that wish to register a package, resource factory, or URI mapping:

- Extending `generated_package` registers a package against a namespace URI in `EPackage.Registry.INSTANCE`.
- Extending `extension_parser` registers a resource factory against a URI extension in `Resource.Factory.Registry.INSTANCE`.
- Extending `protocol_parser` registers a resource factory against a URI protocol, or scheme, in `ResourceFactory.Registry.INSTANCE`.
- Extending `uri_mapping` registers a source to target URI prefix mapping in `URIConverter.URI_MAP`.

It also contributes to its own `generated_package` extension point, registering Ecore to be represented by the `org.eclipse.emf.ecore.EcorePackage` interface.

Requires: `org.apache.xerces`, `org.eclipse.core.resources`,
`org.eclipse.emf.common` (`org.eclipse.core.runtime`) →

Extension Points: `generated_package`, `extension_parser`, `protocol_parser`,
`uri_mapping`

Extensions: `org.eclipse.emf.ecore.generated_package`

The `org.eclipse.emf.ecore` Package

The `org.eclipse.emf.ecore` package provides an API for Ecore, which is used to create and manipulate core models to define structures for instance models in EMF. As discussed in Chapter 5, this API is based on, and primarily generated from, the Ecore model itself. As such, it includes one Java interface for each modeled class and for the Ecore package and factory.

The Ecore model is thoroughly discussed in Chapter 5; the roles of particular attributes and references are not repeated here. Thus, the focus is on methods other than the simple accessors.

EAnnotation

```
org.eclipse.emf.ecore
eobject
```

The **EAnnotation** interface provides a mechanism by which additional information can be attached to any other core model object. It includes only ordinary accessor methods generated for the **source** attribute and for the **contents**, **references**, **eModelElements**, and **details** references.

```
public interface EAnnotation extends EModelElement {
    // Property Accessor Methods (by property name)
    public abstract EList getContents();
    public abstract EMap getDetails();
    public abstract EModelElement getEModelElement();
    public abstract void setEModelElement(EModelElement value);
    public abstract EList getReferences();
    public abstract String getSource();
    public abstract void setSource(String value);
}
```

Hierarchy: (EAnnotation (EModelElement (EObject (Notifier))))

EAttribute

```
org.eclipse.emf.ecore
eobject
```

The **EAttribute** interface represents an attribute, a simple component of object data. To the methods defined by **EStructuralFeature**, it adds accessors for the **iD** attribute and the **eAttributeType** reference.

```
public interface EAttribute extends EStructuralFeature {
    // Property Accessor Methods (by property name)
    public abstract EDataType getEAttributeType();
    public abstract boolean isID();
    public abstract void setID(boolean value);
}
```

Hierarchy: (EAttribute (EStructuralFeature (ETypedElement (ENamedElement (EModelElement (EObject (Notifier)))))))

EClass

```
org.eclipse.emf.ecore
eobject
```

The **EClass** interface represents a class or interface in a core model. To the methods defined by **EClassifier**, it adds accessors for the **abstract** and **interface** attributes, and for the **eAttributes**, **eReferences**, **eOperations**, **eSuperTypes**, **eAllSuperTypes**, **eAllAttributes**, **eIDAttribute**, **eAllReferences**, **eAllContainments**, **eAllStructuralFeatures**, and **eAllOperations** references. In addition, two **getEStructuralFeature()** methods allow the retrieval of a single structural feature by ID or by name, and **isSuperTypeOf()** tests whether another class is a derived type.

```
public interface EClass extends EClassifier {
    // Property Accessor Methods (by property name)
    public abstract boolean isAbstract();
    public abstract void setAbstract(boolean value);
    public abstract EList getAllAttributes();
    public abstract EList getAllContainments();
```

```

public abstract EList getEAllOperations();
public abstract EList getEAllReferences();
public abstract EList getEAllStructuralFeatures();
public abstract EList getEAllSuperTypes();
public abstract EList getEAttributes();
public abstract EAttribute getEIDAttribute();
public abstract EList getEOperations();
public abstract EList getEReferences();
public abstract EList getESuperTypes();
public abstract boolean isInterface();
public abstract void setInterface(boolean value);
// Public Instance Methods
public abstract EStructuralFeature getEStructuralFeature(int featureID);
public abstract EStructuralFeature getEStructuralFeature(String featureName);
public abstract boolean isSuperTypeOf(EClass someClass);
}

```

Hierarchy: (EClass (EClassifier (ENamedElement (EModelElement (EObject (Notifier)))))))

EClassifier

```

org.eclipse.emf.ecore
EObject

```

The EClassifier interface represents a type in a core model. As such, it is the base class for EClass and EDataType. It includes accessors for the **instanceClassName**, **instanceClass**, and **defaultValue** attributes, and for the **ePackage** reference. It also defines **isInstance()**, which tests whether an object is of the type that the EClassifier represents, and **getClassifierID()**, which returns its unique numeric identifier.

```

public interface EClassifier extends ENamedElement {
// Property Accessor Methods (by property name)
    public abstract int getClassifierID();
    public abstract Object getDefaultValue();
    public abstract EPackage getEPackage();
    public abstract Class getInstanceClass();
    public abstract void setInstanceClass(Class value);
    public abstract String getInstanceClassName();
    public abstract void setInstanceClassName(String value);
// Public Instance Methods
    public abstract boolean isInstance(Object object);
}

```

Hierarchy: (EClassifier (ENamedElement (EModelElement (EObject (Notifier))))))

EcoreFactory

```

org.eclipse.emf.ecore
EObject

```

The EcoreFactory interface is implemented by the factory responsible for creating instances of Ecore model objects, which implement the other interfaces in this package. It includes an accessor for its associated **EcorePackage** object and one method to create an instance of each type of object—that is, each model class—defined by Ecore.

There is a default implementation of this factory available as the constant **eINSTANCE** field, which should normally be used for creating Ecore model objects. Note that this refers to the same factory that could be obtained via the package registry as follows: **EPackage.Registry.INSTANCE.getEPackage(EcorePackage.eNS_URI).getEFactoryInstance()**.

```

public interface EcoreFactory extends EFactory {
    // Public Constants
    public static final EcoreFactory eINSTANCE;
    // Property Accessor Methods (by property name)
    public abstract EcorePackage getEcorePackage();
    // Public Instance Methods
    public abstract EAnnotation createEAnnotation();
    public abstract EAttribute createEAttribute();
    public abstract EClass createEClass();
    public abstract EDataType createEDataType();
    public abstract EEnum createEEnum();
    public abstract EEnumLiteral createEEnumLiteral();
    public abstract EFactory createEFactory();
    public abstractEObject createEObject();
    public abstract EOperation createEOperation();
    public abstract EPackage createEPackage();
    public abstract EParameter createEParameter();
    public abstract EReference createEReference();
}

```

Hierarchy: (EcoreFactory(EFactory(EModelElement(EObject(Notifier))))))

EcorePackage

```

org.eclipse.emf.ecore
eobject

```

EcorePackage defines an interface for accessing Ecore's metadata. For each classifier and structural feature in the model, it includes a classifier or feature ID and an accessor for the object itself. These objects and integers are used to identify model elements in reflective API operations and notification handling.

The last accessor, `getEcoreFactory()`, returns the factory associated with the Ecore package.

The default implementation of the Ecore package is available as the constant `eINSTANCE` field. This is the package object that should be used universally for accessing metadata for the Ecore package itself. It is the same one that would be obtained via the package registry as follows: `EPackage.Registry.INSTANCE.getEPackage(EcorePackage.ens_URI)`, where `ens_URI` is the constant string that uniquely identifies the package.

```

public interface EcorePackage extends EPackage {
    // Public Constants
    public static final int EANNOTATION; =1
    public static final int EANNOTATION_FEATURE_COUNT; =6
    public static final int EANNOTATION__CONTENTS; =4
    public static final int EANNOTATION__DETAILS; =2
    public static final int EANNOTATION__EANNOTATIONS; =0
    public static final int EANNOTATION__EMODEL_ELEMENT; =3
    public static final int EANNOTATION__REFERENCES; =5
    public static final int EANNOTATION__SOURCE; =0
    public static final int EATTRIBUTE; =1
    public static final int EATTRIBUTE_FEATURE_COUNT; =17
    public static final int EATTRIBUTE__CHANGEABLE; =3
    public static final int EATTRIBUTE__DEFAULT_VALUE; =8
    public static final int EATTRIBUTE__DEFAULT_VALUE_LITERAL; =7
    public static final int EATTRIBUTE__EANNOTATIONS; =0
    public static final int EATTRIBUTE__EATTRIBUTE_TYPE; =16
    public static final int EATTRIBUTE__ECONTAINING_CLASS; =14
    public static final int EATTRIBUTE__ETYPE; =2
    public static final int EATTRIBUTE__ID; =15
    public static final int EATTRIBUTE__LOWER_BOUND; =9
    public static final int EATTRIBUTE__MANY; =11
    public static final int EATTRIBUTE__NAME; =1
    public static final int EATTRIBUTE__REQUIRED; =12
}

```

```

public static final int EATTRIBUTE_TRANSIENT; =5
public static final int EATTRIBUTE_UNIQUE; =6
public static final int EATTRIBUTE_UNSETTABLE; =13
public static final int EATTRIBUTE_UPPER_BOUND; =10
public static final int EATTRIBUTE_VOLATILE; =4
public static final int EBOOLEAN; =18
public static final int EBOOLEAN_OBJECT; =19
public static final int EBYTE; =22
public static final int EBYTE_OBJECT; =23
public static final int ECHAR; =20
public static final int ECHARACTER_OBJECT; =21
public static final int ECLASS; =2
public static final int ECLASSIFIER; =3
public static final int ECLASSIFIER_FEATURE_COUNT; =6
public static final int ECLASSIFIER_DEFAULT_VALUE; =4
public static final int ECLASSIFIER_EANNOTATIONS; =0
public static final int ECLASSIFIER_EPACKAGE; =5
public static final int ECLASSIFIER_INSTANCE_CLASS; =3
public static final int ECLASSIFIER_INSTANCE_CLASS_NAME; =2

=2
public static final int ECLASSIFIER_NAME; =1
public static final int ECLASS_FEATURE_COUNT; =19
public static final int ECLASS_ABSTRACT; =6
public static final int ECLASS_DEFAULT_VALUE; =4
public static final int ECLASS_EALL_ATTRIBUTES; =10
public static final int ECLASS_EALL_CONTAINMENTS; =14
public static final int ECLASS_EALL_OPERATIONS; =15
public static final int ECLASS_EALL_REFERENCES; =11
public static final int ECLASS_EALL_STRUCTURAL_FEATURES; =16

=16
public static final int ECLASS_EALL_SUPER_TYPES; =17
public static final int ECLASS_EANNOTATIONS; =0
public static final int ECLASS_EATTRIBUTES; =13
public static final int ECLASS_EID_ATTRIBUTE; =18
public static final int ECLASS_EOPERATIONS; =9
public static final int ECLASS_EPACKAGE; =5
public static final int ECLASS_EREFERENCES; =12
public static final int ECLASS_ESUPER_TYPES; =8
public static final int ECLASS_INSTANCE_CLASS; =3
public static final int ECLASS_INSTANCE_CLASS_NAME; =2
public static final int ECLASS_INTERFACE; =7
public static final int ECLASS_NAME; =1
public static final int EDATA_TYPE; =4
public static final int EDATA_TYPE_FEATURE_COUNT; =7
public static final int EDATA_TYPE_DEFAULT_VALUE; =4
public static final int EDATA_TYPE_EANNOTATIONS; =0
public static final int EDATA_TYPE_EPACKAGE; =5
public static final int EDATA_TYPE_INSTANCE_CLASS; =3
public static final int EDATA_TYPE_INSTANCE_CLASS_NAME; =2
public static final int EDATA_TYPE_NAME; =1
public static final int EDATA_TYPE_SERIALIZABLE; =6
public static final int EDOUBLE; =24
public static final int EDOUBLE_OBJECT; =25
public static final int EENUM; =5
public static final int ENUMERATOR; =27
public static final int EENUM_FEATURE_COUNT; =8
public static final int EENUM_LITERAL; =6
public static final int EENUM_LITERAL_FEATURE_COUNT; =5
public static final int EENUM_LITERAL_EANNOTATIONS; =0
public static final int EENUM_LITERAL_EENUM; =4
public static final int EENUM_LITERAL_INSTANCE; =3
public static final int EENUM_LITERAL_NAME; =1
public static final int EENUM_LITERAL_VALUE; =2
public static final int EENUM_DEFAULT_VALUE; =4
public static final int EENUM_EANNOTATIONS; =0
public static final int EENUM_ELITERALS; =7
public static final int EENUM_EPACKAGE; =5
public static final int EENUM_INSTANCE_CLASS; =3
public static final int EENUM_INSTANCE_CLASS_NAME; =2
public static final int EENUM_NAME; =1
public static final int EENUM_SERIALIZABLE; =6

```

```

public static final int EE_LIST; =26
public static final int EFACTORY; =7
public static final int EFACTORY_FEATURE_COUNT; =2
public static final int EFACTORY_EANNOTATIONS; =0
public static final int EFACTORY_EPACKAGE; =1
public static final int EFLOAT; =28
public static final int EFLOAT_OBJECT; =29
public static final int EINT; =30
public static final int EINTEGER_OBJECT; =31
public static final int EJAVA_CLASS; =32
public static final int EJAVA_OBJECT; =33
public static final int ELONG; =34
public static final int ELONG_OBJECT; =35
public static final int EMODEL_ELEMENT; =8
public static final int EMODEL_ELEMENT_FEATURE_COUNT; =1
public static final int EMODEL_ELEMENT_EANNOTATIONS; =0
public static final int ENAMED_ELEMENT; =9
public static final int ENAMED_ELEMENT_FEATURE_COUNT; =2
public static final int ENAMED_ELEMENT_EANNOTATIONS; =0
public static final int ENAMED_ELEMENT_NAME; =1
public static final int EOBJECT; =10
public static final int EOBJECT_FEATURE_COUNT; =0
public static final int EOPERATION; =11
public static final int EOPERATION_FEATURE_COUNT; =6
public static final int EOPERATION_EANNOTATIONS; =0
public static final int EOPERATION_ECONTAINING_CLASS; =3
public static final int EOPERATION_EXCEPTIONS; =5
public static final int EOPERATION_EPARAMETERS; =4
public static final int EOPERATIONETYPE; =2
public static final int EOPERATION_NAME; =1
public static final int EPACKAGE; =12
public static final int EPACKAGE_FEATURE_COUNT; =8
public static final int EPACKAGE_EANNOTATIONS; =0
public static final int EPACKAGE_ECLASSIFIERS; =5
public static final int EPACKAGE_EFACTORY_INSTANCE; =4
public static final int EPACKAGE_ESUBPACKAGES; =6
public static final int EPACKAGE_ESUPER_PACKAGE; =7
public static final int EPACKAGE_NAME; =1
public static final int EPACKAGE_NS_PREFIX; =3
public static final int EPACKAGE_NS_URI; =2
public static final int EPARAMETER; =13
public static final int EPARAMETER_FEATURE_COUNT; =4
public static final int EPARAMETER_EANNOTATIONS; =0
public static final int EPARAMETER_EOPERATION; =3
public static final int EPARAMETERETYPE; =2
public static final int EPARAMETER_NAME; =1
public static final int EREFERENCE; =14
public static final int EREFERENCE_FEATURE_COUNT; =20
public static final int EREFERENCE_CHANGEABLE; =3
public static final int EREFERENCE_CONTAINER; =16
public static final int EREFERENCE_CONTAINMENT; =15
public static final int EREFERENCE_DEFAULT_VALUE; =8
public static final int EREFERENCE_DEFAULT_VALUE_LITERAL; =7

public static final int EREFERENCE_EANNOTATIONS; =0
public static final int EREFERENCE_ECONTAINING_CLASS; =14
public static final int EREFERENCE_EOPPOSITE; =18
public static final int EREFERENCE_EREFERENCE_TYPE; =19
public static final int EREFERENCEETYPE; =2
public static final int EREFERENCE_LOWER_BOUND; =9
public static final int EREFERENCE_MANY; =11
public static final int EREFERENCE_NAME; =1
public static final int EREFERENCE_REQUIRED; =12
public static final int EREFERENCE_RESOLVE_PROXY; =17
public static final int EREFERENCE_TRANSIENT; =5
public static final int EREFERENCE_UNIQUE; =6
public static final int EREFERENCE_UNSETTABLE; =13
public static final int EREFERENCE_UPPER_BOUND; =10
public static final int EREFERENCE_VOLATILE; =4
public static final int ERESOURCE; =36
public static final int ESHORT; =37

```

```

public static final int ESHORT_OBJECT; =38
public static final int ESTRING; =39
public static final int ESTRING_TO_STRING_MAP_ENTRY; =17
public static final int ESTRING_TO_STRING_MAP_ENTRY_FEATURE_COUNT;
=2
public static final int ESTRING_TO_STRING_MAP_ENTRY_KEY;
=0
public static final int ESTRING_TO_STRING_MAP_ENTRY_VALUE;
=1
public static final int ESTRUCTURAL_FEATURE; =15
public static final int ESTRUCTURAL_FEATURE_FEATURE_COUNT;
=15
public static final int ESTRUCTURAL_FEATURE_CHANGEABLE; =3
public static final int ESTRUCTURAL_FEATURE_DEFAULT_VALUE;
=8
public static final int ESTRUCTURAL_FEATURE_DEFAULT_VALUE_LITERAL;
=7
public static final int ESTRUCTURAL_FEATURE_EANNOTATIONS;
=0
public static final int ESTRUCTURAL_FEATURE_ECONTAINING_CLASS;
=14
public static final int ESTRUCTURAL_FEATUREETYPE; =2
public static final int ESTRUCTURAL_FEATURE_LOWER_BOUND;
=9
public static final int ESTRUCTURAL_FEATURE_MANY; =11
public static final int ESTRUCTURAL_FEATURE_NAME; =1
public static final int ESTRUCTURAL_FEATURE_REQUIRED; =12
public static final int ESTRUCTURAL_FEATURE_TRANSIENT; =5
public static final int ESTRUCTURAL_FEATURE_UNIQUE; =6
public static final int ESTRUCTURAL_FEATURE_UNSETTABLE;
=13
public static final int ESTRUCTURAL_FEATURE_UPPER_BOUND;
=10
public static final int ESTRUCTURAL_FEATURE_VOLATILE; =4
public static final int ETREE_ITERATOR; =40
public static final int ETYPED_ELEMENT; =16
public static final int ETYPED_ELEMENT_FEATURE_COUNT; =3
public static final int ETYPED_ELEMENT_EANNOTATIONS; =0
public static final int ETYPED_ELEMENTETYPE; =2
public static final int ETYPED_ELEMENT_NAME; =1
public static final EcorePackage eINSTANCE; = "ecore"
public static final String eNAME; = "ecore"
public static final String eNS_PREFIX; = "ecore"
public static final String eNS_URI; = "http://www.eclipse.org/emf/2002/
Ecore"
// Property Accessor Methods (by property name)
public abstract EClass getEAnnotation();
public abstract EReference getEAnnotationContents();
public abstract EReference getEAnnotationDetails();
public abstract EReference getEAnnotationEModelElement();
public abstract EReference getEAnnotationReferences();
public abstract EAttribute getEAnnotationSource();
public abstract EClass getEAttribute();
public abstract EReference getEAttributeEAttributeType();
public abstract EAttribute getEAttributeID();
public abstract EDataType getEBoolean();
public abstract EDataType getEBooleanObject();
public abstract EDataType getEByte();
public abstract EDataType getEByteObject();
public abstract EDataType getEChar();
public abstract EDataType getECharacterObject();
public abstract EClass getEClass();
public abstract EAttribute getEClassAbstract();
public abstract EReference getEClassEAllAttributes();
public abstract EReference getEClassEAllContainments();
public abstract EReference getEClassEAllOperations();
public abstract EReference getEClassEAllReferences();
public abstract EReference getEClassEAllStructuralFeatures();
public abstract EReference getEClassEAllSuperTypes();
public abstract EReference getEClassEAttributes();
public abstract EReference getEClassEIDAttribute();

```

```

public abstract EReference getEClass_EOperations();
public abstract EReference getEClass_EReferences();
public abstract EReference getEClass_ESuperTypes();
public abstract EAttribute getEClass_Interface();
public abstract EClass getEClassifier();
public abstract EAttribute getEClassifier_DefaultValue();
public abstract EReference getEClassifier_EPackage();
public abstract EAttribute getEClassifier_InstanceClass();
public abstract EAttribute getEClassifier_InstanceClassName();
public abstract EClass getEDataType();
public abstract EAttribute getEDataType_Serializable();
public abstract EDataType getEDouble();
public abstract EDataType getEDoubleObject();
public abstract EDataType getEEList();
public abstract EClass getEEnum();
public abstract EClass getEEnumLiteral();
public abstract EReference getEEnumLiteral_EEnum();
public abstract EAttribute getEEnumLiteral_Instance();
public abstract EAttribute getEEnumLiteral_Value();
public abstract EReference getEEnum_ELiterals();
public abstract EDataType getEEnumurator();
public abstract EClass getEFactory();
public abstract EReference getEFactory_EPackage();
public abstract EDataType getEFloat();
public abstract EDataType getEFloatObject();
public abstract EDataType getInt();
public abstract EDataType getEIntegerObject();
public abstract EDataType getEJavaClass();
public abstract EDataType getEJavaObject();
public abstract EDataType getELong();
public abstract EDataType getELongObject();
public abstract EClass getEModelElement();
public abstract EReference getEModelElement_EAnnotations();
public abstract EClass getENamedElement();
public abstract EAttribute getENamedElement_Name();
public abstract EClass getEObject();
public abstract EClass getEOperation();
public abstract EReference getEOperation_EContainingClass();
public abstract EReference getEOperation_EExceptions();
public abstract EReference getEOperation_EParameters();
public abstract EClass getEPackage();
public abstract EReference getEPackage_EClassifiers();
public abstract EReference getEPackage_EFactoryInstance();
public abstract EReference getEPackage_ESubpackages();
public abstract EReference getEPackage_ESuperPackage();
public abstract EAttribute getEPackage_NsPrefix();
public abstract EAttribute getEPackage_NsURI();
public abstract EClass getParameter();
public abstract EReference getEParameter_EOperation();
public abstract EClass getEReference();
public abstract EAttribute getEReference_Container();
public abstract EAttribute getEReference_Containment();
public abstract EReference getEReference_EOpposite();
public abstract EReference getEReference_EReferenceType();
public abstract EAttribute getEReference_ResolveProxies();
public abstract EDataType getResource();
public abstract EDataType getEShort();
public abstract EDataType getEShortObject();
public abstract EDataType getString();
public abstract EClass getEStringToStringMapEntry();
public abstract EAttribute getEStringToStringMapEntry_Key();
public abstract EAttribute getEStringToStringMapEntry_Value();
public abstract EClass getEStructuralFeature();
public abstract EAttribute getEStructuralFeature_Changeable();
public abstract EAttribute getEStructuralFeature_DefaultValue();
public abstract EAttribute getEStructuralFeature_DefaultValueLiteral();
public abstract EReference getEStructuralFeature_EContainingClass();
public abstract EAttribute getEStructuralFeature_LowerBound();
public abstract EAttribute getEStructuralFeature_Many();
public abstract EAttribute getEStructuralFeature_Required();
public abstract EAttribute getEStructuralFeature_Transient();

```

```

public abstract EAttribute getEStructuralFeature_Unique();
public abstract EAttribute getEStructuralFeature_Unsettable();
public abstract EAttribute getEStructuralFeature_UpperBound();
public abstract EAttribute getEStructuralFeature_Volatile();
public abstract EDataType getETreeIterator();
public abstract EClass getETypedElement();
public abstract EReference getETypedElement_EType();
public abstract EcoreFactory getEcoreFactory();
}

```

Hierarchy: (EcorePackage (EPackage (ENamedElement (EModelElement (EObject (Notifier)))))))

EDataType

org.eclipse.emf.ecore
eobject

The **EDataType** interface represents a simple data type—a Java primitive or object type that is not modeled as a class in EMF. To the methods defined by **EClassifier**, it adds accessors for the **serializable** attribute.

```

public interface EDataType extends EClassifier {
    // Property Accessor Methods (by property name)
    public abstract boolean isSerializable();
    public abstract void setSerializable(boolean value);
}

```

Hierarchy: (EDataType (EClassifier (ENamedElement (EModelElement (EObject (Notifier)))))))

EEnum

org.eclipse.emf.ecore
eobject

The **EEnum** interface represents an enumerated type, a simple data type defined by an explicit list of values that it may possibly take, called its literals. It defines an accessor for the **eLiterals** reference, as well as two **getEEnumLiteral()** convenience methods, which retrieve a literal by its name or value.

```

public interface EEnum extends EDataType {
    // Property Accessor Methods (by property name)
    public abstract EList getELiterals();
    // Public Instance Methods
    public abstract EEnumLiteral getEEnumLiteral(String name);
    public abstract EEnumLiteral getEEnumLiteral(int value);
}

```

Hierarchy: (EEnum (EDataType (EClassifier (ENamedElement (EModelElement (EObject (Notifier)))))))

EEnumLiteral

org.eclipse.emf.ecore
eobject

The **EEnumLiteral** interface represents a value that can be taken by an enumerated type. It defines accessors for the **value** and **instance** attributes, and for the **eEnum** reference.

```

public interface EEnumLiteral extends ENamedElement, Enumerator {
    // Property Accessor Methods (by property name)
    public abstract EEnum getEEnum();
    public abstract Enumerator getInstance();
    public abstract void setInstance(Enumerator value);
    public abstract int getValue();
    public abstract void setValue(int value);                                Overrides: Enumerator
}

```

Hierarchy: (EEnumLiteral(ENamedElement(EModelElement(EObject(Notifier)))) , Enumerator)

EFactory

org.eclipse.emf.ecore
eobject

The **EFactory** interface is implemented by an object responsible for creating instances of model objects. For example, **EcoreFactory** extends this interface to further define the behavior of the factory for Ecore, itself.

This interface includes accessors for the **ePackage** reference to the factory's corresponding package. It also defines three reflective methods for object creation and data type value conversion. The **create()** method returns a new instance of the model class that is specified as an argument. All objects created by this method must implement **EObject**, the base interface for modeled objects in EMF. The **createFromString()** and **convertToString()** methods convert between a data type value and a string representation of that value, where the type of the value is determined from the specified data type.

```

public interface EFactory extends EModelElement {
    // Property Accessor Methods (by property name)
    public abstract EPackage getEPackage();
    public abstract void setEPackage(EPackage value);
    // Public Instance Methods
    public abstract String convertToString(EDataType eDataType, Object instanceValue);
    public abstract EObject create(EClass eClass);
    public abstract Object createFromString(EDataType eDataType, String literalValue);
}

```

Hierarchy: (EFactory(EModelElement(EObject(Notifier))))

EModelElement

org.eclipse.emf.ecore
eobject

The **EModelElement** interface is the base for all other interfaces representing model elements in Ecore. It provides an accessor for the **eAnnotations** reference, allowing additional information to be attached to any core model object. The **getEAnnotation()** convenience method returns the first annotation from that reference whose **source** attribute matches the specified value.

```

public interface EModelElement extends EObject {
    // Property Accessor Methods (by property name)
    public abstract EList getEAnnotations();
    // Public Instance Methods
    public abstract EAnnotation getEAnnotation(String source);
}

```

Hierarchy: (EModelElement(EObject(Notifier)))

ENamedElement

```
org.eclipse.emf.ecore
eobject
```

The `ENamedElement` interface is extended by most other Ecore interfaces to provide accessors for a `name` attribute.

```
public interface ENamedElement extends EModelElement {
    // Property Accessor Methods (by property name)
    public abstract String getName();
    public abstract void setName(String value);
}
```

Hierarchy: (ENamedElement (EModelElement (EObject (Notifier))))

EObject

```
org.eclipse.emf.ecore
notifier
```

`EObject` is the base interface that must be implemented by all modeled objects in EMF; its method names start with “e” to distinguish them from methods introduced in derived interfaces for specific model elements. `EObject` provides support for the content, reflection, and serialization behaviors and features common to all modeled objects, and it includes the means for participation in the EMF common notification framework by way of its inheritance from `org.eclipse.emf.common.notify.Notifier`.

Any class implementing `EObject` is assumed to also implement `InternalEObject`, which provides lower level access that is not necessarily suitable for general consumption but is required for maintaining the EMF support mechanisms.

If an `EObject` is contained by another, its container is accessed via the `eContainer()` method; `eContainmentFeature()` returns the particular reference through which the object can be reached from its container. An object's contents are available as an unmodifiable list from `eContents()`, and `eAllContents()` returns a `TreeIterator` that iterates over all its direct and indirect contents. The targets of an object's non-containment references are also available as an unmodifiable list, from `eCrossReferences()`. The `Resource` that has contents that include the `EObject`, or one of its containers, is accessed via `eResource()`. An object must belong to a `Resource` to be serialized.

An `EObject`'s metaobject, which defines the structural features available for reflective access, is obtained from `eClass()`. An `EStructuralFeature` can then be used as a parameter to `eGet()`, `eSet()`, `eUnset()`, and `eIsSet()` to get, set and unset its value, and test if its value is not unset. For a feature that is not unsettable, the latter two operations reset its value to its default and test whether its value is not equal to its default, respectively. The second parameter in the two-parameter form of `eGet()` specifies whether the value, if a proxy, should be automatically resolved before it is returned. The single-parameter form always performs proxy resolution.

A proxy is an object defined in a `Resource` that has not been loaded; `eIsProxy()` indicates whether an object is a proxy. An object can be a proxy either because it was accessed with proxy resolution disabled or because proxy resolution failed.

```
public interface EObject extends Notifier {
    // Public Instance Methods
    public abstract TreeIterator eAllContents();
    public abstract EClass eClass();
```

```

public abstract EObject eContainer();
public abstract EReference eContainmentFeature();
public abstract EList eContents();
public abstract EList eCrossReferences();
public abstract Object eGet(EStructuralFeature feature);
public abstract Object eGet(EStructuralFeature feature, boolean resolve);
public abstract boolean eIsProxy();
public abstract boolean eIsSet(EStructuralFeature feature);
public abstract Resource eResource();
public abstract void eSet(EStructuralFeature feature, Object newValue);
public abstract void eUnset(EStructuralFeature feature);
}

```

Hierarchy. (EObject (Notifier))

EOperation

```

org.eclipse.emf.ecore
eobject

```

The EOperation interface represents an operation, a behavior offered by a class of objects. To the methods defined by ETypedElement, it adds accessors for the eContainingClass, eExceptions, and eParameters references.

```

public interface EOperation extends ETypedElement {
// Property Accessor Methods (by property name)
    public abstract EClass getEContainingClass();
    public abstract EList getEExceptions();
    public abstract EList getEParameters();
}

```

Hierarchy. (EOperation (ETypedElement (ENamedElement (EModelElement (EObject (Notifier))))))

EPackage

```

org.eclipse.emf.ecore
eobject

```

The EPackage interface represents a package, a container for related classes and data types. It includes accessors for the nsPrefix and nsURI attributes, and for the eClassifiers, eFactoryInstance, eSubpackages, and eSuperPackage references. The getEClassifier() convenience method returns the contained class or data type whose name attribute matches the specified value.

The Descriptor and Registry inner interfaces provide a mechanism for package registration.

```

public interface EPackage extends ENamedElement {
// Public Inner Classes
    public static interface Descriptor;
    public static interface Registry;
// Property Accessor Methods (by property name)
    public abstract EList getEClassifiers();
    public abstract EFactory getEFactoryInstance();
    public abstract void setEFactoryInstance(EFactory value);
    public abstract EList getESubpackages();
    public abstract EPackage getESuperPackage();
    public abstract String getNsPrefix();
    public abstract void setNsPrefix(String value);
    public abstract String getNsURI();
}

```

```

    public abstract void setNsURI(String value);
    // Public Instance Methods
    public abstract EClassifier getEClassifier(String name);
}

```

Hierarchy: (EPackage (ENamedElement (EModelElement (EObject (Notifier)))))

EPackage.Descriptor

org.eclipse.emf.ecore

EPackage.Descriptor defines a simple package wrapper interface that can be used to defer loading a particular EPackage implementation class and initializing an instance of it. The `getEPackage()` method performs any such preparation that it requires and then returns the package.

```

public static interface EPackage.Descriptor {
    // Property Accessor Methods (by property name)
    public abstract EPackage getEPackage();
}

```

EPackage.Registry

org.eclipse.emf.ecore

EPackage.Registry is the interface, based on `java.util.Map`, for providing Ecore package registration. Packages can be registered programmatically, using the `put()` method, or through an extension to the `generated_package` extension point declared in a plug-in manifest file.

The key for an entry in the registry is the namespace URI for a package, and the value is usually the EPackage object that represents the package. However, for the purposes of registration via plug-in manifest files, the initialization of such an object can be deferred by using an EPackage.Descriptor as the value instead.

So, the `getEPackage()` method should be used to look up a package in the registry. If the retrieved object is an instance of EPackage.Descriptor, this method automatically triggers initialization of the package, returning the resulting EPackage and updating the registry to refer directly to it.

The constant `INSTANCE` field is the single package registry instance that should be used globally in EMF. It is where registrations made via plug-in manifest files appear.

```

public static interface EPackage.Registry extends Map {
    // Public Constants
    public static final EPackage.Registry INSTANCE;
    // Public Instance Methods
    public abstract EPackage getEPackage(String nsURI);
}

```

Hierarchy: (EPackage.Registry (Map))

EParameter

org.eclipse.emf.ecore eobject

The EParameter interface represents a parameter of an operation. It adds an accessor for the eOperation reference to the methods that it inherits from ETypedElement.

```
public interface EParameter extends ETypedElement {
    // Property Accessor Methods (by property name)
    public abstract EOperation getEOperation();
}
```

Hierarchy: (EParameter (ETypedElement (ENamedElement (EModelElement (EObject (Notifier)))))))

EReference

org.eclipse.emf.ecore
eobject

The EReference interface represents a reference, one direction of an association between classes. To the methods defined by EStructuralFeature, it adds accessors for the **containment**, **container**, and **resolveProxies** attributes, and for the **eOpposite** and **eReferenceType** references.

```
public interface EReference extends EStructuralFeature {
    // Property Accessor Methods (by property name)
    public abstract boolean isContainer();
    public abstract boolean isContainment();
    public abstract void setContainment(boolean value);
    public abstract EReference getEOpposite();
    public abstract void setEOpposite(EReference value);
    public abstract EClass getEReferenceType();
    public abstract boolean isResolveProxies();
    public abstract void setResolveProxies(boolean value);
}
```

Hierarchy: (EReference (EStructuralFeature (ETypedElement (ENamedElement (EModelElement (EObject (Notifier)))))))

EStructuralFeature

org.eclipse.emf.ecore
eobject

The EStructuralFeature interface represents a structural feature of a class, a single component of object state. To the methods defined by ETypedElement, it adds accessors for the **changeable**, **transient**, **unique**, **unsettable**, **volatile**, **defaultValueLiteral**, **defaultValue**, **lowerBound**, **upperBound**, **many**, and **required** attributes, and for the **eContainingClass** reference. Also, **getFeatureID()** returns the unique integer that identifies the feature within its containing class, and **getContainerClass()** returns the generated Java class corresponding to the containing class, if there is one.

The constant **UNBOUNDED_MULTIPLICITY** field defines the value of **upperBound** used to indicate unbounded multiplicity. The **Setting** inner interface is used to represent the value held by one feature of an object.

```
public interface EStructuralFeature extends ETypedElement {
    // Public Constants
    public static final int UNBOUNDED_MULTIPLICITY; =-1
    // Public Inner Classes
    public static interface Setting;
    // Property Accessor Methods (by property name)
    public abstract boolean isChangeable();
    public abstract void setChangeable(boolean value);
    public abstract Class getContainerClass();
    public abstract Object getDefaultValue();
    public abstract void setDefaultValue(Object value);
    public abstract String getDefaultValueLiteral();
```

```

public abstract void setDefaultValueLiteral(String value);
public abstract EClass getEContainingClass();
public abstract int getFeatureID();
public abstract int getLowerBound();
public abstract void setLowerBound(int value);
public abstract boolean isMany();
public abstract booleanisRequired();
public abstract boolean isTransient();
public abstract void setTransient(boolean value);
public abstract boolean isUnique();
public abstract void setUnique(boolean value);
public abstract boolean isUnsettable();
public abstract void setUnsettable(boolean value);
public abstract int getUpperBound();
public abstract void setUpperBound(int value);
public abstract boolean isVolatile();
public abstract void setVolatile(boolean value);
}

```

Hierarchy: (EStructuralFeature (ETypedElement (ENamedElement (EModelElement (EObject (Notifier)))))))

EStructuralFeature Setting

org.eclipse.emf.ecore

The EStructuralFeature Setting interface represents the value held by one feature of an object. In a sense, it represents an instance of an EStructuralFeature. The object and structural feature are available via the `getEObject()` and `getEStructuralFeature()` methods. The usual accessors, `get()`, `set()`, `unset()`, and `isSet()`, are provided for the value. They are equivalent to the corresponding EObject methods, except that the structural feature is not identified by a parameter.

The Setting interface is implemented by EcoreEList, which is the base class for the lists used to implement multiplicity-many features. So, for a multiplicity-many feature, we consider its list to be its setting.

For a multiplicity-1 feature, a single value is stored as part of the object itself, which a simple Setting implementation wraps and manipulates appropriately.

```

public static interface EStructuralFeature.Setting {
    // Property Accessor Methods (by property name)
    public abstract EObject getEObject();
    public abstract EStructuralFeature getEStructuralFeature();
    public abstract boolean isSet();
    // Public Instance Methods
    public abstract Object get(boolean resolve);
    public abstract void set(Object newValue);
    public abstract void unset();
}

```

ETypedElement

org.eclipse.emf.ecore eobject

The ETypedElement interface is extended by other Ecore interfaces that represent elements with type: structural features, operations, and parameters. It contributes accessors for an eType reference.

```
public interface ETypedElement extends ENamedElement {
    // Property Accessor Methods (by property name)
    public abstract EClassifier getEType();
    public abstract void setEType(EClassifier value);
}
```

Hierarchy: (ETypedElement (ENamedElement (EModelElement (EObject (Notifier)))))

InternalEObject

```
org.eclipse.emf.ecore
EObject
```

The `InternalEObject` interface is assumed to be implemented by any object that implements `EObject`. It is used internally by Ecore to support mechanisms for containment, bidirectional reference handshaking, dynamic objects, notification, proxy resolution, and serialization.

Strictly speaking, this interface should be considered part of the internal implementation of Ecore and, as a result, subject to change. It is included in this API summary for the benefit of clients who might find it useful, nonetheless.

The implementation of containment relies on every object keeping track of its container, which is exposed through the `EObject` interface, as well as the ID of the feature through which it is contained, which is available from `eContainerFeatureID()`. If it is contained through a one-way reference of its container, then this method returns the value computed by subtracting that opposite reference's feature ID from `EOPPOSITE_FEATURE_BASE`, instead.

The integrity of bidirectional references is maintained through a series of calls to the handshaking methods: `eInverseAdd()`, `eInverseRemove()`, `eBasicSetContainer()`, and `eBasicRemoveFromContainer()`.

To be considered dynamic, an object must have its metaobject explicitly set via `eSetClass()`. Then, it can offer dynamically implemented structural features as settings. A setting is obtained for a particular feature, whether dynamic or static, by calling `eSetting()`.

For efficiency, construction of notifications should be guarded by a test of `eNotificationRequired()`, which returns true if the object has adapters and its notification delivery has not been disabled.

The `Resource` interface represents a container for serialization; like `EObject`, it has an internal interface. The `eInternalResource()` method returns, as an instance of `Resource.Internal`, the resource that contains the object, whether directly or indirectly. The `eSetResource()` method is only called by the resource, when the object is directly added to its contents to move it from any previous container to that resource.

A proxy is an object defined in a `Resource` that has not been loaded; it can be identified as such when `eProxyURI()` returns a non-null value. A resource sets the proxy URI by calling `eSetProxyURI()`. Resources can also make use of `eURIFragmentSegment()` and `eObjectForURIFragmentSegment()` in creating path-based URI fragments to identify objects and in locating the objects referred to by such fragments.

```
public interface InternalEObject extends EObject {
    // Public Constants
    public static final int EOPPOSITE_FEATURE_BASE; =-1
    // Public Instance Methods
    public abstract int eBaseStructuralFeatureID(int derivedFeatureID, Class baseClass);
    public abstract NotificationChain eBasicRemoveFromContainer(NotificationChain ←
        notifications);
```

```

public abstract NotificationChain eBasicSetContainer(InternalEObject newContainer, int ←
newContainerFeatureID, NotificationChain notifications);
public abstract int eContainerFeatureID();
public abstract int eDerivedStructuralFeatureID(int baseFeatureID, Class baseClass);
public abstract Resource.Internal eInternalResource();
public abstract NotificationChain eInverseAdd(InternalEObject otherEnd, int featureID, ←
Class baseClass, NotificationChain notifications);
public abstract NotificationChain eInverseRemove(InternalEObject otherEnd, int featureID, ←
Class baseClass, NotificationChain notifications);
public abstract boolean eNotificationRequired();
public abstract EObject eObjectForURIFragmentSegment(String uriFragmentSegment);
public abstract URI eProxyURI();
public abstract void eSetClass(EClass eClass);
public abstract void eSetProxyURI(URI uri);
public abstract NotificationChain eSetResource(Resource.Internal resource, ←
NotificationChain notifications);
public abstract EStructuralFeature.Setting eSetting(EStructuralFeature feature);
public abstract String eURIFragmentSegment(EStructuralFeature eFeature, EObject eObject);
}

```

Hierarchy: (InternalEObject (EObject (Notifier)))

The org.eclipse.emf.ecore.plugin Package

The `org.eclipse.emf.ecore.plugin` package includes the plug-in class for the `org.eclipse.emf.ecore` plug-in. In addition to its usual duties as the plug-in's provider of resources and logging facilities, this class also reads the plug-in registry for extensions to the plug-in's extension points and provides relocatable project support when running outside of Eclipse.

EcorePlugin

`org.eclipse.emf.ecore.plugin`

`EcorePlugin` is the plug-in class for the `org.eclipse.emf.ecore` plug-in. Like all other EMF plug-in classes, it extends `EMFPlugin` and uses the Singleton design pattern, with its single instance available as the constant `INSTANCE` field.

When running within Eclipse, the Eclipse plug-in class, of type `EcorePlugin.Implementation`, is available from the static `getPlugin()` method. The same object is returned by `getPluginResourceLocator()` and is used as a delegate by the `EMFPlugin` implementations of `getBaseURL()`, `getImage()`, `getString()`, and `log()` to make use of the platform's plug-in support facilities. When running stand-alone, `getPluginResourceLocator()` returns `null`, and the base class uses its own implementations.

In addition, this plug-in class maintains a platform resource map, which is used to provide transparent support for relocatable projects when running outside of Eclipse. This map is returned by `getPlatformResourceMap()`; it maps from `String`s representing project names to absolute URLs representing file locations. The `resolvePlatformResourcePath()` method uses this map to convert a project-based path into a URI by removing its first segment, finding the corresponding URI, and resolving the remainder against it. The `handlePlatformResourceOptions()` method is a convenience for stand-alone applications; it populates the platform resource map from an array of command-line options and returns the same array with the recognized options removed.

The workspace root is returned by the `getWorkspaceRoot()` method. In Eclipse, this represents the top of the resource hierarchy in the workspace. When running outside of Eclipse, the method just returns `null`.

```

public class EcorePlugin extends EMFPlugin {
    // No Constructor
    // Public Constants
    public static final EcorePlugin INSTANCE;
    // Public Inner Classes
    public static class Implementation;
    // Public Class Methods
    public static Map<String, Object> getPlatformResourceMap();
    public static EcorePlugin.Implementation getPlugin();
    public static IWorkspaceRoot getWorkspaceRoot();
    public static String[] handlePlatformResourceOptions(String[] arguments);
    public static URI resolvePlatformResourcePath(String platformResourcePath);
    // Property Accessor Methods (by property name)
    public ResourceLocator getResourceLocator();
    Overrides:EMFPlugin
}

```

Hierarchy: Object → EMFPlugin(ResourceLocator, Logger) → EcorePlugin

EcorePlugin.Implementation

org.eclipse.emf.ecore.plugin

EcorePlugin.Implementation extends org.eclipse.core.runtime.Plugin, the base class that represents a plug-in within Eclipse, via EMFPlugin.EclipsePlugin. It is instantiated by the platform, and then used to provide platform-based implementations for most of the methods of the EcorePlugin plug-in class. It is also through this class that other Eclipse facilities, including the preference store and plug-in descriptor, can be accessed.

The platform invokes startup() when the plug-in is started. This method checks the plug-in registry for any extensions to the generated_package, extension_parser, protocol_parser, and uri_mapping extension points that may have been declared and updates the appropriate Ecore registries.

```

public static class EcorePlugin.Implementation extends EMFPlugin.EclipsePlugin {
    // Public Constructors
    public Implementation(IPluginDescriptor descriptor);
    // Public Methods Overriding Plugin
    public void startup();
}

```

Hierarchy: Object → Plugin → EMFPlugin.EclipsePlugin(ResourceLocator, Logger) → EcorePlugin.Implementation

The org.eclipse.emf.ecore.resource Package

The org.eclipse.emf.ecore.resource package defines interfaces for modeling abstract persistent resources, for working across multiple resources, for associating particular resource implementations with the URIs that identify persistent forms.

Resource

`org.eclipse.emf.ecore.resource.Notifier`

The `Resource` interface represents a persistent container of modeled objects. Although `Resource` is not itself generated from an Ecore-modeled class, resources behave very much like `EObjects`. In particular, `Resource` extends the `Notifier` interface; resources must deliver notifications for changes to the following conceptual “features”: `resourceSet`, `URI`, `contents`, `modified`, `loaded`, `trackingModification`, `errors`, and `warnings`. The interface defines its own IDs for these features as constant fields.

A resource is usually contained, along with other related resources, by an object that implements the `ResourceSet` interface. This container is available from the `getResourceSet()` method. The location to which the resource persists is specified by the `URI` that is set via `setURI()` and retrieved via `getURI()`.

An EMF object is added directly to a resource by insertion into its `contents`, which is returned by the `getContents()` method. Moreover, any object added to a containment reference of an object in the resource, directly or indirectly, is itself considered to belong indirectly to that resource. Adding to a resource's `contents` is much like adding to an ordinary containment reference; the object is automatically removed from any existing container or resource. A complete tree of objects contained by a resource, directly and indirectly, can be obtained by calling `getAllContents()`.

An object can also be retrieved according to a `URI` fragment that identifies it within the resource, often using the path-based notation defined by the `eURIFragmentSegment()` method of `InternalEObject`. The `getEObject()` method does this, whereas `getURIFragment()` returns the fragment that would identify a given object.

The persistent format that a resource produces and consumes is defined by its particular implementation of `save()` and `load()`. Options can be passed to these two methods as name-value pairs in an instance of `java.util.Map`. The options recognized by a resource are implementation-specific. When faced with errors in the persistent form, implementations should perform as complete a load as is possible. They should accumulate errors and warnings as instances of `Resource.Diagnostic`, making them available via `getErrors()` and `getWarnings()`. A resource is unloaded by calling `unload()`, which replaces the model objects it contains by proxies.

The `isLoaded()` method reports whether a resource is currently loaded. Also, the `modified` attribute can be used to track whether it has been changed since its last save or load. This attribute can be updated manually by user code or by an automatic mechanism, depending on the value of `trackingModification`. By default, automatic modification tracking is disabled, as the default implementation is expensive and ill-suited for use with undoable commands, in which an undo looks like a change.

```
public interface Resource extends Notifier {
    // Public Constants
    public static final int RESOURCE__CONTENTS; =2
    public static final int RESOURCE__ERRORS; =6
    public static final int RESOURCE__IS_LOADED; =4
    public static final int RESOURCE__IS_MODIFIED; =3
    public static final int RESOURCE__IS_TRACKING_MODIFICATION; =5
    public static final int RESOURCE__RESOURCE_SET; =0
    public static final int RESOURCE__URI; =1
    public static final int RESOURCE__WARNINGS; =7
    // Public Inner Classes
    public static interface Diagnostic;
    public static interface Factory;
    public static class IOWrappedException;
```

```

public static interface Internal {
    // Property Accessor Methods (by property name)
    public abstract TreeIterator getAllContents();
    public abstract EList getContents();
    public abstract EList getErrors();
    public abstract boolean isLoaded();
    public abstract boolean isModified();
    public abstract void setModified(boolean isModified);
    public abstract ResourceSet getResourceSet();
    public abstract boolean isTrackingModification();
    public abstract void setTrackingModification(boolean isTrackingModification);
    public abstract URI getURI();
    public abstract void setURI(URI uri);
    public abstract EList getWarnings();
}
// Public Instance Methods
public abstract EObject getEObject(String uriFragment);
public abstract String getURIFragment(EObject eObject);
public abstract void load(Map options);
public abstract void load(InputStream inputStream, Map options);
public abstract void save(Map options);
public abstract void save(OutputStream outputStream, Map options);
public abstract void unload();
}

```

Hierarchy. (Resource(Notifier))

Resource.Diagnostic

org.eclipse.emf.ecore.resource

The `Resource.Diagnostic` interface represents an error or warning generated while loading a resource from its persistent form. It defines methods to access the location of the problem, as a line and column or generically as a `String`, and a message describing the problem.

```

public static interface Resource.Diagnostic {
    // Property Accessor Methods (by property name)
    public abstract int getColumn();
    public abstract int getLine();
    public abstract String getLocation();
    public abstract String getMessage();
}

```

Resource.Factory

org.eclipse.emf.ecore.resource

The `Resource.Factory` interface is implemented by an object responsible for creating instances of a particular type of resource. Usually, a factory is registered in a registry, defined by `Resource.Factory.Registry`, for a category of URIs.

Its only method, `createResource()`, creates and returns a resource with the specified URI. The method is typically not called directly by clients, but rather by a resource set.

```

public static interface Resource.Factory {
    // Public Inner Classes
    public static interface Descriptor;
    public static interface Registry;
    // Public Instance Methods
    public abstract Resource createResource(URI uri);
}

```

Resource.Factory.Descriptor

org.eclipse.emf.ecore.resource

`Resource.Factory.Descriptor` defines a simple wrapper interface for resource factories, which can be used to defer loading a particular `Resource.Factory` implementation class and initializing an instance of it. The `createFactory()` method performs any such preparation that it requires and then returns the factory.

```
public static interface Resource.Factory.Descriptor {
    // Public Instance Methods
    public abstract Resource.Factory createFactory();
}
```

Resource.Factory.Registry

org.eclipse.emf.ecore.resource

`Resource.Factory.Registry` is the interface for providing resource factory registration in EMF. A resource factory is registered for a category of URLs. Resources with URLs that fit in that category should then be created using that factory. URLs are categorized by protocol, also known as scheme, or by extension, the portion of the last segment following its last period. Both protocols and extensions are specified using `Strings`.

Registration is performed by adding an entry representing either a protocol-to-factory or extension-to-factory mapping to the `java.util.Map` returned by `getProtocolToFactoryMap()` or `getExtensionToFactoryMap()`. The `getFactory()` method is used to look up a factory in the registry: It consults the protocol-to-factory map and then the extension-to-factory map. If it does not find an appropriate factory, it returns the factory registered in the latter against `DEFAULT_EXTENSION`. If there is no such default factory, it can use other means to find a factory or, failing that, return `null`. When running within Eclipse, a default factory is automatically registered for XMI-based serialization. Clients are strongly discouraged from changing this default.

For the purposes of registration via plug-in manifest files, the initialization of a resource factory can be deferred by registering a `Resource.Factory.Descriptor`, instead. If such an object is retrieved by `getFactory()`, the factory's initialization is automatically triggered by invoking the descriptor's `createFactory()` method and returning the result.

The constant `INSTANCE` field refers to the global resource factory registry for EMF, which is where registrations made via plug-in manifest files appear.

```
public static interface Resource.Factory.Registry {
    // Public Constants
    public static final String DEFAULT_EXTENSION;
    public static final Resource.FactoryRegistry INSTANCE;
    =""*"
    // Property Accessor Methods (by property name)
    public abstract Map getExtensionToFactoryMap();
    public abstract Map getProtocolToFactoryMap();
    // Public Instance Methods
    public abstract Resource.Factory getFactory(URI uri);
}
```

Resource.IOWrappedException

org.eclipse.emf.ecore.resource.checked

The `Resource.IOWrappedException` class is derived from `java.io.IOException` and is used to wrap another exception. It is most useful in implementing the `save()` and `load()` methods of the `Resource` interface.

The exception to wrap is supplied to the constructor and is available via `getWrappedException()`. All other methods delegate to that exception.

```
public static class Resource.IOWrappedException extends IOException {
    // Public Constructors
    public IOWrappedException(Exception exception);
    // Property Accessor Methods (by property name)
    public String getLocalizedMessage();
    public String getMessage();
    public Exception getWrappedException();
    // Public Methods Overriding Throwable
    public void printStackTrace();
    public void printStackTrace(PrintStream printStream);
    public void printStackTrace(PrintWriter printWriter);
    // Protected Instance Fields
    protected Exception exception;
}
```

Overrides:Throwable
Overrides:Throwable

Hierarchy: Object → Throwable(Serializable) → Exception → IOException → `Resource.IOWrappedException`

Resource.Internal

org.eclipse.emf.ecore.resource.notifier

The `Resource.Internal` interface is assumed to be implemented by any object that implements `Resource`. It is used internally by EMF to maintain the referential integrity of the containment relation between a resource set and a resource and to allow the resource to act on an object that is added to or removed from the resource.

Strictly speaking, this interface should be considered part of the internal implementation of EMF and, as a result, subject to change. It is included in this API summary for the benefit of clients who might find it useful nonetheless.

```
public static interface Resource.Internal extends Resource {
    // Public Instance Methods
    public abstract void attached(EObject eObject);
    public abstract NotificationChain basicSetResourceSet(ResourceSet resourceSet, ↵
    NotificationChain notifications);
    public abstract void detached(EObject eObject);
}
```

Hierarchy: (`Resource.Internal`(`Resource.Notifier`))

ResourceSet

```
org.eclipse.emf.ecore.resource
notifier
```

The `ResourceSet` interface represents a set of resources with the potential to refer to each others' objects. It provides the main client API for resource management.

Although `ResourceSet` is not, itself, generated from an Ecore-modeled class, resource sets behave very much like `EObjects`. In particular, `ResourceSet` extends the `Notifier` interface; resource sets must deliver notifications for changes to its conceptual **resources** feature. The interface defines its own ID for this feature as the constant `RESOURCE_SET__RESOURCES` field. The `getResources()` method acts as the accessor for this feature and imposes containment reference semantics. The entire tree of resources contained in the set and objects contained directly and indirectly in those resources is available via `getAllContents()`.

Generally, resources are created via a call to `createResource()`, `getResource()`, or `getEObject()`. In the simplest case, `createResource()` is passed a URI. It uses its resource factory registry to obtain a factory for that category of URI. The resource is created by this factory, added to the **resources** feature, and returned. The registry that a resource set uses is returned by `getResourceFactoryRegistry()`. If it has not been set explicitly via `setResourceFactoryRegistry()`, a default local registry implementation is returned. If this registry fails to obtain an appropriate factory for a given URI, it delegates to the global registry available as the `INSTANCE` field of the `Resource.Factory.Registry` interface.

A resource can also be demand-created and/or demand-loaded by `getResource()` or `getEObject()`, depending on the value of the `loadOnDemand` parameter. The former method returns the resource with the URI that matches the given URI, after both have been normalized using the resource set's URI converter. The latter method returns an object contained in the resource, as identified by the URI fragment. The URI converter that a set uses is returned by `getURIConverter()`. If it has not been set via `setURIConverter()`, a default implementation is returned, which applies standard URI prefix mappings.

Typically, a resource is demand-created exactly as by `createResource()`, using the URI as specified, without normalization. It is demand-loaded by calling its `load()` method, using the options returned by `getLoadOptions()`.

A resource set maintains a list of adapter factories, accessible via `getAdapterFactories()`, which is used by EcoreUtil's `getRegisteredAdapter()` method to create and return an adapter of a given type for a given object.

```
public interface ResourceSet extends Notifier {
    // Public Constants
    public static final int RESOURCE_SET__RESOURCES; =0
    // Property Accessor Methods (by property name)
    public abstract EList getAdapterFactories();
    public abstract TreeIterator getAllContents();
    public abstract Map getLoadOptions();
    public abstract Resource.Factory.Registry getResourceFactoryRegistry();
    public abstract void setResourceFactoryRegistry(Resource.Factory.Registry <
resourceFactoryRegistry);
    public abstract EList getResources();
    public abstract URIConverter getURIConverter();
    public abstract void setURIConverter(URIConverter converter);
    // Public Instance Methods
    public abstract Resource createResource(URI uri);
    public abstract EObject getEObject(URI uri, boolean loadOnDemand);
    public abstract Resource getResource(URI uri, boolean loadOnDemand);
}
```

Hierarchy: (ResourceSet (Notifier))

URIConverter

org.eclipse.emf.ecore.resource

URIConverter provides an interface for customized treatment of URIs within a resource set. The set uses it to perform URI normalization, and its resources can use it to create streams for input and output.

Normalization is performed by `normalize()` and might involve making any arbitrary change to a URI; however, it generally includes the application of the source-target prefix mappings that are defined in the map returned by `getURIMap()`. The constant `URI_MAP` field refers to the global URI map for EMF, which is where mappings specified via plug-in manifest files appear. The default URI converter implementation keeps its own local URI map, delegating to the global map only when it fails to match a source prefix.

Streams are created by `createInputStream()` and `createOutputStream()`. These methods normalize the specified URI before opening a stream for it. The default URI converter implementation provides transparent support for relocatable projects identified by a URI beginning with “platform:/resource” whether running within Eclipse or not.

```
public interface URIConverter {
    // Public Constants
    public static final Map URI_MAP;
    // Property Accessor Methods (by property name)
    public abstract Map getURIMap();
    // Public Instance Methods
    public abstract InputStream createInputStream(URI uri);
    public abstract OutputStream createOutputStream(URI uri);
    public abstract URI normalize(URI uri);
}
```

The org.eclipse.emf.ecore.util Package

The `org.eclipse.emf.ecore.util` package contains a number of miscellaneous utility classes and interfaces for use with, and for the use of, Ecore.

DelegatingEcoreEList

org.eclipse.emf.ecore.util collection

The `DelegatingEcoreEList` class provides a highly extensible, abstract base for lists implementing multiplicity-many structural features in Ecore, backed by the delegate list that a subclass provides. The mechanisms for providing this delegate are inherited from `DelegatingEList`: Either `delegateList()` can be implemented to return an object implementing the `List` interface, or the methods that access `delegateList()`, whose names all start with “delegate,” can be overridden to use some other backing.

Together with its immediate base class, `DelegatingNotifyingListImpl`, `DelegatingEcoreEList` adds the same functionality to `DelegatingEList` that `EcoreEList` adds to `BasicEList`: notification, inverse handshaking, proxy resolution, implementation of the `InternalEList` and `EStructuralFeature`.`Setting` interfaces, and efficient algorithm customization based on the particular character of a feature.

Because `DelegatingEcoreEList` does not have all of the behavior-specific subclasses that `EcoreEList` enjoys, it does not provide constant definitions for the boolean tests that define those behaviors, with the intent that they will be overridden. Instead, methods like `canContainNull()`, `isUnique()`, `isEObject()`, `isNotificationRequired()`, `hasInverse()`, `hasNavigableInverse()`, `hasManyInverse()`, `isContained()`, `hasProxies()`, and `hasInstanceClass()` are all implemented reflectively using the structural feature returned by `getEStructuralFeature()`. Note that the implementation of this method and that of `getFeatureID()` are cyclic; subclasses must take care to override one to break this cycle.

```

public abstract class DelegatingEcoreEList extends DelegatingNotifyingListImpl implements ↵
InternalEList.Unsettable, EStructuralFeature.Setting {
// Public Constructors
    public DelegatingEcoreEList(InternalEObject owner);
// Public Inner Classes
    public static abstract class Dynamic;
    public static abstract class Generic;
    public static class UnmodifiableEList;
    public static abstract class Unsettable;
// Property Accessor Methods (by property name)
    public EObject getEObject();
Implements:EstructuralFeature.Setting
    public EStructuralFeature getEstructuralFeature();
Implements:EstructuralFeature.Setting
    public Object getFeature();
Overrides:DelegatingNotifyingListImpl
    public int getFeatureID();
Overrides:DelegatingNotifyingListImpl
    public Object getNotifier();
Overrides:DelegatingNotifyingListImpl
    public boolean isSet();
Implements:InternalEList.Unsettable
// Public Methods Overriding DelegatingNotifyingListImpl
    public NotificationChain inverseAdd(Object object, NotificationChain notifications);
    public NotificationChain inverseRemove(Object object, NotificationChain notifications);
// Public Methods Overriding DelegatingEList
    public Iterator basicIterator();
    public List basicList();
    public ListIterator basicListIterator();
    public ListIterator basicListIterator(int index);
    public boolean contains(Object object);
    public int indexOf(Object object);
    public int lastIndexOf(Object object);
    public Object[] toArray();
    public Object[] toArray(Object[] array);
// Methods Implementing InternalEList.Unsettable
    public void unset();
// Methods Implementing EStructuralFeature.Setting
    public Object get(boolean resolve);
    public void set(Object newValue);
// Protected Instance Methods
    protected boolean canContainNull();
Overrides:DelegatingNotifyingListImpl
    protected NotificationImpl createNotification(int eventType, Object oldObject, Object ↵
newObject, int index);

Overrides:DelegatingNotifyingListImpl
    protected NotificationImpl createNotification(int eventType, boolean oldValue, boolean ↵
newValue);
    protected void dispatchNotification(Notification notification);
Overrides:DelegatingNotifyingListImpl
    protected EClassifier getFeatureType();
    protected EReference getInverseEReference();
    protected Class getInverseFeatureClass();
    protected int getInverseFeatureID();
    protected boolean hasInstanceClass();
    protected boolean hasInverse();
Overrides:DelegatingNotifyingListImpl

```

```

protected boolean hasManyInverse();
protected boolean hasNavigableInverse();
protected boolean hasProxies();
protected boolean isContainment();
protected boolean isEObject();
protected boolean isNotificationRequired();
Overrides:DelegatingNotifyingListImpl
    protected boolean isUnique();
Overrides:DelegatingEList
    protected Object resolve(int index, Object object);
Overrides:DelegatingEList
    protected EObject resolveProxy(EObject eObject);
    protected Object validate(int index, Object object);
Overrides:DelegatingEList
// Protected Instance Fields
    protected final InternalEObject owner;
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → DelegatingEList(EList(List), Cloneable, Serializable) → DelegatingNotifyingListImpl(NotifyingList(EList)) → DelegatingEcoreEList(InternalEList.Unsettable(InternalEList(EList))), EStructuralFeature.Setting)

DelegatingEcoreEList.Dynamic

```

org.eclipse.emf.ecore.util
collection

```

The `DelegatingEcoreEList.Dynamic` class is a single abstract base for implementing any type of multiplicity-many structural feature as a delegating list. Its constructor takes the structural feature that it is to implement as a parameter and, if needed, uses the `kind()` method from `DelegatingEcoreEList.Generic` to determine a value for `kind`. The feature is also stored so that any required information about it or its inverse can be obtained reflectively, as needed.

Because this class uses cached information about the kind of feature being implemented, it provides better performance than the basic `DelegatingEcoreEList`. However, it does not perform as well as the specialized `EcoreEList` subclasses.

```

public static abstract class DelegatingEcoreEList.Dynamic extends DelegatingEcoreEList.Generic {
{
// Public Constructors
    public Dynamic(InternalEObject owner, EStructuralFeature eStructuralFeature);
    public Dynamic(int kind, InternalEObject owner, EStructuralFeature eStructuralFeature);
// Property Accessor Methods (by property name)
    public EStructuralFeature getEStructuralFeature();
Overrides:DelegatingEcoreEList
// Protected Instance Fields
    protected EStructuralFeature eStructuralFeature;
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → DelegatingEList(EList(List), Cloneable, Serializable) → DelegatingNotifyingListImpl(NotifyingList(EList)) → DelegatingEcoreEList(InternalEList.Unsettable(InternalEList(EList))), EStructuralFeature.Setting) → DelegatingEcoreEList.Generic → DelegatingEcoreEList.Dynamic

DelegatingEcoreEList.Generic

org.eclipse.emf.ecore.util collection

`DelegatingEcoreEList.Generic` is the generic base for the flexible `DelegatingEcoreEList.Dynamic` class, which offers a performance improvement over `DelegatingEcoreEList`. Rather than implementing the boolean test methods reflectively, this class caches a `kind` field, which is set in the constructor and then consulted in these tests. The values of the constant fields are bit flags indicating the results that should be returned by the various tests; combined by a bitwise or, they characterize a feature that the list can be used to implement.

One of these bit flags, `IS_SET`, is actually set and cleared by `didChange()` and `unset()`. When `isUnsettable()` returns `true`, this flag is required to properly implement `unset()` and `isSet()`, adding an unset state for the list that is distinguishable from simply being empty.

The static `kind()` method can be used to analyze a structural feature and determine the value that characterizes it.

```
public static abstract class DelegatingEcoreEList.Generic extends DelegatingEcoreEList {
    // Public Constructors
    public Generic(int kind, InternalEObject owner);
    // Public Constants
    public static final int HAS_INSTANCE_CLASS; =0x0004
    public static final int HAS_MANY_INVERSE; =0x0010
    public static final int HAS_NAVIGABLE_INVERSE; =0x0008
    public static final int HAS_PROXYSES; =0x0800
    public static final int IS_CONTAINER; =0x0040
    public static final int IS_CONTAINMENT; =0x0020
    public static final int IS_ENUM; =0x0200
    public static final int IS_EOBJECT; =0x0400
    public static final int IS_PRIMITIVE; =0x0100
    public static final int IS_SET; =0x0001
    public static final int IS_UNIQUE; =0x0080
    public static final int IS_UNSETTABLE; =0x0002
    // Public Class Methods
    public static int kind(EStructuralFeature eStructuralFeature);
    // Property Accessor Methods (by property name)
    public boolean isSet(); Overrides:DelegatingEcoreEList
    // Public Methods Overriding DelegatingEcoreEList
    public void unset();
    // Protected Instance Methods
    protected boolean canContainNull();
    Overrides:DelegatingEcoreEList
    protected void didChange(); Overrides:DelegatingEcoreEList
    protected boolean hasInstanceClass();
    Overrides:DelegatingEcoreEList
    protected boolean hasInverse();
    Overrides:DelegatingEcoreEList
    protected boolean hasManyInverse();
    Overrides:DelegatingEcoreEList
    protected boolean hasNavigableInverse();
    Overrides:DelegatingEcoreEList
    protected boolean hasProxyses();
    Overrides:DelegatingEcoreEList
    protected boolean isContainer();
    protected boolean isContainment();
    Overrides:DelegatingEcoreEList
    protected boolean isEObject();
    Overrides:DelegatingEcoreEList
    protected boolean isUnique();
    Overrides:DelegatingEcoreEList
    protected boolean isUnsettable();
```

```

protected boolean useEquals();
// Protected Instance Fields
protected int kind;
}

Overrides:DelegatingEList

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → DelegatingEList(EList(List), Cloneable, Serializable) → DelegatingNotifyingListImpl(NotifyingList(EList)) → DelegatingEcoreEList(InternalEList.Unsettable(InternalEList(EList))), EStructuralFeature.Setting) → DelegatingEcoreEList.Generic

DelegatingEcoreEList.UnmodifiableEList

org.eclipse.emf.ecore.util collection

DelegatingEcoreEList.UnmodifiableEList is an unmodifiable version of DelegatingEcoreEList. All of the methods that would otherwise change the list instead throw an UnsupportedOperationException.

```

public static class DelegatingEcoreEList.UnmodifiableEList extends ↵
    DelegatingEList.UnmodifiableEList implements InternalEList.Unsettable, ↵
    EStructuralFeature.Setting {
    // Public Constructors
    public UnmodifiableEList(InternalEObject owner, EStructuralFeature eStructuralFeature, List ↵
        underlyingList);
    // Property Accessor Methods (by property name)
    public EObject getEObject();
    Implements:EstructuralFeature.Setting
    public EStructuralFeature getEstructuralFeature();
    Implements:EstructuralFeature.Setting
    public boolean isSet();
    Implements:InternalEList.Unsettable
    // Public Methods Overriding DelegatingEList
    public Iterator basicIterator();
    public List basicList();
    public ListIterator basicListIterator();
    public ListIterator basicListIterator(int index);
    // Methods Implementing InternalEList.Unsettable
    public void unset();
    // Methods Implementing EStructuralFeature.Setting
    public Object get(boolean resolve);
    public void set(Object newValue);
    // Methods Implementing InternalEList
    public NotificationChain basicAdd(Object object, NotificationChain notifications);
    public NotificationChain basicRemove(Object object, NotificationChain notifications);
    // Protected Instance Fields
    protected final EStructuralFeature eStructuralFeature;
    protected final InternalEObject owner;
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → DelegatingEList(EList(List), Cloneable, Serializable) → DelegatingEList.UnmodifiableEList → DelegatingEcoreEList.UnmodifiableEList(InternalEList.Unsettable(InternalEList(EList))), EStructuralFeature.Setting)

DelegatingEcoreEList.Unsettable

org.eclipse.emf.ecore.util collection

DelegatingEcoreEList.Unsettable provides an abstract base for unsettable lists backed by the delegate list that a subclass provides. It overrides `unset()`, `isSet()`, and `didChange()` to add an unset state that is distinguishable from simply being empty.

```
public static abstract class DelegatingEcoreEList.Unsettable extends DelegatingEcoreEList {
    // Public Constructors
    public Unsettable(InternalEObject owner);
    // Property Accessor Methods (by property name)
    public boolean isSet();
    // Public Methods Overriding DelegatingEcoreEList
    public void unset();                                              Overrides:DelegatingEcoreEList
    // Protected Instance Methods
    protected void didChange();                                         Overrides:DelegatingEList
    // Protected Instance Fields
    protected boolean isSet;
}
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → DelegatingEList(EList(List), Cloneable, Serializable) → DelegatingNotifyingListImpl(NotifyingList(EList)) → DelegatingEcoreEList(InternalEList.Unsettable(InternalEList(EList))), EStructuralFeature.Setting → DelegatingEcoreEList.Unsettable

EContentAdapter

org.eclipse.emf.ecore.util adapter

EContentAdapter is a convenience adapter base class that can be used to adapt all the objects in the content tree rooted by an EObject, Resource, or ResourceSet. Clients can then override the `notifyChanged()` method, which is called whenever a change happens to any of the objects in the object tree, resource, or resource set.

When installed on a root object, it automatically adds itself to the adapter list of every object in the tree. It then maintains itself as an adapter for all the contained objects as they come and go. Because it detects and deals with these structure changes in the `notifyChanged()` method, subclasses must be sure to call `super.notifyChanged()` or `selfAdapt()`, which `notifyChanged()` calls to do this work, in their overrides.

```
public class EContentAdapter extends AdapterImpl {
    // Public Constructors
    public EContentAdapter();                                              empty
    // Property Accessor Methods (by property name)
    public void setTarget(Notifier target);
    Overrides:AdapterImpl
    // Public Methods Overriding AdapterImpl
    public void notifyChanged(Notification notification);
    // Protected Instance Methods
    protected void handleContainment(Notification notification);
    protected void selfAdapt(Notification notification);
}
```

Hierarchy: Object → AdapterImpl(Adapter) → EContentAdapter

EContentsEList

`org.eclipse.emf.ecore.util
collection`

The `EContentsEList` class implements an unmodifiable, virtual `EList` that aggregates elements from one or more structural features of a modeled object. The constructor is passed the object and, optionally, the list or array of features to include. If no list or array is specified, then all of the containment references are used. Such an instance of `EContentsEList` is returned by `eContents()` in the default implementation of `EObject`.

All of the methods that would normally modify the list—`move()`, `addUnique()`, `basicAdd()`, and so on—instead throw an `UnsupportedOperationException`. The `size()` and `isEmpty()` methods compute their results from the sizes of the lists for the included structural features.

The `resolve()` method controls whether the implementation tries to resolve any proxies before returning them. A nonresolving view of the list is always available from `basicList()`; `basicIterator()` and `basicListIterator()` return nonresolving versions of the iterators.

The `isIncluded()` method returns whether a given structural feature should be included; it is used as a guard for the inclusion of elements from that structural feature in the list. The `useIsSet()` method determines whether to exclude elements from structural features for which `isSet()` returns `false`. In this implementation, `resolve()`, `isIncluded()`, and `useIsSet()` all return `true`; a subclass can override them to customize the behavior of the list.

```
public class EContentsEList extends AbstractSequentialList implements EList, InternalEList {
    // Public Constructors
    public EContentsEList(EObject eObject);
    public EContentsEList(EObject eObject, List eStructuralFeatures);
    public EContentsEList(EObject eObject, EStructuralFeature[] eStructuralFeatures);
    // Public Inner Classes
    public static interface FeatureIterator;
    public static class FeatureIteratorImpl;
    public static interface FeatureListIterator;
    public static class ResolvingFeatureIteratorImpl;
    // Property Accessor Methods (by property name)
    public boolean isEmpty();
    Overrides:AbstractCollection
    // Public Methods Overriding AbstractSequentialList
    public Iterator iterator();
    public ListIterator listIterator(int index);
    // Public Methods Overriding AbstractCollection
    public int size();
    // Methods Implementing EList
    public void move(int newPosition, Object o);
    public Object move(int newPosition, int oldPosition);
    // Methods Implementing InternalEList
    public void addUnique(Object object);
    public void addUnique(int index, Object object);
    public NotificationChain basicAdd(Object object, NotificationChain notifications);
    public Iterator basicIterator();
    public List basicList();
    public ListIterator basicListIterator();
    public ListIterator basicListIterator(int index);
    public NotificationChain basicRemove(Object object, NotificationChain notifications);
    public Object setUnique(int index, Object object);
    // Protected Instance Methods
    protected boolean isIncluded(EStructuralFeature eStructuralFeature);
    constant
    protected Iterator newIterator();
    protected ListIterator newListIterator();
    protected boolean resolve();
    protected boolean useIsSet();
    constant
    constant
```

```
// Protected Instance Fields
protected final EObject eObject;
protected final EStructuralFeature[] eStructuralFeatures;
}
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → AbstractSequentialList → EContentsEList(EList(List), InternalEList(EList))

EContentsEList.FeatureIterator

org.eclipse.emf.ecore.util

The EContentsEList.FeatureIterator interface is supported by the iterator implementations returned by the iterator() and basicIterator() methods of an EContentsEList. It adds to the standard Iterator interface a feature() method, which returns the structural feature in which the last element returned by next() was found.

```
public static interface EContentsEList.FeatureIterator extends Iterator {
// Public Instance Methods
    public abstract EStructuralFeature feature();
}
```

Hierarchy: (EContentsEList.FeatureIterator(Iterator))

EContentsEList.FeatureListIterator

org.eclipse.emf.ecore.util

The EContentsEList.FeatureListIterator interface is supported by the iterator implementations returned by the listIterator() and basicListIterator() methods of an EContentsEList. It adds to the standard ListIterator interface a feature() method, which returns the structural feature in which the last element returned by next() or previous() was found.

```
public static interface EContentsEList.FeatureListIterator extends ↪
EContentsEList.FeatureIterator, ListIterator {
}
```

Hierarchy: (EContentsEList.FeatureListIterator(EContentsEList.FeatureIterator(Iterator), ListIterator(Iterator)))

EcoreAdapterFactory

org.eclipse.emf.ecore.util

EcoreAdapterFactory is a standard generated adapter factory class, as described in Chapter 9, for the Ecore model. It implements the create() method using an instance of EcoreSwitch that dispatches to class-specific create() methods, one for each class in Ecore.

```
public class EcoreAdapterFactory extends AdapterFactoryImpl {
// Public Constructors
    public EcoreAdapterFactory();
// Public Instance Methods
    public Adapter createEAnnotationAdapter();
    public Adapter createEAttributeAdapter();
    public Adapter createEClassAdapter();
    public Adapter createEClassifierAdapter();
    public Adapter createEDataTypeAdapter();
```

constant

```

public Adapter createEEnumAdapter(); constant
public Adapter createEEnumLiteralAdapter(); constant
public Adapter createEFactoryAdapter(); constant
public Adapter createEModelElementAdapter(); constant
public Adapter createENamedElementAdapter(); constant
public Adapter createEObjectAdapter(); constant
public Adapter createEOperationAdapter(); constant
public Adapter createEPackageAdapter(); constant
public Adapter createEParameterAdapter(); constant
public Adapter createEReferenceAdapter(); constant
public Adapter createEStringToStringMapEntryAdapter(); constant
constant
public Adapter createEStructuralFeatureAdapter(); constant
public Adapter createETypedElementAdapter(); constant
// Public Methods Overriding AdapterFactoryImpl
public Adapter createAdapter(Notifier target);
public boolean isFactoryForType(Object object);
// Protected Class Fields
protected static EcorePackage modelPackage;
// Protected Instance Fields
protected EcoreSwitch modelSwitch;
}

```

Hierarchy: Object → AdapterFactoryImpl (AdapterFactory) → EcoreAdapterFactory

EcoreEList

org.eclipse.emf.ecore.util collection

The `EcoreEList` class provides a highly extensible, array-backed base for lists implementing multiplicity-many structural features in Ecore. Other list implementations extend this class, mostly overriding simple boolean tests to return constant values, efficiently customizing their behavior. As an alternative, the `Generic` and `Dynamic` inner classes extend `EcoreEList` to include a variable that characterizes a feature, and they implement the boolean tests based on this variable.

`EcoreEList` is an extension of `NotifyingListImpl`, an implementation of `NotifyingList` derived from `BasicEList`. Essentially, `NotifyingListImpl` overrides the `addUnique()`, `addAllUnique()`, `remove()`, `removeAll()`, `clear()`, `setUnique()`, and `move()` methods to perform notification and inverse handshaking. Notifications are created by `createNotification()` and dispatched by `dispatchNotification()`. Inverse handshaking should be performed in `inverseAdd()` and `inverseRemove()`, which do nothing in `NotifyingListImpl`. `EcoreEList` overrides these methods with appropriate implementations. `NotifyingListImpl` also defines `basicAdd()`, `basicRemove()`, and `basicSet()`, which do not call the inverse handshaking methods.

The type of the array allocated by `newData()` is specified as the `dataClass` parameter to the constructor. The other parameter, `owner`, is the object that owns the feature implemented by the list. It is returned by `getEObject()` and `getNotifier()` and used extensively within the list implementation.

Both `getEStructuralFeature()` and `getFeature()` return the structural feature that the list implements. Note, however, that the implementations of `getEStructuralFeature()` and `getFeatureID()` are cyclic; subclasses must take care to override one to break this cycle.

`EcoreEList` inherits the `canContainNull()`, `isUnique()`, and `useEquals()` boolean tests from `BasicEList`, as well as `hasInverse()` and `isNotificationRequired()` from `NotifyingListImpl`. It also adds a number of its own: `isEObject()`, `hasNavigableInverse()`, `hasManyInverse()`, `isContained()`, `hasProxies()`, and `hasInstanceClass()`. All of

these methods return false, except for `canContainNull()`, `useEquals()`, `isEObject()`, and `hasInstance()`. The first of these negates the result of `hasInverse()`; the rest return true. It is expected that subclasses will override whichever of these methods are necessary to properly characterize the type of features that they implement.

The implementations of `resolve()`, `toArray()`, and `contains()` depend on whether or not proxy resolution is to be performed, as determined by `hasProxies()`. Proxy resolution is handled by `resolveProxy()`, using the standard mechanism: the `resolve()` method of `EcoreUtil`. The `contains()` method is optimized for containment references and for bidirectional references with single-valued opposites.

To implement the `InternalEList` interface, the `basicList()`, `basicIterator()`, and `basicListIterator()` methods defined by `BasicEList` are made public. Also, the `EstructuralFeatureSetting` interface is implemented by the following methods: `get()` just returns the list itself; `unset()` clears the list; `set()` clears it and then adds back the elements from the `newValue` parameter, taken as a list; and `isSet()` returns whether the list is empty.

```
public class EcoreEList extends NotifyingListImpl implements InternalEList.Unsettable, ↵
EStructuralFeatureSetting {
// Public Constructors
    public EcoreEList(Class dataClass, InternalEObject owner);
// Public Inner Classes
    public static class Dynamic;
    public static class Generic;
    public static class UnmodifiableEList;
// Property Accessor Methods (by property name)
    public EObject getEObject();
Implements:EStructuralFeatureSetting
    public EStructuralFeature getEStructuralFeature();
Implements:EStructuralFeatureSetting
    public Object getFeature();
Overrides:NotifyingListImpl
    public int getFeatureID();
Overrides:NotifyingListImpl
    public Object getNotifier();
Overrides:NotifyingListImpl
    public boolean isSet();
Implements:InternalEList.Unsettable
// Public Methods Overriding NotifyingListImpl
    public NotificationChain inverseAdd(Object object, NotificationChain notifications);
    public NotificationChain inverseRemove(Object object, NotificationChain notifications);
// Public Methods Overriding BasicEList
    public Iterator basicIterator();
    public List basicList();
    public ListIterator basicListIterator();
    public ListIterator basicListIterator(int index);
    public boolean contains(Object object);
    public int indexOf(Object object);
    public int lastIndexOf(Object object);
    public Object[] toArray();
    public Object[] toArray(Object[] array);
// Methods Implementing InternalEList.Unsettable
    public void unset();
// Methods Implementing EStructuralFeatureSetting
    public Object get(boolean resolve);
    public void set(Object newValue);
// Protected Instance Methods
    protected NotificationImpl createNotification(int eventType, Object oldObject, Object ↵
newObject, int index);                                              Overrides:NotifyingListImpl
    protected NotificationImpl createNotification(int eventType, boolean oldValue, boolean ↵
newValue);
    protected void dispatchNotification(Notification notification);
Overrides:NotifyingListImpl
    protected EClassifier getFeatureType();
    protected EReference getInverseEReference();
```

```

protected Class getInverseFeatureClass();
protected int getInverseFeatureID();
protected boolean hasInstanceClass();
constant
protected boolean hasManyInverse();
constant
protected boolean hasNavigableInverse();
constant
protected boolean hasProxies();
constant
protected boolean isContainment();
constant
protected boolean isEObject();
constant
protected boolean isNotificationRequired();
Overrides:NotifyingListImpl
protected Object[] newData(int capacity);
Overrides:BasicEList
protected Object resolve(int index, Object object);
Overrides:BasicEList
protected EObject resolveProxy(EObject eObject);
protected Object validate(int index, Object object);
Overrides:BasicEList
// Protected Instance Fields
protected final Class dataClass;
protected final InternalEObject owner;
}

```

Hierarchy. Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → NotifyingListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsettable(InternalEList(EList)), EStructuralFeature.Setting)

EcoreEList.Dynamic

`org.eclipse.emf.ecore.util
collection`

The EcoreEList.Dynamic class can be used in place of any of the specialized EcoreEList subclasses to implement any type of multiplicity-many structural feature. Its constructor takes the structural feature that it is to implement as a parameter and, if needed, uses the kind() method from EcoreList.Generic to determine a value for kind. The feature is also stored, so that any required information about it or its inverse can be obtained reflectively, as needed.

The flexibility of this class, compared to the behavior-specific EcoreEList subclasses, comes at a cost of performance.

```

public static class EcoreEList.Dynamic extends EcoreEList.Generic {
// Public Constructors
public Dynamic(InternalEObject owner, EStructuralFeature eStructuralFeature);
public Dynamic(int kind, InternalEObject owner, EStructuralFeature eStructuralFeature);
public Dynamic(int kind, Class dataClass, InternalEObject owner, EStructuralFeature ←
eStructuralFeature);
// Property Accessor Methods (by property name)
public EStructuralFeature getEStructuralFeature();
Overrides:EcoreEList
// Protected Instance Fields
protected EStructuralFeature eStructuralFeature;
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsetta-ble(InternalEList(EList)), EStructuralFeature.Setting) → EcoreEL-ist.Generic → EcoreEList.Dynamic

EcoreEList.Generic

org.eclipse.emf.ecore.util collection

EcoreEList.Generic is the generic base for the very flexible EcoreEList.Dynamic class. Rather than having constant return values, its boolean test methods consult the kind field, which is set in the constructor. The values of the constant fields are bit flags indicating the results that should be returned by the various tests; combined by a bitwise or, they characterize a feature that the list can be used to implement.

One of these bit flags, IS_SET, is actually set and cleared by didChange() and unset(). When isUnsettable() returns true, this flag is required to properly implement unset() and isSet(), adding an unset state for the list that is distinguishable from simply being empty.

The static kind() method can be used to analyze a structural feature and determine the value that characterizes it. For a dynamic feature whose type has no instance class, the HAS_INSTANCE_CLASS bit is unset, so that hasInstanceClass() returns false. The implementation of validate() inherited from EcoreEList then performs a special test to dynamically ensure type safety.

```
public static class EcoreEList.Generic extends EcoreEList {
    // Public Constructors
    public Generic(int kind, Class dataClass, InternalEObject owner);
    // Public Constants
    public static final int HAS_INSTANCE_CLASS;                                =0x0004
    public static final int HAS_MANY_INVERSE;                                    =0x0010
    public static final int HAS_NAVIGABLE_INVERSE;                             =0x0008
    public static final int HAS_PROXYIES;                                       =0x0800
    public static final int IS_CONTAINER;                                       =0x0040
    public static final int IS_CONTAINMENT;                                      =0x0020
    public static final int IS_ENUM;                                            =0x0200
    public static final int IS_EOBJECT;                                         =0x0400
    public static final int IS_PRIMITIVE;                                       =0x0100
    public static final int IS_SET;                                             =0x0001
    public static final int IS_UNIQUE;                                          =0x0080
    public static final int IS_UNSETTABLE;                                     =0x0002
    // Public Class Methods
    public static int kind(EStructuralFeature eStructuralFeature);
    public static Class wrapperClassFor(Class javaClass);
    // Property Accessor Methods (by property name)
    public boolean isSet();                                                 Overrides:EcoreEList
    // Public Methods Overriding EcoreEList
    public void unset();
    // Protected Instance Methods
    protected boolean canContainNull();
    Overrides:NotifyingListImpl
    protected void didChange();                                              Overrides:BasicEList
    protected boolean hasInstanceClass();
    Overrides:EcoreEList
    protected boolean hasInverse();
    Overrides:NotifyingListImpl
    protected boolean hasManyInverse();                                         Overrides:EcoreEList
    protected boolean hasNavigableInverse();
    Overrides:EcoreEList
    protected boolean hasProxies();                                           Overrides:EcoreEList
```

```

protected boolean isContainer();
protected boolean isContainment();
protected boolean isEObject();
protected boolean isUnique();
protected boolean isUnsettable();
protected boolean useEquals();
// Protected Instance Fields
protected int kind;
}

Overrides:EcoreEList
Overrides:EcoreEList
Overrides:BasicEList
Overrides:BasicEList
Overrides:BasicEList

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → NotifyingListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsettable(InternalEList(EList))), EStructuralFeature.Setting → EcoreEList.Generic

EcoreEList.UnmodifiableEList

org.eclipse.emf.ecore.util collection

EcoreEList.UnmodifiableEList is an unmodifiable version of EcoreEList. All of the methods that would otherwise change the list instead throw an UnsupportedOperationException.

```

public static class EcoreEList.UnmodifiableEList extends BasicEList.UnmodifiableEList {
    implements InternalEList.Unsettable, EStructuralFeature.Setting {
        // Public Constructors
        public UnmodifiableEList(InternalEObject owner, EStructuralFeature eStructuralFeature, int size,
            Object[] data);
        // Property Accessor Methods (by property name)
        public EObject getEObject();
        Implements:EStructuralFeature.Setting
        public EStructuralFeature getEStructuralFeature();
        Implements:EStructuralFeature.Setting
        public boolean isSet();
        Implements:InternalEList.Unsettable
        // Public Methods Overriding BasicEList
        public Iterator basicIterator();
        public List basicList();
        public ListIterator basicListIterator();
        public ListIterator basicListIterator(int index);
        // Methods Implementing InternalEList.Unsettable
        public void unset();
        // Methods Implementing EStructuralFeature.Setting
        public Object get(boolean resolve);
        public void set(Object newValue);
        // Methods Implementing InternalEList
        public NotificationChain basicAdd(Object object, NotificationChain notifications);
        public NotificationChain basicRemove(Object object, NotificationChain notifications);
        // Protected Instance Fields
        protected final EStructuralFeature eStructuralFeature;
        protected final InternalEObject owner;
    }
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → BasicEList.UnmodifiableEList → EcoreEList.UnmodifiableEList(InternalEList.Unsettable(InternalEList(EList)), EStructuralFeature.Setting)

EcoreEMap

**org.eclipse.emf.ecore.util
collection**

EcoreEMap is a simple subclass of BasicEMap, which implements InternalEList, so that it can be used to implement an Ecore map-type feature. It supports type-safe map entries and containment semantics in its delegate list, which is an instance of the internal class DelegateEObjectContainmentEList.

The newList() method is overridden to return an anonymous BasicEList subclass backed by an array with a type specified by the entryClass field. The newEntry() method uses the appropriate factory to create an instance of the Ecore map entry class specified by entryEClass. Values for both of these fields are passed to the constructor.

The rest of the methods from the InternalEList interface are implemented using the delegate list.

```
public class EcoreEMap extends BasicEMap implements InternalEList {
    // Public Constructors
    public EcoreEMap(EClass entryEClass, Class entryClass, InternalEObject owner, int ←
        featureID);
    // Protected Inner Classes
    protected class DelegateEObjectContainmentEList;
    // Methods Implementing InternalEList
    public void addUnique(Object object);
    public void addUnique(int index, Object object);
    public NotificationChain basicAdd(Object object, NotificationChain notifications);
    public Iterator basicIterator();
    public List basicList();
    public ListIterator basicListIterator();
    public ListIterator basicListIterator(int index);
    public NotificationChain basicRemove(Object object, NotificationChain notifications);
    public Object setUnique(int index, Object object);
    // Protected Instance Methods
    protected void initializeDelegateEList();                                     Overrides:BasicEMap ←
    empty
    protected BasicEMap.Entry newEntry(int hash, Object key, Object value);      Overrides:BasicEMap
    protected BasicEList newList();                                              Overrides:BasicEMap
    // Protected Instance Fields
    protected Class entryClass;
    protected EClass entryEClass;
}
```

Hierarchy: Object → BasicEMap(EMap(EList(List(Collection)))), Cloneable, Serializable) → EcoreEMap(InternalEList(EList))

EcoreEMap.DelegateEObjectContainmentEList

**org.eclipse.emf.ecore.util
collection**

The EcoreEMap.DelegateEObjectContainmentEList class is used by EcoreEMap to implement the delegate list. It is a subclass of EObjectContainmentEList that overrides the callback methods to keep the list synchronized with the map's hash table.

```
protected class EcoreEMap.DelegateEObjectContainmentEList extends EObjectContainmentEList {
    // Public Constructors
    public DelegateEObjectContainmentEList(Class entryClass, InternalEObject owner, int ←
        featureID);
    // Protected Instance Methods
```

```

protected void didAdd(int index, Object newObject);
Overrides:BasicEList
protected void didClear(int size, Object[] oldObjects);
Overrides:BasicEList
protected void didMove(int index, Object movedObject, int oldIndex);
Overrides:BasicEList
protected void didRemove(int index, Object oldObject);
Overrides:BasicEList
protected void didSet(int index, Object newObject, Object oldObject);
Overrides:BasicEList
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyList(EList)) → EcoreEList(InternalEList.Unsetta-ble(InternalEList(EList)), EStructuralFeature.Setting) → EOObjectEList → EOObjectContainmentEList → EcoreEMap.DelegateEOObjectContainmentEList

EcoreSwitch

org.eclipse.emf.ecore.util

EcoreSwitch is a standard generated switch class for the Ecore model. It is used in the implementation of the generated EcoreAdapterFactory class. The generated doSwitch() method dispatches to class-specific case() methods, based on the type of its theEObject parameter, as described in Chapter 9.

```

public class EcoreSwitch {
// Public Constructors
    public EcoreSwitch();
// Public Instance Methods
    public Object caseEAnnotation(EAnnotation object);                                constant
    public Object caseEAttribute(EAttribute object);                                constant
    public Object caseEClass(ECClass object);                                constant
    public Object caseEClassifier(EClassifier object);                                constant
    public Object caseEDataType(EDataType object);                                constant
    public Object caseEEnum(EEEnum object);                                constant
    public Object caseEEnumLiteral(EEEnumLiteral object);                                constant
    public Object caseEFactory(EFactory object);                                constant
    public Object caseEModelElement(EModelElement object);                                constant
    public Object caseENamedElement(ENamedElement object);                                constant
    public Object caseEOperation(EOperation object);                                constant
    public Object caseEPackage(EPackage object);                                constant
    public Object caseEParameter(EParameter object);                                constant
    public Object caseEReference(EReference object);                                constant
    public Object caseEStringToStringMapEntry(Map.Entry object);                                constant
    public Object caseESTructuralFeature(ESTructuralFeature object);                                constant
    public Object caseETypedElement(ETypedElement object);                                constant
    public Object defaultCase(EOObject object);                                constant
    public Object doSwitch(EOObject theEObject);                                constant
// Protected Class Fields
    protected static EcorePackage modelPackage;
}

```

EcoreUtil

org.eclipse.emf.ecore.util

The EcoreUtil class provides a number of convenient utility methods for working with EMF objects. This class only contains static utility members and is never instantiated.

The `getExistingAdapter()` method is used to locate an existing adapter for a specified notifier and type. If no such adapter exists, it returns `null`. The `getRegisteredAdapter()` method also returns an existing adapter for a specified object and type. However, if none exists, it uses the adapter factory registered with the object's resource factory, for that type, to create one. The `getAdapter()` and `getAdapterFactory()` methods return the first adapter and adapter factory, respectively, of the specified type in the specified list.

The `getAllContents()` method returns an instance of `EcoreUtil.ContentTreeIterator` for the specified collection of EMF objects. The `copy()` and `copyAll()` methods can be used to perform deep copies, by containment tree, of one or more objects. They use `EcoreUtil.Copier` in their implementations. The `createFromString()` and `convertToString()` methods locate the specified data type's factory and then delegate to the same methods on it.

The four `isAncestor()` methods can be used to determine if an EMF object is contained, directly or indirectly, by another given EMF object, a given resource, a given resource set, or any of the objects in a collection. The `getRootContainer()` method navigates up the container hierarchy starting at the specified object, until the `EObject.eContainer()` method returns `null`, and returns the root.

The three `remove()` methods can be used to remove a value from a setting, specified directly or by the combination of the EMF object and structural feature that define it, or to remove an EMF object from its immediate container object or resource. The `replace()` methods can be used to replace an old value in a feature with a new value, or a contained object with a replacement. The `resolve()` methods resolve a proxy using the specified resource set, or the resource set of the specified resource or object. The resolved object is returned unless the proxy is not resolvable, in which case the proxy itself is returned.

The `getURI()` method can be used to access a URI for a specified object, and the `getIdentification()` method returns a unique string identification of an object. The `getID()` method returns the value of the specified object's `iD` attribute as a string. If the object's class has no `iD` attribute or the `iD` attribute is not set, it returns `null`. The `setID()` method sets the value of the specified object's `iD` attribute according to the value represented by the specified string. If this string is `null`, the attribute will be unset.

The `getObjectByType()` method returns the first object in the specified collection that is an instance of the specified `EClassifier` type, and `getObjectsByType()` returns all of the objects in the collection that are instances of the specified type. The `wrapperClassFor()` method returns the Java wrapper class for a specified primitive type, or, if passed any other Java class, that class itself.

The `setEList()` method can be used to set the contents of the specified `EList` to be exactly that of the prototype collection. This implementation minimizes the number of notifications the operation will produce. Objects already in the list are moved, missing objects are added, and extra objects are removed. If the `EList`'s contents and order are already exactly that of the prototype collection, no change is made.

```
public class EcoreUtil {
    // No Constructor
    // Public Inner Classes
    public static class ContentTreeIterator;
    public static class Copier;
    public static class CrossReferencer;
    public static class ExternalCrossReferencer;
    public static class ProxyCrossReferencer;
    public static class UnresolvedProxyCrossReferencer;
    public static class UsageCrossReferencer;
    // Public Class Methods
}
```

```

public static String convertToString(EDataType eDataType, Object value);
public static EObject copy(EObject eObject);
public static Collection copyAll(Collection eObjects);
public static EObject create(EClass eClass);
public static Object createFromString(EDataType eDataType, String literal);
public static Adapter getAdapter(List adapters, Object type);
public static AdapterFactory getAdapterFactory(List adapterFactories, Object type);
public static TreeIterator getAllContents(Collection emfObjects);
public static Adapter getExistingAdapter(Notifier notifier, Object type);
public static String getID(EObject eObject);
public static String getIdentification(EObject eObject);
public static Object getObjectByType(Collection objects, EClassifier type);
public static Collection getObjectsByType(Collection objects, EClassifier type);
public static Adapter getRegisteredAdapter(EObject eObject, Object type);
public static EObject getRootContainer(EObject eObject);
public static URI getURI(EObject eObject);                                         empty
public static boolean isAncestor(EObject ancestorEObject, EObject eObject);
public static boolean isAncestor(Resource ancestorResource, EObject eObject);
public static boolean isAncestor(ResourceSet ancestorResourceSet, EObject eObject);
public static boolean isAncestor(Collection ancestorEMFOBJECTS, EObject eObject);
public static void remove(EstructuralFeature.Setting setting, Object value);
public static void remove(EObject eObject, EstructuralFeature eStructuralFeature, Object ←
value);
public static void remove(EObject eObject);
public static void replace(EstructuralFeature.Setting setting, Object oldValue, Object ←
newValue);
public static void replace(EObject eObject, EstructuralFeature eStructuralFeature,
Object oldValue, Object newValue);
public static void replace(EObject eObject, EObject replacementEObject);
public static EObject resolve(EObject proxy, ResourceSet resourceSet);
public static EObject resolve(EObject proxy, Resource resourceContext);
public static EObject resolve(EObject proxy, EObject objectContext);
public static void setList(ELIST eList, Collection prototypeList);
public static void setID(EObject eObject, String id);
public static Class wrapperClassFor(Class javaClass);
}

```

EcoreUtil.ContentTreeIterator

`org.eclipse.emf.ecore.util collection`

EcoreUtil.ContentTreeIterator provides an implementation of TreeIterator over the contents of a collection of objects, resources, and resource sets. It implements the AbstractTreeIterator.getChildren() method by delegation to one of the protected get() methods. These methods iterate over the collections returned by the eContents(), getContents(), or getResources() method if the specified object is an EObject, Resource, or ResourceSet. Otherwise, getObjectChildren() returns an empty list iterator. The iterator returned by getResourceSetChildren() is tolerant of growth in the underlying collection, which results from demand loading of resources; the iterator walks these additional resources.

```

public static class EcoreUtil.ContentTreeIterator extends AbstractTreeIterator {
// Protected Constructors
protected ContentTreeIterator(Collection emfObjects);
// Public Methods Overriding AbstractTreeIterator
public Iterator getChildren(Object object);
// Protected Instance Methods
protected Iterator getObjectChildren(EObject eObject);
protected Iterator getObjectChildren(Object object);
protected Iterator getResourceChildren(Resource resource);
protected Iterator getResourceSetChildren(ResourceSet resourceSet);
// Protected Instance Fields
protected Collection emfObjects;
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → AbstractTreeIterator(TreeIterator(Iterator)) → EcoreUtil.ContentTreeIterator

EcoreUtil.Copier

org.eclipse.emf.ecore.util

The EcoreUtil.Copier class is used to perform a deep copy of one or more root objects, including all objects within their containment trees. This class is implemented as an extension of HashMap for convenience.

The creation of new objects is done by `copy()`, which is meant to be called repeatedly. As it proceeds, it adds the original object and its copy to the map, to be revisited later on. It then repeatedly calls `copyAttribute()`, initializing the copy's attributes, and `copyContainment()`, recursively performing the copy on containment features via `copy()` or `copyAll()`. The `copyAll()` method just calls `copy()` on each object in a collection.

The last step in the deep copy is hooking up the non-containment references, which is performed by `copyReferences()`. This method iterates through the pairs of original and copied objects in the map, repeatedly calling `copyReference()`.

Clients typically copy objects using EcoreUtil's static `copy()` and `copyAll()` methods, which are implemented using an instance of EcoreUtil.Copier.

```
public static class EcoreUtil.Copier extends HashMap {
    // Public Constructors
    public Copier();                                         empty
    // Public Instance Methods
    public EObject copy(EObject eObject);
    public Collection copyAll(Collection eObjects);
    public void copyReferences();
    // Protected Instance Methods
    protected void copyAttribute(EAttribute eAttribute, EObject eObject, EObject copiedEObject);
    protected void copyContainment(EReference eReference, EObject eObject, EObject ↴
        copiedEObject);
    protected void copyReference(EReference eReference, EObject eObject, EObject copiedEObject);
}
```

Hierarchy: Object → AbstractMap(Map) → HashMap(Map, Cloneable, Serializable)
→ EcoreUtil.Copier

EcoreUtil.CrossReferencer

org.eclipse.emf.ecore.util

ECoreUtil.CrossReferencer is a convenient utility class for locating cross-references (that is, non-containment references) in the containment tree, or trees, rooted at a specified EMF object, resource, or resource set, or arbitrary collection of all three. The result of the search is stored in the cross-referencer's base HashMap. Entries are keyed by reference targets, and their values are the collections of EStructuralFeature.Settings that identify all the source object-feature pairs referencing the keyed target.

The main search algorithm is implemented in the no-argument `crossReference()` method. It iterates over the content trees rooted by the objects in the `emfObjects` field, which contains the one or more objects with which the cross-referencer was constructed. For each object in the possibly pruned (see the following) search tree(s), `handleCrossReference()` is called. The `handleCrossReference()` method iterates over the specified object's `EObject.eCrossReferences()`, calling the three-argument `crossReference()` method with each reference. If it returns `true`, which it does by default, the target and corresponding setting are added to the result map.

This implementation is highly customizable and is used as the base class for a number of more specific cross-referencers. What actually constitutes a cross-reference can be filtered by overriding the three-argument `crossReference()` method. The default implementation, which always returns `true`, results in all non-containment references being added to the map. How deep to traverse the content structure can be controlled by overriding the `containment()` method to return `false`, which will remove, or prune, the specified object and its contents from the search tree. The `resolve()` method can be overridden to prevent objects in the tree that are proxies from being resolved during the search; by default it returns `true`. The `newCollection()` method returns the collection for a map value—that is, a settings collection. By default it returns an `ArrayList`. The `newContentsIterator()` method returns a tree iterator over the content trees of the cross-referencer's `emfObjects`. By default, it returns an instance of `ContentTreeIterator`.

Client use of this class is via the static `find()` method, which simply constructs an instance with the specified EMF objects. It then calls the `crossReference()` method to perform the search, followed by `done()`, which sets the `emfObjects` field to `null`. Finally, it returns the cross-referencer instance as the result Map. Two convenience static `print()` methods are also provided for printing a cross-reference map, or just an individual setting collection, to a stream.

```
public static class EcoreUtil.CrossReferencer extends HashMap {
    // Protected Constructors
    protected CrossReferencer(EObject eObject);
    protected CrossReferencer(Resource resource);
    protected CrossReferencer(ResourceSet resourceSet);
    protected CrossReferencer(Collection emfObjects);
    // Public Class Methods
    public static Map find(Collection emfObjects);
    public static void print(PrintStream out, Map crossReferenceMap);
    public static void print(PrintStream out, Collection settings);
    // Public Methods Overriding AbstractMap
    public String toString();
    // Protected Instance Methods
    protected boolean containment(EObject eObject);                                constant
    protected boolean crossReference(EObject eObject, EReference eReference, EObject ←
        crossReferencedEObject);                                                 constant
    protected void crossReference();                                                 constant
    protected void done();                                                       constant
    protected Collection getCollection(Object key);
    protected void handleCrossReference(EObject eObject);
    protected Collection newCollection();                                         constant
    protected TreeIterator newContentsIterator();                                 constant
    protected boolean resolve();                                                 constant
    // Protected Instance Fields
    protected Collection emfObjects;
}
```

Hierarchy: Object → AbstractMap (Map) → HashMap (Map, Cloneable, Serializable)
→ EcoreUtil.CrossReferencer

EcoreUtil.ExternalCrossReferencer

org.eclipse.emf.ecore.util

`ECoreUtil.ExternalCrossReferencer` is a convenient utility class for locating external cross-references in the containment tree, or trees, rooted at a specified EMF object, resource, or resource set, or arbitrary collection of all three. An external cross-reference is one that has a target that is not within the containment structure rooted at the object(s) being searched. This is a simple subclass of `EcoreUtil.CrossReferencer` that overrides the `crossReference()` method to only return true if `crossReferencedObject` is not contained, directly or indirectly, by one of the objects in the `CrossReferencer.emfObjects` collection.

```
public static class EcoreUtil.ExternalCrossReferencer extends EcoreUtil.CrossReferencer {
    // Protected Constructors
    protected ExternalCrossReferencer(Collection emfObjects);
    protected ExternalCrossReferencer(EObject eObject);
    protected ExternalCrossReferencer(Resource resource);
    protected ExternalCrossReferencer(ResourceSet resourceSet);
    // Public Class Methods
    public static Map find(EObject eObject);
    public static Map find(Resource resource);
    public static Map find(ResourceSet resourceSet);
    public static Map find(Collection emfObjectsToSearch);
    // Protected Instance Methods
    protected boolean crossReference(EObject eObject, EReference eReference, EObject ←
        crossReferencedEObject);
    Overrides:EcoreUtil.CrossReferencer
        protected Map findExternalCrossReferences();
}
```

Hierarchy: Object → AbstractMap (Map) → HashMap (Map, Cloneable, Serializable)
→ `EcoreUtil.CrossReferencer` → `EcoreUtil.ExternalCrossReferencer`

EcoreUtil.ProxyCrossReferencer

org.eclipse.emf.ecore.util

`ECoreUtil.ProxyCrossReferencer` is a convenient utility class for locating proxies in the containment tree, or trees, rooted at a specified EMF object, resource, or resource set, or arbitrary collection of all three. This cross-referencer overrides the `resolve()` method to return false so that it will not cause proxies to be resolved. It is a simple subclass of `EcoreUtil.CrossReferencer` that overrides the `crossReference()` method to only return true if `crossReferencedObject` is a proxy.

```
public static class EcoreUtil.ProxyCrossReferencer extends EcoreUtil.CrossReferencer {
    // Protected Constructors
    protected ProxyCrossReferencer(EObject eObject);
    protected ProxyCrossReferencer(Resource resource);
    protected ProxyCrossReferencer(ResourceSet resourceSet);
    protected ProxyCrossReferencer(Collection emfObjects);
    // Public Class Methods
    public static Map find(EObject eObject);
    public static Map find(Resource resource);
    public static Map find(ResourceSet resourceSet);
    public static Map find(Collection emfObjects);
    // Protected Instance Methods
    protected boolean crossReference(EObject eObject, EReference eReference, EObject ←
        crossReferencedEObject);
    Overrides:EcoreUtil.CrossReferencer
```

```

protected Map findProxyCrossReferences();
protected boolean resolve();                                     Overrides:EcoreUtil.CrossReferencer ←
constant
}

```

Hierarchy: Object → AbstractMap (Map) → HashMap (Map, Cloneable, Serializable)
→ EcoreUtil.CrossReferencer → EcoreUtil.ProxyCrossReferencer

EcoreUtil.UnresolvedProxyCrossReferencer

org.eclipse.emf.ecore.util

ECoreUtil.UnresolvedProxyCrossReferencer is a convenient utility class for locating unresolvable (broken) proxies in the containment tree, or trees, rooted at a specified EMF object, resource, or resource set, or arbitrary collection of all three. This cross-referencer uses an ordinary, resolving iterator, so that it will resolve nonbroken proxies during the search. It is a simple subclass of EcoreUtil.CrossReferencer that overrides the crossReference() method to only return true if crossReferencedObject is a proxy. Because the iterator is a resolving one, only broken proxies still meet this criterion.

```

public static class EcoreUtil.UnresolvedProxyCrossReferencer extends EcoreUtil.CrossReferencer ←
{
// Protected Constructors
protected UnresolvedProxyCrossReferencer(EObject eObject);
protected UnresolvedProxyCrossReferencer(Resource resource);
protected UnresolvedProxyCrossReferencer(ResourceSet resourceSet);
protected UnresolvedProxyCrossReferencer(Collection emfObjects);
// Public Class Methods
public static Map find(EObject eObject);
public static Map find(Resource resource);
public static Map find(ResourceSet resourceSet);
public static Map find(Collection emfObjects);
// Protected Instance Methods
protected boolean crossReference(EObject eObject, EReference eReference, EObject ←
crossReferencedEObject);                                     Overrides:EcoreUtil.CrossReferencer
protected Map findUnresolvedProxyCrossReferences();           ←
}

```

Hierarchy: Object → AbstractMap (Map) → HashMap (Map, Cloneable, Serializable)
→ EcoreUtil.CrossReferencer → EcoreUtil.UnresolvedProxyCrossReferencer

EcoreUtil.UsageCrossReferencer

org.eclipse.emf.ecore.util

ECoreUtil.UsageCrossReferencer is a convenient utility class for locating cross-references in the containment tree, or trees, rooted at a specified EMF object, resource, or resource set, or arbitrary collection of all three, with targets that are specific objects of interest. One or more objects of interest can be passed to the static find() or findAll() methods, respectively, which in turn construct an instance, with the search root(s), and then call their corresponding findUsage() or findAllUsage() instance methods. These methods simply set the eObjectsOfInterest field, and then invoke the usual one-argument crossReference() method in the base class, EcoreUtil.CrossReferencer, to do the search. It overrides the three-argument crossReference() method to only return true if crossReferencedObject is one of the objects in the eObjectsOfInterest collection.

```

public static class EcoreUtil.UsageCrossReferencer extends EcoreUtil.CrossReferencer {
    // Protected Constructors
    protected UsageCrossReferencer(EOBJECT eObject);
    protected UsageCrossReferencer(Resource resource);
    protected UsageCrossReferencer(ResourceSet resourceSet);
    protected UsageCrossReferencer(Collection emfObjects);
    // Public Class Methods
    public static Collection find(EOBJECT eObjectOfInterest, EOBJECT eObject);
    public static Collection find(EOBJECT eObjectOfInterest, Resource resource);
    public static Collection find(EOBJECT eObjectOfInterest, ResourceSet resourceSet);
    public static Collection find(EOBJECT eObjectOfInterest, Collection emfObjectsToSearch);
    public static Map findAll(Collection eObjectsOfInterest, EOBJECT eObject);
    public static Map findAll(Collection eObjectsOfInterest, Resource resource);
    public static Map findAll(Collection eObjectsOfInterest, ResourceSet resourceSet);
    public static Map findAll(Collection eObjectsOfInterest, Collection emfObjectsToSearch);
    // Protected Instance Methods
    protected boolean crossReference(EOBJECT eObject, EReference eReference, EOBJECT ←
        crossReferencedEOBJECT);                                                 Overrides:EcoreUtil.CrossReferencer
    protected Map findAllUsage(Collection eObjectsOfInterest);
    protected Collection findUsage(EOBJECT eObject);
    // Protected Instance Fields
    protected Collection eObjectsOfInterest;
}

```

Hierarchy: Object → AbstractMap(Map) → HashMap(Map, Cloneable, Serializable)
→ EcoreUtil.CrossReferencer → EcoreUtil.UsageCrossReferencer

ECrossReferenceEList

org.eclipse.emf.ecore.util collection

The ECrossReferenceEList class implements an unmodifiable, virtual EList that aggregates elements from one or more cross-references—that is, non-containment references—of a modeled object. The list is derived from EContentsEList, but it overrides isIncluded() to return false for containment and container references.

The constructor is passed the object and, optionally, the list or array of structural features to include. If no list or array is specified, then all of the references are used. The isIncluded() method is still applied to each of these features, resulting in the exclusion of containment references from the group that contribute elements.

Such an instance of ECrossReferenceEList is returned by eCrossReferences() in the default implementation of EObject.

```

public class ECrossReferenceEList extends EContentsEList {
    // Public Constructors
    public ECrossReferenceEList(EOBJECT eObject);
    // Protected Constructors
    protected ECrossReferenceEList(EOBJECT eObject, EStructuralFeature[] eStructuralFeatures);
    // Public Inner Classes
    public static class FeatureIteratorImpl;
    public static class ResolvingFeatureIteratorImpl;
    // Public Methods Overriding EContentsEList
    public Iterator basicIterator();
    public List basicList();
    public ListIterator basicListIterator();
    public ListIterator basicListIterator(int index);
    // Protected Instance Methods
    protected boolean isIncluded(EStructuralFeature eStructuralFeature);
    Overrides:EContentsEList
    protected ListIterator newListIterator();
    Overrides:EContentsEList
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → AbstractSequentialList → EContentsEList(EList(List), InternalEList(EList)) → ECrossReferenceEList

EDataTypeEList

**org.eclipse.emf.ecore.util
collection**

EDataTypeEList is a specialized EcoreEList subclass used to implement nonunique attributes. It overrides the `getFeatureID()` method to return the value specified as a constructor parameter. Because the type of an attribute can only be a data type, `isEObject()` is overridden to return false.

```
public class EDataTypeEList extends EcoreEList {
    // Public Constructors
    public EDataTypeEList(Class dataClass, InternalEObject owner, int featureID);
    // Public Inner Classes
    public static class Unsettable;
    // Property Accessor Methods (by property name)
    public int getFeatureID();
    Overrides:EcoreEList
    // Protected Instance Methods
    protected boolean isEObject();                                Overrides:EcoreEList ←
    constant
    // Protected Instance Fields
    protected final int featureID;
}
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyList(EList)) → EcoreEList(InternalEList.Unsettable(InternalEList(EList)), EStructuralFeature.Setting) → EDataTypeEList

EDataTypeEList.Unsettable

**org.eclipse.emf.ecore.util
collection**

EDataTypeEList.Unsettable is a specialized EcoreEList subclass used to implement non-unique, unsettable attributes. It overrides `unset()`, `isSet()`, and `didChange()` to add an unset state that is distinguishable from simply being empty.

```
public static class EDataTypeEList.Unsettable extends EDataTypeEList {
    // Public Constructors
    public Unsettable(Class dataClass, InternalEObject owner, int featureID);
    // Property Accessor Methods (by property name)
    public boolean isSet();                                         Overrides:EcoreEList
    // Public Methods Overriding EcoreEList
    public void unset();                                           Overrides:BasicEList
    // Protected Instance Methods
    protected void didChange();                                     Overrides:BasicEList
    // Protected Instance Fields
    protected boolean isSet;
}
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsettable(InternalEList(EList)), EStructuralFeature.Setting) → EDataTypeEList → EDataTypeEList.Unsettable

EDataTypeUniqueEList

**org.eclipse.emf.ecore.util
collection**

EDataTypeUniqueEList is a specialized EcoreEList subclass used to implement unique attributes. It just overrides isUnique() to return true.

```
public class EDataTypeUniqueEList extends EDataTypeEList {
    // Public Constructors
    public EDataTypeUniqueEList(Class dataClass, InternalEObject owner, int featureID);
    // Public Inner Classes
    public static class Unsettable;
    // Protected Instance Methods
    protected boolean isUnique();                                Overrides:BasicEList constant
}
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsettable(InternalEList(EList)), EStructuralFeature.Setting) → EDataTypeEList → EDataTypeUniqueEList

EDataTypeUniqueEList.Unsettable

**org.eclipse.emf.ecore.util
collection**

EDataTypeUniqueEList.Unsettable is a specialized EcoreEList subclass used to implement unique, unsettable attributes. It just overrides isUnique() to return true.

```
public static class EDataTypeUniqueEList.Unsettable extends EDataTypeEList.Unsettable {
    // Public Constructors
    public Unsettable(Class dataClass, InternalEObject owner, int featureID);
    // Protected Instance Methods
    protected boolean isUnique();                                Overrides:BasicEList constant
}
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsettable(InternalEList(EList)), EStructuralFeature.Setting) → EDataTypeEList → EDataTypeEList.Unsettable → EDataTypeUniqueEList.Unsettable

EObjectContainmentEList

**org.eclipse.emf.ecore.util
collection**

EObjectContainmentEList is a specialized EcoreEList subclass used to implement unidirectional, containment references. It overrides isContainment() and hasInverse() to return true and hasNavigableInverse() to return false.

```
public class EObjectContainmentEList extends EObjectEList {
    // Public Constructors
    public EObjectContainmentEList(Class dataClass, InternalEObject owner, int featureID);
    // Public Inner Classes
    public static class Unsettable;
    // Protected Instance Methods
    protected boolean hasInverse();
    protected boolean hasNavigableInverse();
    constant                                     Overrides:EObjectEList constant
    protected boolean isContainment();           Overrides:EcoreEList   ↵
    constant                                     Overrides:EcoreEList   ↵
    }
}
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyList(EList)) → EcoreEList(InternalEList.Unsettable(InternalEList(EList)), EStructuralFeature.Setting) → EObjectEList → EObjectContainmentEList

EObjectContainmentEList.Unsettable

**org.eclipse.emf.ecore.util
collection**

EObjectContainmentEList.Unsettable is a specialized EcoreEList subclass used to implement unidirectional, containment, unsettable references. It overrides unset(), isSet(), and didChange() to add an unset state that is distinguishable from simply being empty.

```
public static class EObjectContainmentEList.Unsettable extends EObjectContainmentEList {
    // Public Constructors
    public Unsettable(Class dataClass, InternalEObject owner, int featureID);
    // Property Accessor Methods (by property name)
    public boolean isSet();                                Overrides:EcoreEList
    // Public Methods Overriding EcoreEList
    public void unset();                                   Overrides:BasicEList
    // Protected Instance Methods
    protected void didChange();                           Overrides:BasicEList
    // Protected Instance Fields
    protected boolean isSet;
}
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyList(EList)) → EcoreEList(InternalEList.Unsettable(InternalEList(EList)), EStructuralFeature.Setting) → EObjectEList → EObjectContainmentEList → EObjectContainmentEList.Unsettable

EObjectContainmentWithInverseEList

org.eclipse.emf.ecore.util collection

`EObjectContainmentWithInverseEList` is a specialized `EcoreEList` subclass used to implement bidirectional, containment references. It overrides `getInverseFeatureID()` and `getInverseFeatureClass()` to return directly the feature ID and Java class for the opposite reference, which are specified as constructor parameters. Also, `hasNavigableInverse()` is overridden to return `true`.

```
public class EObjectContainmentWithInverseEList extends EObjectContainmentEList {
    // Public Constructors
    public EObjectContainmentWithInverseEList(Class dataClass, InternalEObject owner, int ←
    featureID, int inverseFeatureID);
    // Public Inner Classes
    public static class Unsettable;
    // Property Accessor Methods (by property name)
    public Class getInverseFeatureClass();                                Overrides:EcoreEList
    public int getInverseFeatureID();                                     Overrides:EcoreEList
    // Protected Instance Methods
    protected boolean hasNavigableInverse();                            Overrides:EObjectContainmentEList ←
    constant
    // Protected Instance Fields
    protected final int inverseFeatureID;
}
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-
gListImpl(NotifyinList(EList)) → EcoreEList(InternalEList.Unsettable(InternalEList(EList)), EStructuralFeature.Setting) → EObjectEList →
`EObjectContainmentEList` → `EObjectContainmentWithInverseEList`

EObjectContainmentWithInverseEList.Unsettable

org.eclipse.emf.ecore.util collection

`EObjectContainmentWithInverseEList.Unsettable` is a specialized `EcoreEList` sub-
class used to implement bidirectional, containment, unsettable references. It overrides `unset()`,
`isSet()`, and `didChange()` to add an unset state that is distinguishable from simply being empty.

```
public static class EObjectContainmentWithInverseEList.Unsettable extends ←
EObjectContainmentWithInverseEList {
    // Public Constructors
    public Unsettable(Class dataClass, InternalEObject owner, int featureID, int ←
    inverseFeatureID);
    // Property Accessor Methods (by property name)
    public boolean isSet();                                         Overrides:EcoreEList
    // Public Methods Overriding EcoreEList
    public void unset();                                           Overrides:BasicEList
    // Protected Instance Methods
    protected void didChange();                                     Overrides:BasicEList
    // Protected Instance Fields
    protected boolean isSet;
}
```

Hierarchy. Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsetta-ble(InternalEList(EList)), EStructuralFeature.Setting) → EObjectEList → EObjectContainmentEList → EObjectContainmentWithInverseEList → EObjectContainmentWithInverseEList.Unsettable

EObjectEList

**org.eclipse.emf.ecore.util
collection**

EObjectEList is a specialized EcoreEList subclass used to implement unidirectional, non-containment references without automatic proxy resolution. It overrides the getFeatureID() method to return the value specified as a constructor parameter. The hasInverse() and hasProxies() methods are overridden to return false. To satisfy the type and value constraints of a reference, isEObject() and isUnique() return true, and canContainNull() returns false. For efficiency, useEquals() also returns false.

```
public class EObjectEList extends EcoreEList {
    // Public Constructors
    public EObjectEList(Class dataClass, InternalEObject owner, int featureID);
    // Public Inner Classes
    public static class Unsettable;
    // Property Accessor Methods (by property name)
    public int getFeatureID();
    Overrides:EcoreEList
    // Protected Instance Methods
    protected boolean canContainNull();                                Overrides:NotifyingListImpl ←
    constant
    protected boolean hasInverse();                                     Overrides:NotifyingListImpl ←
    constant
    protected boolean hasProxies();                                    Overrides:EcoreEList ←
    constant
    protected boolean isEObject();                                     Overrides:EcoreEList ←
    constant
    protected boolean isUnique();                                      Overrides:BasicEList ←
    constant
    protected boolean useEquals();                                     Overrides:BasicEList ←
    constant
    // Protected Instance Fields
    protected final int featureID;
}
```

Hierarchy. Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsetta-ble(InternalEList(EList)), EStructuralFeature.Setting) → EObjectEList

EObjectEList.Unsettable

**org.eclipse.emf.ecore.util
collection**

EObjectEList.Unsettable is a specialized EcoreEList subclass used to implement unidirectional, non-containment, unsettable references without automatic proxy resolution. It overrides unset(), isSet(), and didChange() to add an unset state that is distinguishable from simply being empty.

```

public static class EObjectEList.Unsettable extends EObjectEList {
    // Public Constructors
    public Unsettable(Class dataClass, InternalEObject owner, int featureID);
    // Property Accessor Methods (by property name)
    public boolean isSet();                                     Overrides:EcoreEList
    // Public Methods Overriding EcoreEList
    public void unset();
    // Protected Instance Methods
    protected void didChange();                                Overrides:BasicEList
    // Protected Instance Fields
    protected boolean isSet;
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-
gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsetta-
ble(InternalEList(EList)), EStructuralFeature.Setting) → EObjectEList →
EOBJECTEList.Unsettable

EOBJECTResolvingEList

**org.eclipse.emf.ecore.util
collection**

EOBJECTResolvingEList is a specialized EcoreEList subclass used to implement unidirectional, non-containment references with automatic proxy resolution. It just overrides the hasProxies() method to return true.

```

public class EObjectResolvingEList extends EObjectEList {
    // Public Constructors
    public EObjectResolvingEList(Class dataClass, InternalEObject owner, int featureID);
    // Public Inner Classes
    public static class Unsettable;
    // Protected Instance Methods
    protected boolean hasProxies();                                Overrides:EOBJECTEList ←
    constant
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-
gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsetta-
ble(InternalEList(EList)), EStructuralFeature.Setting) → EObjectEList →
EOBJECTResolvingEList

EOBJECTResolvingEList.Unsettable

**org.eclipse.emf.ecore.util
collection**

EOBJECTResolvingEList.Unsettable is a specialized EcoreEList subclass used to implement unidirectional, non-containment, unsettable references with automatic proxy resolution. It just overrides the hasProxies() method to return true.

```

public static class EObjectResolvingEList.Unsettable extends EObjectEList.Unsettable {
    // Public Constructors
    public Unsettable(Class dataClass, InternalEObject owner, int featureID);
    // Protected Instance Methods
    protected boolean hasProxies();                                Overrides:EOBJECTEList ←
    constant
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsetta-ble(InternalEList(EList)), EStructuralFeature.Setting) → EObjectEList → EObjectEList.Unsettable → EObjectResolvingEList.Unsettable

EObjectWithInverseEList

**org.eclipse.emf.ecore.util
collection**

EObjectWithInverseEList is a specialized EcoreEList subclass used to implement bidirectional, non-containment references without automatic proxy resolution. It overrides getInverse-FeatureID() and getInverseFeatureClass() to return directly the feature ID and Java class for the opposite reference, which are specified as constructor parameters. Also, hasInverse() and hasNavigableInverse() are overridden to return true.

```
public class EObjectWithInverseEList extends EObjectEList {
    // Public Constructors
    public EObjectWithInverseEList(Class dataClass, InternalEObject owner, int featureID, int ←
        inverseFeatureID);
    // Public Inner Classes
    public static class ManyInverse;
    public static class Unsettable;
    // Property Accessor Methods (by property name)
    public Class getInverseFeatureClass();
    Overrides:EcoreEList
    public int getInverseFeatureID();
    Overrides:EcoreEList
    // Protected Instance Methods
    protected boolean hasInverse();
    Overrides:EObjectEList      ←
    constant
    protected boolean hasNavigableInverse();
    Overrides:EcoreEList      ←
    constant
    // Protected Instance Fields
    protected final int inverseFeatureID;
}
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsetta-ble(InternalEList(EList)), EStructuralFeature.Setting) → EObjectEList → EObjectWithInverseEList

EObjectWithInverseEList.ManyInverse

**org.eclipse.emf.ecore.util
collection**

EObjectWithInverseEList.ManyInverse is a specialized EcoreEList subclass used to implement bidirectional, non-containment references without automatic proxy resolution, when the opposite reference is multiplicity-many. It just overrides the hasManyInverse() method to return true.

```
public static class EObjectWithInverseEList.ManyInverse extends EObjectWithInverseEList {
    // Public Constructors
    public ManyInverse(Class dataClass, InternalEObject owner, int featureID, int ←
        inverseFeatureID);
```

```
// Protected Instance Methods
protected boolean hasManyInverse();
constant
}
```

Overrides:EcoreEList ←

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsetta-ble(InternalEList(EList)), EStructuralFeature.Setting) → EObjectEList → EObjectWithInverseEList → EObjectWithInverseEList.ManyInverse

EObjectWithInverseEList.Unsettable

**org.eclipse.emf.ecore.util
collection**

EObjectWithInverseEList.Unsettable is a specialized EcoreEList subclass used to implement bidirectional, non-containment, unsettable references without automatic proxy resolution. It overrides unset(), isSet(), and didChange() to add an unset state that is distinguishable from simply being empty.

```
public static class EObjectWithInverseEList.Unsettable extends EObjectWithInverseEList {
// Public Constructors
public Unsettable(Class dataClass, InternalEObject owner, int featureID, int ←
inverseFeatureID);
// Public Inner Classes
public static class ManyInverse;
// Property Accessor Methods (by property name)
public boolean isSet();
// Public Methods Overriding EcoreEList
public void unset();
// Protected Instance Methods
protected void didChange();
// Protected Instance Fields
protected boolean isSet;
}
```

Overrides:EcoreEList

Overrides:BasicEList

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsetta-ble(InternalEList(EList)), EStructuralFeature.Setting) → EObjectEList → EObjectWithInverseEList → EObjectWithInverseEList.Unsettable

EObjectWithInverseEList.Unsettable.ManyInverse

**org.eclipse.emf.ecore.util
collection**

EObjectWithInverseEList.Unsettable.ManyInverse is a specialized EcoreEList sub-class used to implement bidirectional, non-containment, unsettable references without automatic proxy resolution, when the opposite reference is multiplicity-many. It just overrides the hasMa-nyInverse() method to return true.

```
public static class EObjectWithInverseEList.Unsettable.ManyInverse extends ←
EOBJECTWithInverseEList.Unsettable {
// Public Constructors
public ManyInverse(Class dataClass, InternalEObject owner, int featureID, int ←
inverseFeatureID);
```

```
// Protected Instance Methods
protected boolean hasManyInverse();  
constant  
}
```

Overrides:EcoreEList ←

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsetta-ble(InternalEList(EList)), EStructuralFeature.Setting) → EOobjectEList → EOobjectWithInverseEList → EOobjectWithInverseEList.Unsettable → EOobjectWi-thInverseEList.Unsettable.ManyInverse

EOobjectWithInverseResolvingEList

org.eclipse.emf.ecore.util
collection

EOobjectWithInverseResolvingEList is a specialized EcoreEList subclass used to implement bidirectional, non-containment references with automatic proxy resolution. It just overrides the hasProxies() method to return true.

```
public class EOobjectWithInverseResolvingEList extends EOobjectWithInverseEList {
// Public Constructors
public EOobjectWithInverseResolvingEList(Class dataClass, InternalEOobject owner, int ←
featureID, int inverseFeatureID);
// Public Inner Classes
public static class ManyInverse;
public static class Unsettable;
// Protected Instance Methods
protected boolean hasProxies();  
constant  
}
```

Overrides:EOobjectEList ←

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsetta-ble(InternalEList(EList)), EStructuralFeature.Setting) → EOobjectEList → EOobjectWithInverseEList → EOobjectWithInverseResolvingEList

EOobjectWithInverseResolvingEList.ManyInverse

org.eclipse.emf.ecore.util
collection

EOobjectWithInverseResolvingEList.ManyInverse is a specialized EcoreEList subclass used to implement bidirectional, non-containment references with automatic proxy resolution, when the opposite reference is multiplicity-many. It just overrides the hasManyInverse() method to return true.

```
public static class EOobjectWithInverseResolvingEList.ManyInverse extends ←
EOobjectWithInverseResolvingEList {
// Public Constructors
public ManyInverse(Class dataClass, InternalEOobject owner, int featureID, int ←
inverseFeatureID);
// Protected Instance Methods
protected boolean hasManyInverse();  
constant  
}
```

Overrides:EcoreEList ←

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsetta-ble(InternalEList(EList)), EStructuralFeature.Setting) → EObjectEList → EObjectWithInverseEList → EObjectWithInverseResolvingEList → EObjectWi-thInverseResolvingEList.ManyInverse

EObjectWithInverseResolvingEList.Unsettable

org.eclipse.emf.ecore.util collection

EObjectWithInverseResolvingEList.Unsettable is a specialized EcoreEList subclass used to implement bidirectional, non-containment, unsettable references with automatic proxy resolution. It just overrides the hasProxies() method to return true.

```
public static class EObjectWithInverseResolvingEList.Unsettable extends ↵
EObjectWithInverseEList.Unsettable {  

// Public Constructors  

    public Unsettable(Class dataClass, InternalEObject owner, int featureID, int ↵
inverseFeatureID);  

// Public Inner Classes  

    public static class ManyInverse;  

// Protected Instance Methods  

    protected boolean hasProxies();  

constant  

} Overrides:EObjectEList ↵
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsetta-ble(InternalEList(EList)), EStructuralFeature.Setting) → EObjectEList → EObjectWithInverseEList → EObjectWithInverseEList.Unsettable → EObjectWi-thInverseResolvingEList.Unsettable

EObjectWithInverseResolvingEList.Unsettable.ManyInverse

org.eclipse.emf.ecore.util collection

EObjectWithInverseResolvingEList.Unsettable.ManyInverse is a specialized Ecor-eEList subclass used to implement bidirectional, non-containment, unsettable references with automatic proxy resolution, when the opposite reference is multiplicity-many. It just overrides the hasManyInverse() method to return true.

```
public static class EObjectWithInverseResolvingEList.Unsettable.ManyInverse extends ↵
EObjectWithInverseResolvingEList.Unsettable {  

// Public Constructors  

    public ManyInverse(Class dataClass, InternalEObject owner, int featureID, int ↵
inverseFeatureID);  

// Protected Instance Methods  

    protected boolean hasManyInverse();  

constant  

} Overrides:EcoreEList ↵
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → Notifyin-gListImpl(NotifyingList(EList)) → EcoreEList(InternalEList.Unsetta-ble(InternalEList(EList)), EStructuralFeature.Setting) → EObjectEList → EObjectWithInverseEList → EObjectWithInverseEList.Unsettable → EObjectWi-thInverseResolvingEList.Unsettable → EObjectWithInverseResolvingEL-ist.Unsettable.ManyInverse

InternalEList

**org.eclipse.emf.ecore.util
collection**

The `InternalEList` interface is implemented by every `EList` implementation used in Ecore, whether derived from `EcoreEList` or `DelegateEcoreEList`. It is used internally by Ecore to support mechanisms for bidirectional reference handshaking and basic access to the list implementation.

Strictly speaking, this interface should be considered part of the internal implementation of Ecore and, as a result, subject to change. It is included in this API summary for the benefit of clients who might find it useful nonetheless.

```
public interface InternalEList extends EList {
    // Public Inner Classes
    public static interface Unsettable;
    // Public Instance Methods
    public abstract void addUnique(Object object);
    public abstract void addUnique(int index, Object object);
    public abstract NotificationChain basicAdd(Object object, NotificationChain notifications);
    public abstract Iterator basicIterator();
    public abstract List basicList();
    public abstract ListIterator basicListIterator();
    public abstract ListIterator basicListIterator(int index);
    public abstract NotificationChain basicRemove(Object object, NotificationChain ←
    notifications);
    public abstract Object setUnique(int index, Object object);
}
```

Hierarchy: (InternalEList(EList(List(Collection))))

InternalEList.Unsettable

**org.eclipse.emf.ecore.util
collection**

The `InternalEList.Unsettable` interface is implemented by every `EList` implementation used in Ecore, whether derived from `EcoreEList` or `DelegateEcoreEList`. It is used internally by Ecore to support unsettable multivalued features.

Strictly speaking, this interface should be considered part of the internal implementation of Ecore and, as a result, subject to change. It is included in this API summary for the benefit of clients who might find it useful nonetheless.

```
public static interface InternalEList.Unsettable extends InternalEList {
    // Property Accessor Methods (by property name)
    public abstract boolean isSet();
    // Public Instance Methods
    public abstract void unset();
}
```

Hierarchy: (InternalEList.Unsettable(InternalEList(EList(List(Collection)))))

Chapter 18. The org.eclipse.emf.ecore.xmi Plug-In

The `org.eclipse.emf.ecore.xmi` plug-in provides the default resource implementations for EMF. As described in Chapter 13, it includes two resource implementations: a highly customizable XML resource and an XMI resource that extends from it. It contributes to the `extension_parser` extension point defined by the `org.eclipse.emf.ecore` plug-in, registering the XMI resource implementation class as the parser for files with a `.ecore` extension¹ and as the default parser for any other extension that is not otherwise handled (that is, for the wildcard character “`*`”).

This plug-in is not needed by models that don't require serialization or that have their own serialization formats provided by custom resource implementations.

Requires: `org.apache.xerces`, `org.eclipse.core.resources`,
`org.eclipse.core.runtime`, `org.eclipse.emf.ecore`

Extensions: `org.eclipse.emf.ecore.extension_parser`

The `org.eclipse.emf.ecore.xmi` Package

The `org.eclipse.emf.ecore.xmi` package provides interfaces for the two primary resource implementations, `XMLResource` and `XMIResource`, along with several other interfaces supporting their implementations.

It also includes a plug-in class and a set of exception classes used by the implementations. Instead of throwing these exceptions at the point where the errors are encountered, the default XML resource implementation instead adds them to its error list and proceeds to process as much of the remaining data as possible.

`ClassNotFoundException`

`org.eclipse.emf.ecore.xmi`
`checked`

`ClassNotFoundException` is used when a `load()` operation on an `XMLResource` is unable to locate an `EClass` identified in the XML. The name of the missing class and the factory whose corresponding `EPackage` was unable to find the class are provided by the `getName()` and `getFactory()` methods, respectively.

```
public class ClassNotFoundException extends XMIEException {
    // Public Constructors
    public ClassNotFoundException(String name, EFactory factory, String location, int line, int column);
```

¹In fact, it is a slight extension of the XMI resource implementation that is registered as the `.ecore` parser. This extension only sets a few XML output options that might not be desired in the general case.

```
// Property Accessor Methods (by property name)
public EFactory getFactory();
public String getName();
// Protected Instance Fields
protected String className;
protected EFactory factory;
}
```

Hierarchy: Object → Throwable(Serializable) → Exception → XMIEception(Resource.Diagnostic) → ClassNotFoundException

DanglingHREFException

org.eclipse.emf.ecore.xmi checked

DanglingHREFException is used to indicate that a referenced object that is not in a resource was encountered during a `save()` operation on an XMLResource. As described in Chapter 13, depending on the value of `OPTION_PROCESS_DANGLING_HREF` of the XMLResource, this exception is either thrown at the end of the `save()` operation, recorded in the resource's error list, or simply discarded.

```
public class DanglingHREFException extends XMIEception {
// Public Constructors
public DanglingHREFException(String message, String location, int line, int column);
}
```

Hierarchy: Object → Throwable(Serializable) → Exception → XMIEception(Resource.Diagnostic) → DanglingHREFException

FeatureNotFoundException

org.eclipse.emf.ecore.xmi checked

FeatureNotFoundException is used when a `load()` operation on an XMLResource is unable to set an attribute or reference of an object because the **EStructuralFeature** named in the XML is not in the object's class. The name of the missing feature and the object whose corresponding **EClass** does not include the feature are provided by the `getName()` and `getObject()` methods, respectively.

```
public class FeatureNotFoundException extends XMIEception {
// Public Constructors
public FeatureNotFoundException(String name, EObject object, String location, int line, int column);
// Property Accessor Methods (by property name)
public String getName();
public EObject getObject();
// Protected Instance Fields
protected String featureName;
protected EObject object;
}
```

Hierarchy: Object → Throwable(Serializable) → Exception → XMIEception(Resource.Diagnostic) → FeatureNotFoundException

IllegalValueException

org.eclipse.emf.ecore.xmi checked

IllegalValueException is used when a `load()` operation on an `XMLResource` is unable to set an attribute or reference of an object to the value specified in the XML. The object, feature, and (bad) value are available using `getObject()`, `getFeature()`, and `getValue()`, respectively. The underlying exception that was thrown by the `eSet()` method can be accessed by calling the `getWrappedException()` method from the base interface, `XMIEexception`.

```
public class IllegalValueException extends XMIEexception {
    // Public Constructors
    public IllegalValueException(EObject object, EStructuralFeature feature, Object value, ←
        Exception emfException, String location, int line, int column);
    // Property Accessor Methods (by property name)
    public EStructuralFeature getFeature();
    public EObject getObject();
    public Object getValue();
    // Protected Instance Fields
    protected EStructuralFeature feature;
    protected EObject object;
    protected Object value;
}
```

Hierarchy: Object → Throwable(Serializable) → Exception → XMIEexception(Resource.Diagnostic) → IllegalValueException

PackageNotFoundException

org.eclipse.emf.ecore.xmi checked

PackageNotFoundException is used when a `load()` operation on an `XMLResource` is unable to locate an `EPackage` whose URI is identified in the XML. The URI of the missing package is provided by the `getURI()` method.

```
public class PackageNotFoundException extends XMIEexception {
    // Public Constructors
    public PackageNotFoundException(String uri, String location, int line, int column);
    // Public Instance Methods
    public String uri();
    // Protected Instance Fields
    protected String uri;
}
```

Hierarchy: Object → Throwable(Serializable) → Exception → XMIEexception(Resource.Diagnostic) → PackageNotFoundException

UnresolvedReferenceException

org.eclipse.emf.ecore.xmi checked

UnresolvedReferenceException is used if, by the end of a `load()` operation, the target of an internal (that is, nonproxy) reference has not been found. The unresolved reference's value is available via `getReference()`.

```

public class UnresolvedReferenceException extends XMIEException {
    // Public Constructors
    public UnresolvedReferenceException(String reference, String location, int line, int column);
    // Property Accessor Methods (by property name)
    public String getReference();
    // Protected Instance Fields
    protected String reference;
}

```

Hierarchy: Object → Throwable(Serializable) → Exception → XMIEException(Resource.Diagnostic) → UnresolvedReferenceException

XMIEException

org.eclipse.emf.ecore.xmi checked

XMIEException is used to wrap exceptions for errors detected by the underlying XML parser. The original parser exception can be retrieved by calling getWrappedException(). XMIEException is also the base class of all the other exceptions in this package. Except for IllegalValueException, described earlier, in these cases getWrappedException() returns null.

The other methods in this class provide access to the details of the exception's cause and location.

```

public class XMIEException extends Exception implements Resource.Diagnostic {
    // Public Constructors
    public XMIEException(String message);
    public XMIEException(Exception exception);
    public XMIEException(String message, Exception exception);
    public XMIEException(String message, String location, int line, int column);
    public XMIEException(String message, Exception exception, String location, int line, int column);
    public XMIEException(Exception exception, String location, int line, int column);
    // Property Accessor Methods (by property name)
    public int getColumn();
    Implements:Resource.Diagnostic
    public int getLine();
    Implements:Resource.Diagnostic
    public String getLocation();
    Implements:Resource.Diagnostic
    public String getMessage();
    Overrides:Throwable
    public Exception getWrappedException();
    // Public Methods Overriding Throwable
    public void printStackTrace();
    public void printStackTrace(PrintStream printStream);
    public void printStackTrace(PrintWriter printWriter);
    // Protected Instance Fields
    protected int column;
    protected Exception exception;
    protected int line;
    protected String location;
}

```

Hierarchy: Object → Throwable(Serializable) → Exception → XMIEException(Resource.Diagnostic)

XMIPlugin

org.eclipse.emf.ecore.xmi

XMIPlugin is the plug-in class, the central provider of resources and logging capabilities, for the org.eclipse.emf.ecore.xmi plug-in. Like all other EMF plug-in classes, it extends from EMFPlugin and uses the Singleton design pattern, with its single instance available as the constant INSTANCE field.

When running within Eclipse, the Eclipse plug-in class, of type XMIPlugin.Implementation, is available from the static getPlugin() method. The same object is returned by getPluginResourceLocator() and is used as a delegate by the EMFPlugin implementations of getBaseUrl(), getImage(), getString(), and log(), to make use of the platform's plug-in support facilities. When running stand-alone, getPluginResourceLocator() returns null, and the base class uses its own implementations.

```
public final class XMIPlugin extends EMFPlugin {
    // No Constructor
    // Public Constants
    public static final XMIPlugin INSTANCE;
    // Public Inner Classes
    public static class Implementation;
    // Public Class Methods
    public static XMIPlugin.Implementation getPlugin();
    // Property Accessor Methods (by property name)
    public ResourceLocator getPluginResourceLocator();                                Overrides:EMFPlugin
}
```

Hierarchy: Object → EMFPlugin(ResourceLocator, Logger) → XMIPlugin

XMIPlugin.Implementation

org.eclipse.emf.ecore.xmi

XMIPlugin.Implementation extends org.eclipse.core.runtime.Plugin, the base class that represents a plug-in within Eclipse, via EMFPlugin.EclipsePlugin. It is instantiated by the platform, and then used to provide platform-based implementations for most of the methods of the XMIPlugin plug-in class. It is also through this class that other Eclipse facilities, including the preference store and plug-in descriptor, can be accessed.

```
public static class XMIPlugin.Implementation extends EMFPlugin.EclipsePlugin {
    // Public Constructors
    public Implementation(IPluginDescriptor descriptor);
}
```

Hierarchy: Object → Plugin → EMFPlugin.EclipsePlugin(ResourceLocator, Logger) → XMIPlugin.Implementation

XMIResource

org.eclipse.emf.ecore.xmi notifier

XMIResource is the interface for resources serialized using XMI 2.0. It provides several constants used by the implementation.

The default implementation of this interface is a simple subclass of the default `XMLResource` implementation that just instantiates XMI-specific implementations of the `XMLLoader`, `XMLSave`, and `XMLHelper` interfaces.

```
public interface XMIResource extends XMLResource {
    // Public Constants
    public static final String OPTION_USE_XMI_TYPE;           = "USE_XMI_TYPE"
    public static final String VERSION_NAME;                   = "version"
    public static final String VERSION_VALUE;                 = "2.0"
    public static final String XMI_ID;                        = "id"
    public static final String XMI_NS;                        = "xmi"
    public static final String XMI_TAG_NAME;                 = "XMI"
    public static final String XMI_URI;                      = "http://www.omg.org/XMI"
}
```

Hierarchy: (XMIResource (XMLResource (Resource (Notifier))))

XMLHelper

org.eclipse.emf.ecore.xmi

`XMLHelper` is a helper interface used by the default resource implementation classes. The default `XMLResource` implementation delegates its `save()` operation to an implementation of the `XMLSave` interface and its `load()` operation to an implementation of the `XMLLoader` interface. Both of these implementations, in turn, use an `XMLHelper` implementation to help carry out their duties.

The methods in the `XMLHelper` interface are the ones that a customized resource implementation will most likely need to override. By providing all of them in the single helper object, the most common customizations can be made by overriding just one class (that is, the default helper implementation class).

The helper methods are only called by the implementation classes, not by clients.

```
public interface XMLHelper {
    // Public Constants
    public static final int DATATYPE_IS_MANY;           =2
    public static final int DATATYPE_SINGLE;            =1
    public static final int IS_MANY_ADD;                =3
    public static final int IS_MANY_MOVE;               =4
    public static final int OTHER;                      =5
    // Public Inner Classes
    public static interface ManyReference;
    // Property Accessor Methods (by property name)
    public abstract DanglingHREFException getDanglingHREFException();
    public abstract EPackage getNoNamespacePackage();
    public abstract void setNoNamespacePackage(EPackage pkg);
    public abstract void setProcessDanglingHREF(String value);
    public abstract XMLResource getResource();
    public abstract XMLResource.XMLMap getXMLMap();
    public abstract void setXMLMap(XMLResource.XMLMap map);
    // Public Instance Methods
    public abstract void addPrefix(String prefix, String uri);
    public abstract EObject createObject(EFactory eFactory, String name);
    public abstract EStructuralFeature getFeature(EClass eClass, String namespaceURI, String name);
    public abstract int getFeatureKind(EStructuralFeature feature);
    public abstract String getHREF(EObject eObject);
    public abstract String getID(EObject eObject);
    public abstract String getIDREF(EObject eObject);
    public abstract String getJavaEncoding(String xmlEncoding);
    public abstract String getName(ENamedElement eNamedElement);
    public abstract String getQName(EClass eClass);
    public abstract String getQName(EStructuralFeature feature);
    public abstract String getURI(String prefix);
```

```

public abstract Object getValue(EObject eObject, EStructuralFeature eStructuralFeature);
public abstract String getXMLEncoding(String javaEncoding);
public abstract EPackage[] packages();
public abstract URI resolve(URI relative, URI base);
public abstract List setManyReference(XMLHelper.ManyReference reference, String location);
public abstract void setValue(EObject eObject, EStructuralFeature eStructuralFeature,
Object value, int position);
}

```

XMLHelper.ManyReference

org.eclipse.emf.ecore.xmi

`XMLHelper.ManyReference` is the type of an argument of the `setManyReference()` method in the `XMLHelper` interface. It includes an object, a many-valued feature of the object, and the values to which it should be set.

```

public static interface XMLHelper.ManyReference {
// Property Accessor Methods (by property name)
public abstract int getColumnNumber();
public abstract EStructuralFeature getFeature();
public abstract int getLineNumber();
public abstract EObject getObject();
public abstract int[] getPositions();
public abstract Object[] getValues();
}

```

XMLLoad

org.eclipse.emf.ecore.xmi

The `XMLLoad` interface is delegated to by the default `XMLResource` implementation to implement the `load()` operation. An instance of it can be thought of as “the XML loader.”

```

public interface XMLLoad {
// Public Instance Methods
public abstract void load(XMLResource resource, InputStream inputStream, Map options);
}

```

XMLResource

org.eclipse.emf.ecore.xmi **notifier**

`XMLResource` is the interface for the highly customizable XML resource implementation. It defines a number of constants used by the implementation, the most important of which identify the load and save options (`OPTION_DECLARE_XML`, `OPTION_DISABLE_NOTIFY`, and so on) supported by `XMLResource`. Options are specified by passing a map keyed by these option constants to the `load()` and `save()` methods, which are inherited from `Resource`. The purpose of each option is thoroughly described in Chapter 13. Default load and save options for a resource can be set with `setDefaultLoadOptions()` and `setDefaultSaveOptions()`.

An XML resource can associate objects with unique IDs that it uses to identify and refer to them. The `setID()` method creates such an association, adding an entry to the two maps accessed via `getEObjectToIDMap()` and `getIDToObjectMap()`, which keep track of these mappings from both directions. The `getID()` method looks up the ID for a given object.

The `XMLResource` interface includes other methods that get and set the character encoding to use and whether or not the resulting serialization is to be zipped.

```

public interface XMLResource extends Resource {
    // Public Constants
    public static final String HREF;                                     ="href"
    public static final String NIL;                                       ="nil"
    public static final String OPTION_DECLARE_XML;                      ="DECLARE_XML"
    public static final String OPTION_DISABLE_NOTIFY;                   ="DISABLE_NOTIFY"
    public static final String OPTION_ENCODING;                         ="ENCODING"
    public static final String OPTION_LINE_WIDTH;                        ="LINE_WIDTH"
    public static final String OPTION_PROCESS_DANGLING_HREF;           ="PROCESS_DANGLING_HREF"
    public static final String OPTION_PROCESS_DANGLING_HREF_DISCARD;   ="DISCARD"
    public static final String OPTION_PROCESS_DANGLING_HREF_RECORD;   ="RECORD"
    public static final String OPTION_PROCESS_DANGLING_HREF_THROW;    ="THROW"
    public static final String OPTION_SCHEMA_LOCATION;                  ="SCHEMA_LOCATION"
    public static final String OPTION_SKIP_ESCAPE;                      ="SKIP_ESCAPE"
    public static final String OPTION_USE_ENCODED_ATTRIBUTE_STYLE;     ="USE_ENCODED_ATTRIBUTE_STYLE"
    public static final String OPTION_XML_MAP;                           ="XML_MAP"
    public static final String SCHEMA_LOCATION;                          ="schemaLocation"
    public static final String TYPE;                                     ="type"
    public static final String XML_NS;                                    ="xmlns"
    public static final String XSI_NS;                                   ="xsi"
    public static final String XSI_URI;                                  ="http://www.w3.org/2001/XMLSchema-instance"
    // Public Inner Classes
    public static interface XMLInfo;
    public static interface XMLMap;
    // Property Accessor Methods (by property name)
    public abstract Map getDefaultLoadOptions();
    public abstract Map getDefaultSaveOptions();
    public abstract Map getEObjectToIDMap();
    public abstract String getEncoding();
    public abstract void setEncoding(String encoding);
    public abstract Map getIDToObjectMap();
    public abstract void setUseZip(boolean useZip);
    // Public Instance Methods
    public abstract String getID(EObject eObject);
    public abstract void setID(EObject eObject, String id);
    public abstract boolean useZip();
}

```

Hierarchy: (XMLResource(Resource(Notifier)))

XMLResource.XMLInfo

org.eclipse.emf.ecore.xmi

An implementation of the XMLResource.XMLInfo interface is used as the value of an entry in an XMLResource.XMLMap. It describes how to serialize the Ecore construct that maps to it.

By default, the object's name is used, as modeled, in serialization. A different name can be assigned by calling the `setName()` method. A target namespace can be specified with `setTargetNamespace()`. The `setRepresentation()` method controls the XML representation of a structural feature. The ATTRIBUTE, ELEMENT, or CONTENT constant is used to indicate that the feature is to be serialized in XML as an attribute, an element, or simple content, respectively.

```

public static interface XMLResource.XMLInfo {
    // Public Constants
    public static final int ATTRIBUTE;                                     =1
    public static final int CONTENT;                                       =2
    public static final int ELEMENT;                                      =0
    public static final int UNSPECIFIED;                                     =-1
    // Property Accessor Methods (by property name)
    public abstract String getName();
    public abstract void setName(String name);
    public abstract String getTargetNamespace();
}

```

```

public abstract void setTargetNamespace(String namespaceURI);
public abstract int getXMLRepresentation();
public abstract void setXMLRepresentation(int representation);
}

```

XMLResource.XMLMap

org.eclipse.emf.ecore.xmi

An implementation of the `XMLResource.XMLMap` interface is used as the value of the `XMLResource` option keyed by `OPTION_XML_MAP`. It provides a mapping from Ecore constructs to `XMLResource.XMLInfo` objects that contain information about how to represent the constructs in XML.

Mappings are added and retrieved using the `add()` and `getInfo()` methods, respectively. The `getClassifier()` and `getFeature()` methods return Ecore constructs specified by the supplied arguments, taking into account any mappings that might apply. The `getFeatures()` method returns the features of the specified class in the order in which they should be serialized.

The `setNoNamespacePackage()` method can be used to specify which package to use for XML elements that are not in an XML namespace. The `setIDAttributeName()` method sets the name of the attribute used for IDs. By default, the `XMLResource` implementation uses the name “`id`”. The `XMIResource` implementation uses the standard XMI attribute “`xmi:id`”.

```

public static interface XMLResource.XMLMap {
    // Property Accessor Methods (by property name)
    public abstract String getIDAttributeName();
    public abstract void setIDAttributeName(String name);
    public abstract EPackage getNoNamespacePackage();
    public abstract void setNoNamespacePackage(EPackage pkg);
    // Public Instance Methods
    public abstract void add(ENamedElement element, XMLResource.XMLInfo info);
    public abstract EClassifier getClassifier(String namespaceURI, String name);
    public abstract EStructuralFeature getFeature(EClass eClass, String namespaceURI, String name);
    public abstract List getFeatures(EClass eClass);
    public abstract XMLResource.XMLInfo getInfo(ENamedElement element);
}

```

XMLSave

org.eclipse.emf.ecore.xmi

The `XMLSave` interface is delegated to by the default `XMLResource` implementation to implement the `save()` operation. An instance of it can be thought of as “the XML serializer.”

```

public interface XMLSave {
    // Public Instance Methods
    public abstract void save(XMLResource resource, OutputStream outputStream, Map options);
}

```

Part VI. EMF.Edit API

Chapter 19. The org.eclipse.emf.edit Plug-In

The `org.eclipse.emf.edit` plug-in contains the Eclipse UI-independent portion of the EMF.Edit framework. A large portion of the EMF.Edit framework function is not directly concerned with the UI, so this plug-in allows reuse of this function outside of the Eclipse UI framework—that is to say, without SWT and JFace. The bulk of the functionality is provided in three subpackages, command, domain, and provider, which provide the command framework, editing domain, and item provider support, respectively.

This plug-in contributes to the `generated_package` extension point defined by the `org.eclipse.emf.ecore` plug-in, registering the Tree model to be represented by the `org.eclipse.emf.edit.tree.TreePackage` package interface.

Requires: `org.eclipse.core.resources`, `org.eclipse.emf.common(org.eclipse.core.runtime)→`, `org.eclipse.emf.ecore(org.apache.xerces, org.eclipse.core.resources)→`

Extensions: `org.eclipse.emf.ecore.generated_package`

The `org.eclipse.emf.edit` Package

The `org.eclipse.emf.edit` package provides the plug-in class for the `org.eclipse.emf.edit` plug-in.

EMFEditPlugin

org.eclipse.emf.edit

`EMFEditPlugin` is the plug-in class for the `org.eclipse.emf.edit` plug-in. Like all other EMF plug-in classes, it uses the Singleton design pattern, with its single instance available as the constant `INSTANCE` field, and extends `EMFPlugin` using the pattern described in Section 10.6.

When running within Eclipse, the Eclipse plug-in class of type `EMFEditPlugin.Implementation` is available from the static `getPlugin()` method. The same object is returned by `getPluginResourceLocator()` and is used as a delegate by the `EMFPlugin` implementations of `getBaseUrl()`, `getImage()`, `getString()`, and `log()` to make use of the platform's plug-in support facilities. When running stand-alone, `getPluginResourceLocator()` returns null, and the base class uses its own implementations.

```
public final class EMFEditPlugin extends EMFPlugin {
    // No Constructor
    // Public Constants
    public static final EMFEditPlugin INSTANCE;
    // Public Inner Classes
    public static class Implementation;
    // Public Class Methods
```

```

    public static EMFEditPlugin.Implementation getPlugin();
    // Property Accessor Methods (by property name)
    public ResourceLocator getPluginResourceLocator();
}

```

Overrides:EMFPlugin

Hierarchy: Object → EMFPlugin(ResourceLocator, Logger) → EMFEditPlugin

EMFEditPlugin.Implementation

org.eclipse.emf.edit

EMFEditPlugin.Implementation extends org.eclipse.core.runtime.Plugin, the base class that represents a plug-in within Eclipse, via EMFPlugin.EclipsePlugin. It is instantiated by the platform, and then used to provide platform-based implementations for most of the methods of the EMFEditPlugin plug-in class. It is also through this class that other Eclipse facilities, including the preference store and plug-in descriptor, can be accessed.

```

public static class EMFEditPlugin.Implementation extends EMFPlugin.EclipsePlugin {
    // Public Constructors
    public Implementation(IPluginDescriptor descriptor);
}

```

Hierarchy: Object → Plugin → EMFPlugin.EclipsePlugin(ResourceLocator, Logger) → EMFEditPlugin.Implementation

The org.eclipse.emf.edit.command Package

The org.eclipse.emf.edit.command package contains a set of generic command implementation classes that, using the reflective EObject API, provide support for setting attributes, adding and removing references, copying objects, and making other kinds of modifications to EMF objects. It also provides a number of supporting interfaces and classes.

All the command implementations in this package are fully undoable.

AbstractOverrideableCommand

org.eclipse.emf.edit.command

AbstractOverrideableCommand is a convenient base for classes that implement the OverrideableCommand interface. It implements all of the Command methods by delegating to an override command, if one exists. Otherwise it delegates each method to a corresponding do() method from the OverrideableCommand interface.

Subclasses should provide implementations of the do() methods instead of the base Command methods, which have final implementations here.

```

public abstract class AbstractOverrideableCommand extends AbstractCommand implements ↪
OverrideableCommand {
    // Protected Constructors
    protected AbstractOverrideableCommand(EditingDomain domain);
    protected AbstractOverrideableCommand(EditingDomain domain, String label);
    protected AbstractOverrideableCommand(EditingDomain domain, String label, String ↪
description);
    // Public Class Methods
    public static EList getOwnerList(EObject owner, EStructuralFeature feature);
    // Property Accessor Methods (by property name)
    public final Collection getAffectedObjects();
}

```

Overrides:AbstractCommand

```

public final Collection getChildrenToCopy();
public final String getDescription();
public EditingDomain getDomain();
public final String getLabel();
public Command getOverride();
public void setOverride(Command overrideCommand);
Implements:OverrideableCommand
public final Collection getResult();
// Public Instance Methods
public Collection doGetChildrenToCopy();
// Public Methods Overriding AbstractCommand
public final boolean canExecute();
public final boolean canUndo();
public final void dispose();
public final void execute();
public final void redo();
public String toString();
public final void undo();
// Methods Implementing OverrideableCommand
public boolean doCanExecute();
public boolean doCanUndo();
public void doDispose();
public abstract void doExecute();
public Collection doGetAffectedObjects();
public String doGetDescription();
public String doGetLabel();
public Collection doGetResult();
public abstract void doRedo();
public abstract void doUndo();
// Protected Instance Fields
protected EditingDomain domain;
protected Command overrideCommand;
}

```

Hierarchy: Object → AbstractCommand(Command) → AbstractOverrideableCommand(OverrideableCommand(Command))

AddCommand

org.eclipse.emf.edit.command

An AddCommand is used to add one or more objects to a many-valued attribute or reference of an owner object, or directly to a specified list (for example, the contents list of a resource). The objects are added at the specified index or to the end of the list if the index value is CommandParameter.NO_INDEX.

The canExecute() method, via prepare(), returns false if any of the collection of objects is not of the correct type, the index is out of range, or a container is being added to its own contents. The getAffectedObjects() method returns the collection of added objects after execute() and redo(), and the owner object after undo(). The getResult() method always returns the collection of added objects.

Like all the commands in this package, an AddCommand is usually created using one of its static create() methods, which delegates to an EditingDomain for the actual command creation.

```

public class AddCommand extends AbstractOverrideableCommand {
// Public Constructors
    public AddCommand(EditingDomain domain, EObject owner, EStructuralFeature feature, Object ←
value);
    public AddCommand(EditingDomain domain, EObject owner, EStructuralFeature feature, Object ←
value, int index);
    public AddCommand(EditingDomain domain, EObject owner, EStructuralFeature feature, ←
Collection collection);
    public AddCommand(EditingDomain domain, EObject owner, EStructuralFeature feature, ←

```

```

Collection collection, int index);
    public AddCommand(EditingDomain domain, EList list, Object value);
    public AddCommand(EditingDomain domain, EList list, Object value, int index);
    public AddCommand(EditingDomain domain, EList list, Collection collection);
    public AddCommand(EditingDomain domain, EList list, Collection collection, int index);
// Protected Constants
    protected static final String DESCRIPTION;
    protected static final String DESCRIPTION_FOR_LIST;
    protected static final String LABEL;
// Public Class Methods
    public static Command create(EditingDomain domain, Object owner, Object feature, Object ←
value);
    public static Command create(EditingDomain domain, Object owner, Object feature, Object ←
value, int index);
    public static Command create(EditingDomain domain, Object owner, Object feature, Collection ←
collection);
    public static Command create(EditingDomain domain, Object owner, Object feature, Collection ←
collection, int index);
// Property Accessor Methods (by property name)
    public Collection getCollection();
    public EStructuralFeature getFeature();
    public int getIndex();
    public EObject getOwner();
    public EList getOwnerList();
// Public Methods Overriding AbstractOverrideableCommand
    public void doExecute();
    public Collection doGetAffectedObjects();
    public Collection doGetResult();
    public void doRedo();
    public void doUndo();
    public String toString();
// Protected Instance Methods
    protected boolean prepare();                                     Overrides:AbstractCommand
// Protected Instance Fields
    protected Collection affectedObjects;
    protected Collection collection;
    protected EStructuralFeature feature;
    protected int index;
    protected EObject owner;
    protected EList ownerList;
}

```

Hierarchy: Object → AbstractCommand(Command) → AbstractOverrideableCommand(OverrideableCommand(Command)) → AddCommand

ChildrenToCopyProvider

org.eclipse.emf.edit.command

ChildrenToCopyProvider is implemented by the CreateCopyCommand, and any potential subclasses of it, to identify the children to copy during a deep copy.

```

public interface ChildrenToCopyProvider {
// Property Accessor Methods (by property name)
    public abstract Collection getChildrenToCopy();
}

```

CommandActionDelegate

org.eclipse.emf.edit.command

The CommandActionDelegate interface is implemented by commands that are used to implement a selection-based menu or toolbar action. The appearance (text, icon, and so on) and enablement state of the action are accessed using this interface, whenever the selection is changed.

```

public interface CommandActionDelegate {
    // Property Accessor Methods (by property name)
    public abstract String getDescription();
    public abstract Object getImage();
    public abstract String getText();
    public abstract String getToolTipText();
    // Public Instance Methods
    public abstract boolean canExecute();
}

```

CommandParameter

org.eclipse.emf.edit.command

A `CommandParameter` is used to pass the parameters needed to create a command using the generic `createCommand()` method of an `EditingDomain`. It provides complete support for the kinds of parameters used to construct all of the built-in EMF.Edit commands, but also serves as a convenient base class for passing command parameters for other kinds of commands.

This implementation provides support for passing a command owner object, a feature of that object, an index of the feature (if many-valued), and a single value and/or collection of values. Depending on the command being created, some of these fields might be unused.

To allow greater flexibility in the framework, the `owner`, `feature`, and `value` are of type `java.lang.Object`, instead of EMF-specific types. Several convenience methods are provided to support the common case, where they are EMF types. For example, the `getEReference()` method returns the downcasted `feature` field if it is an instance of `EReference` and null otherwise.

```

public class CommandParameter {
    // Public Constructors
    public CommandParameter(Object owner);
    public CommandParameter(Object owner, Object feature, Object value);
    public CommandParameter(Object owner, Object feature, Object value, int index);
    public CommandParameter(Object owner, Object feature, Collection collection);
    public CommandParameter(Object owner, Object feature, Collection collection, int index);
    public CommandParameter(Object owner, Object feature, Object value, Collection collection);
    public CommandParameter(Object owner, Object feature, Object value, Collection collection, ←
    int index);
    // Public Constants
    public static final int NO_INDEX; =-1
    // Public Class Methods
    public static String collectionToString(Collection collection);
    // Property Accessor Methods (by property name)
    public Collection getCollection();
    public EAttribute getEAttribute();
    public EObject getEObject();
    public EReference getEReference();
    public EStructuralFeature getEStructuralFeature();
    public EObject getEValue();
    public Object getFeature();
    public int getIndex();
    public List getList();
    public Object getOwner();
    public void setOwner(Object owner);
    public EList getOwnerList();
    public Collection getParameters();
    public Object getValue();
    // Public Methods Overriding Object
    public String toString();
    // Public Instance Fields
    public Collection collection;
    public Object feature;
}

```

```

    public int index;
    public Object owner;
    public Object value;
}

```

CopyCommand

`org.eclipse.emf.edit.command`

The `CopyCommand` performs a deep copy of one or more owner (root) objects. The depth of copy and the initialization details at each level are highly customizable. By default the set of objects within the tree structure implied by the EMF containment hierarchy is copied. If the target of a reference being copied is within the set of copied objects, then the copy's reference target is the copy of the original reference's target. Otherwise, if the reference is one way, its copy is set to the same object as the original. External bidirectional references are never copied.

The copy command is implemented as a compound command consisting of one or more `CreateCopyCommands`, which create each atomic object in the copy, followed by a corresponding number of `InitializeCopyCommands`, which later initialize them. A `CopyCommand.Helper` is used to maintain the set of copied objects along with their corresponding copies. A `CopyCommand` is typically customized by subclassing and overriding methods of either or both of the `CreateCopyCommand` and `InitializeCopyCommand` created at specific nodes in the copy tree, as opposed to overriding the `CopyCommand` itself.

Like all the commands in this package, a `CopyCommand` is usually created using one of its static `create()` methods, which delegates to an `EditingDomain` for the actual command creation.

```

public class CopyCommand extends StrictCompoundCommand {
    // Public Constructors
    public CopyCommand(EditingDomain domain, EObject owner, CopyCommand.Helper copyHelper);
    public CopyCommand(EditingDomain domain, EObject owner, CopyCommand.Helper copyHelper, ←
        boolean optimize);
    // Protected Constants
    protected static final String DESCRIPTION;
    protected static final String LABEL;
    // Public Inner Classes
    public static class Helper;
    // Public Class Methods
    public static Command create(EditingDomain domain, Object owner);
    public static Command create(EditingDomain domain, Collection collection);
    // Public Methods Overriding StrictCompoundCommand
    public void execute();
    public String toString();
    // Public Methods Overriding AbstractCommand
    public boolean canExecute();
    // Protected Instance Methods
    protected void addCreateCopyCommands(CompoundCommand compoundCommand, EObject object);
    protected boolean prepare();                                     Overrides: StrictCompoundCommand
    // Protected Instance Fields
    protected CopyCommand.Helper copyHelper;
    protected EditingDomain domain;
    protected boolean optimize;
    protected EObject owner;
}

```

Hierarchy: Object → AbstractCommand(Command) → CompoundCommand → StrictCompoundCommand → CopyCommand

CopyCommand.Helper

org.eclipse.emf.edit.command

`CopyCommand.Helper` is used by a `CopyCommand` to maintain the complete set of copied objects along with their associated copies. The `getCopyTarget()` method is used to determine the target of non-containment references that are being copied. It returns the copy of the original reference's target if one exists, otherwise the original target itself if the reference is one-way or `null` if it is bidirectional.

The `initializationIterator()` method returns an iterator over the copies, in the order that the `InitializeCopyCommands` should be executed by the `CopyCommand`.

```
public static class CopyCommand.Helper extends HashMap {
    // Public Constructors
    public Helper();
    // Public Instance Methods
    public EObject getCopy(EObject object);
    public EObject getCopyTarget(EObject target, boolean copyRequired);
    public Iterator initializationIterator();
    // Public Methods Overriding HashMap
    public Object put(Object key, Object value);
    public Object remove(Object key);
    // Protected Instance Fields
    protected ArrayList initializationList;
}
```

Hierarchy: `Object` → `AbstractMap(Map)` → `HashMap(Map, Cloneable, Serializable)`
 → `CopyCommand.Helper`

CopyToClipboardCommand

org.eclipse.emf.edit.command

`CopyToClipboardCommand` delegates to and works exactly like a `CopyCommand` but sets the copy result to the clipboard maintained by the `EditingDomain`. The `doExecute()` method executes an ordinary `CopyCommand` to perform the actual copy, and then calls `setClipboard()` on the editing domain with the copy result. The previous clipboard value is also saved, in `oldClipboard`, so that it can be restored if the command is undone.

```
public class CopyToClipboardCommand extends AbstractOverrideableCommand {
    // Public Constructors
    public CopyToClipboardCommand(EditingDomain domain, Collection collection);
    // Protected Constants
    protected static final String DESCRIPTION;
    protected static final String LABEL;
    // Public Class Methods
    public static Command create(EditingDomain domain, Collection collection);
    public static Command create(EditingDomain domain, Object owner);
    // Property Accessor Methods (by property name)
    public Collection getSourceObjects();
    // Public Methods Overriding AbstractOverrideableCommand
    public void doDispose();
    public void doExecute();
    public Collection doGetAffectedObjects();
    public Collection doGetResult();
    public void doRedo();
    public void doUndo();
    public String toString();
    // Protected Instance Methods
    protected boolean prepare();
    // Protected Instance Fields
}
```

Overrides: `AbstractCommand`

```

protected Command copyCommand;
protected Collection oldClipboard;
protected Collection sourceObjects;
}

```

Hierarchy: Object → AbstractCommand(Command) → AbstractOverrideableCommand(OverrideableCommand(Command)) → CopyToClipboardCommand

CreateChildCommand

org.eclipse.emf.edit.command

The CreateChildCommand wraps an AddCommand or SetCommand, depending on the feature multiplicity, to provide the higher level operation of “creating” an appropriate child object and adding it to a feature of an owner object. The new child object is created, typically by the owner’s item provider, before this command is created and specified in the newChildDescriptor argument of the static create() method.

Once constructed, this class essentially just creates and delegates to the appropriate lower level EMF (add or set) command, and delegates matters of appearance (text, icon, result) to a CreateChildCommand.Helper, typically the item provider of the command owner, so that it can be handled correctly for the given model.

CreateChildCommand is used to implement both child and sibling creation actions. To support this, a selection (different from the owner) can be explicitly specified, so that the getAffectedObjects() method returns it after undo() instead of the owner (that is, the normal affected object produced by the wrapped add or set command).

```

public class CreateChildCommand extends CommandWrapper implements CommandActionDelegate {
// Public Constructors
    public CreateChildCommand(EditingDomain domain, EObject owner, EReference feature, EObject ←
child, Collection selection);
    public CreateChildCommand(EditingDomain domain, EObject owner, EReference feature, EObject ←
child, Collection selection, CreateChildCommand.Helper helper);
    public CreateChildCommand(EditingDomain domain, EObject owner, EReference feature, EObject ←
child, int index, Collection selection);
    public CreateChildCommand(EditingDomain domain, EObject owner, EReference feature, EObject ←
child, int index, Collection selection, CreateChildCommand.Helper helper);
// Protected Constants
    protected static final int NO_INDEX;                                         =-1
// Public Inner Classes
    public static interface Helper;
// Public Class Methods
    public static Command create(EditingDomain domain, Object owner, Object newChildDescriptor, ←
Collection selection);
// Property Accessor Methods (by property name)
    public Collection getAffectedObjects();
Overrides:CommandWrapper
    public String getDescription();
    public Object getImage();
    public Collection getResult();
    public String getText();
    public String getToolTipText();
Implements:CommandActionDelegate
// Public Methods Overriding CommandWrapper
    public void execute();
    public void redo();
    public String toString();
    public void undo();
// Protected Instance Methods
    protected Command createCommand();
Overrides:CommandWrapper
// Protected Instance Fields
    protected Collection affectedObjects;
    protected EObject child;

```

```

protected EditingDomain domain;
protected EReference feature;
protected CreateChildCommand.Helper helper;
protected int index;
protected EObject owner;
protected Collection selection;
}

```

Hierarchy: Object → AbstractCommand (Command) → CommandWrapper → CreateChildCommand (CommandActionDelegate)

CreateChildCommand.Helper

org.eclipse.emf.edit.command

CreateChildCommand.Helper is the helper interface to which some of the CreateChildCommand functionality (that is, matters of appearance) is delegated. It is typically implemented by an item provider of the owner object under which a new child is being created.

```

public static interface CreateChildCommand.Helper {
    // Public Instance Methods
    public abstract String getCreateChildDescription(Object owner, Object feature, Object child, Collection selection);
    public abstract Object getCreateChildImage(Object owner, Object feature, Object child, Collection selection);
    public abstract Collection getCreateChildResult(Object child);
    public abstract String getCreateChildText(Object owner, Object feature, Object child, Collection selection);
    public abstract String getCreateChildToolTipText(Object owner, Object feature, Object child, Collection selection);
}

```

CreateCopyCommand

org.eclipse.emf.edit.command

CreateCopyCommand is used by a CopyCommand to create an uninitialized object of the same type as its owner, which is later initialized using InitializeCopyCommand. After copying, the owner and its copy are added to the associated CopyCommand.Helper.

The create copy command is also used to control the depth and structure of the copy command's deep copy. The doGetChildrenToCopy() method is called by the CopyCommand at each node in the tree being copied, to determine which children to copy. This default implementation returns the contained objects, eContents(), of the owner.

```

public class CreateCopyCommand extends AbstractOverrideableCommand implements ChildrenToCopyProvider {
    // Public Constructors
    public CreateCopyCommand(EditingDomain domain, EObject owner, CopyCommand.Helper copyHelper);
    // Protected Constants
    protected static final String DESCRIPTION;
    protected static final String LABEL;
    // Public Class Methods
    public static Command create(EditingDomain domain, Object owner, CopyCommand.Helper copyHelper);
    // Property Accessor Methods (by property name)
    public CopyCommand.Helper getCopyHelper();
    public EObject getOwner();
    // Public Methods Overriding AbstractOverrideableCommand
    public void doExecute();
    public Collection doGetChildrenToCopy();
    public Collection doGetResult();
}

```

```

public void doRedo();
public void doUndo();
public String toString();
// Protected Instance Methods
protected boolean prepare();                                     Overrides:AbstractCommand constant
// Protected Instance Fields
protected EObject copy;
protected CopyCommand.Helper copyHelper;
protected EObject owner;
}

}

```

Hierarchy: Object → AbstractCommand(Command) → AbstractOverrideableCommand(OverrideableCommand(Command)) → CreateCopyCommand(ChildrenToCopyProvider)

CutToClipboardCommand

org.eclipse.emf.edit.command

CutToClipboardCommand wraps and works exactly like a RemoveCommand but sets the removed objects to the clipboard maintained by the EditingDomain. The doExecute() method executes an ordinary RemoveCommand (that is, its wrapped command) to perform the cut, and then calls setClipboard() on the editing domain with the remove command's result. The previous clipboard value is also saved, in oldClipboard, so that it can be restored if the command is later undone.

```

public class CutToClipboardCommand extends CommandWrapper {
// Public Constructors
public CutToClipboardCommand(EditingDomain domain, Command command);
// Protected Constants
protected static final String DESCRIPTION;
protected static final String LABEL;
// Public Class Methods
public static Command create(EditingDomain domain, Object value);
public static Command create(EditingDomain domain, Object owner, Object feature, Object ←
value);
public static Command create(EditingDomain domain, Collection collection);
public static Command create(EditingDomain domain, Object owner, Object feature, Collection ←
collection);
// Public Methods Overriding CommandWrapper
public void execute();
public void redo();
public String toString();
public void undo();
// Protected Instance Fields
protected EditingDomain domain;
protected Collection oldClipboard;
}

```

Hierarchy: Object → AbstractCommand(Command) → CommandWrapper → CutToClipboardCommand

DragAndDropCommand

org.eclipse.emf.edit.command

The DragAndDropCommand acts on an owner object onto which a collection of objects is being dragged. It ultimately delegates all of its behavior to other types of commands. It is created early during a drag-and-drop operation, and used to incrementally control the enablement feedback and then to perform the final operation.

The `prepare()` method is called, as usual, by the `canExecute()` method in its base class, `AbstractCommand`. Its implementation delegates to one of the operation-specific `prepare()` methods, which create appropriate underlying commands (`dragCommand` and `dropCommand`) to execute the required operation. For example, `prepareDropCopyInsert()` creates a `CopyCommand` for the `dragCommand` and an `AddCommand` for the `dropCommand`. The `prepareDropMoveInsert()` uses just a `MoveCommand`, if the move is local to the same parent, or a `RemoveCommand` and `AddCommand` otherwise. The other `prepare()` methods work similarly to create the most appropriate commands for the required operation.

The executability status for most commands is constant, as `prepare()` is only called once and its return value is cached by `canExecute()`. By contrast, `DragAndDropCommand`'s `prepare()` method is called repeatedly during the drag-and-drop operation, as the result of repeated calls to `DragAndDropFeedback.validate()`. Each call to `validate()` resets the command's fields, based on the latest position of the cursor, including setting `isPrepared` back to `false`, and then calls `prepare()` via `canExecute()` again. The `getFeedback()` and `getOperation()` methods are then used to update the UI selection feedback.

A `DragAndDropCommand` is usually created using its static `create()` method, which delegates to the `EditingDomain` for the actual command creation.

```
public class DragAndDropCommand extends AbstractCommand implements DragAndDropFeedback {
    // Public Constructors
    public DragAndDropCommand(EditingDomain domain, Object owner, float location, int ←
    operations, int operation, Collection collection);
    public DragAndDropCommand(EditingDomain domain, Object owner, float location, int ←
    operations, int operation, Collection collection, boolean optimize);
    // Protected Constants
    protected static final String DESCRIPTION;
    protected static final String LABEL;
    // Public Inner Classes
    public static class Detail;
    // Public Class Methods
    public static Command create(EditingDomain domain, Object owner, float location, int ←
    operations, int operation, Collection collection);
    // Property Accessor Methods (by property name)
    public Collection getAffectedObjects();
    Overrides:AbstractCommand
    public Collection getCollection();
    public int getFeedback();
    Implements:DragAndDropFeedback
    public float getLocation();
    public int getOperation();
    Implements:DragAndDropFeedback
    public int getOperations();
    public Object getOwner();
    public Collection getResult();
    Overrides:AbstractCommand
    // Public Methods Overriding AbstractCommand
    public void dispose();
    public void execute();
    public void redo();
    public String toString();
    public void undo();
    / Methods Implementing DragAndDropFeedback
    public boolean validate(Object owner, float location, int operations, int operation, ←
    Collection collection);
    // Protected Instance Methods
    protected Collection getChildren(Object object);
    protected Object getParent(Object object);
    protected boolean optimizedCanExecute();
    empty
    protected boolean prepare();
    Overrides:AbstractCommand
    protected boolean prepareDropCopyInsert(Object parent, Collection children, int index);
```

```

protected boolean prepareDropCopyOn();
protected boolean prepareDropInsert();
protected boolean prepareDropLinkInsert(Object parent, Collection children, int index);
empty
protected boolean prepareDropLinkOn();
protected boolean prepareDropMoveInsert(Object parent, Collection children, int index);
empty
protected boolean prepareDropMoveOn();
protected boolean prepareDropOn();
protected void reset();
// Protected Instance Fields
protected Collection collection;
protected EditingDomain domain;
protected Command dragCommand;
protected Command dropCommand;
protected int feedback;
protected boolean isDragCommandExecuted;
protected float location;
protected float lowerLocationBound;
protected int operation;
protected int operations;
protected boolean optimize;
protected Object optimizedDropCommandOwner;
protected Object owner;
protected float upperLocationBound;
}

```

Hierarchy: Object → AbstractCommand (Command) → DragAndDropCommand (DragAndDropFeedback)

DragAndDropCommand.Detail

org.eclipse.emf.edit.command

DragAndDropCommand.Detail is used to encode the drag and drop arguments into an object that will be passed to a DragAndDropCommand as the feature of a CommandParameter.

```

public static class DragAndDropCommand.Detail {
// Public Constructors
public Detail(float location, int operations, int operation);
// Public Instance Fields
public float location;
public int operation;
public int operations;
}

```

DragAndDropFeedback

org.eclipse.emf.edit.command

The DragAndDropFeedback interface is used to provide detailed drag-and-drop status for the DragAndDropCommand, which implements it. The validate() method is called repeatedly as the command progresses to update the drag-and-drop status.

To remain independent of the Eclipse UI framework, it provides its own synonyms for all of the important org.eclipse.swt.dnd.DND constants. The getFeedback() method returns one of the FEEDBACK_ constants, and getOperation() returns one of the DROP_ constants.

```

public interface DragAndDropFeedback {
// Public Constants
public static final int
DROP_COPY; =1
public static final int

```

```

DROP_LINK; =4
    public static final int
DROP_MOVE; =2
    public static final int
DROP_NONE; =0
    public static final int
FEEDBACK_INSERT_AFTER; =4
    public static final int
FEEDBACK_INSERT_BEFORE; =2
    public static final int
FEEDBACK_NONE; =0
    public static final int
FEEDBACK_SELECT; =1
// Property Accessor Methods (by property name)
    public abstract int getFeedback();
    public abstract int getOperation();
// Public Instance Methods
    public abstract boolean validate(Object owner, float location, int operations, int ↪
operation, Collection collection);
}

```

InitializeCopyCommand

org.eclipse.emf.edit.command

InitializeCopyCommand sets the values of an object copy based on those of the original (copied) object. Instances of this command are used by a CopyCommand to initialize each of the objects created using a CreateCopyCommand. It copies the attributes and references from the owner to its copy.

The doExecute() method calls the two helper methods copyAttributes() and copyReferences(), which reflectively copy all of the attributes and non-containment references (excluding external bidirectional ones) respectively. The associated CopyCommand.Helper is used to determine copied reference targets.

```

public class InitializeCopyCommand extends AbstractOverrideableCommand {
// Public Constructors
    public InitializeCopyCommand(EditingDomain domain, EObject owner, CopyCommand.Helper ↪
copyHelper);
// Protected Constants
    protected static final String DESCRIPTION;
    protected static final String LABEL;
// Public Class Methods
    public static Command create(EditingDomain domain, Object owner, CopyCommand.Helper ↪
copyHelper);
// Property Accessor Methods (by property name)
    public EObject getCopy();
    public CopyCommand.Helper getCopyHelper();
    public EObject getOwner();
// Public Methods Overriding AbstractOverrideableCommand
    public void doExecute();
    public Collection doGetAffectedObjects();
    public Collection doGetResult();
    public void doRedo();
    public void doUndo();
    public String toString();
// Protected Instance Methods
    protected void copyAttributes();
    protected void copyReferences();
    protected Collection getAttributesToCopy();
    protected Collection getReferencesToCopy();
    protected boolean prepare();
// Protected Instance Fields
}

```

empty
empty
Overrides:AbstractCommand

```

protected EObject copy;
protected CopyCommand.Helper copyHelper;
protected EObject owner;
}

```

Hierarchy: Object → AbstractCommand(Command) → AbstractOverrideableCommand(OverrideableCommand(Command)) → InitializeCopyCommand

MoveCommand

org.eclipse.emf.edit.command

A MoveCommand is used to move an object to a new position in a many-valued attribute or reference of an owner object, or to a new position in a specified list (for example, the contents list of a resource). The objects are moved to the specified index or to the end of the list if the index value is CommandParameter.NO_INDEX.

The canExecute() method, via prepare(), returns false if the object being moved is not in the list or if the index is out of range. The getAffectedObjects() and getResult() methods both return the moved object.

A MoveCommand is usually created using its static create() method, which delegates to the EditingDomain for the actual command creation. The MoveCommand.create() method's feature argument is often null and deduced by the editing domain.

```

public class MoveCommand extends AbstractOverrideableCommand {
// Public Constructors
    public MoveCommand(EditingDomain domain, EObject owner, EStructuralFeature feature, Object ←
value, int index);
    public MoveCommand(EditingDomain domain, EList list, Object value, int index);
// Protected Constants
    protected static final String DESCRIPTION;
    protected static final String DESCRIPTION_FOR_LIST;
    protected static final String LABEL;
// Public Class Methods
    public static Command create(EditingDomain domain, Object owner, Object feature, Object ←
value, int index);
// Property Accessor Methods (by property name)
    public EStructuralFeature getFeature();
    public int getIndex();
    public int getOldIndex();
    public EObject getOwner();
    public EList getOwnerList();
    public Object getValue();
// Public Methods Overriding AbstractOverrideableCommand
    public void doExecute();
    public Collection doGetAffectedObjects();
    public Collection doGetResult();
    public void doRedo();
    public void doUndo();
    public String toString();
// Protected Instance Methods
    protected boolean prepare();                                            Overrides:AbstractCommand
// Protected Instance Fields
    protected EStructuralFeature feature;
    protected int index;
    protected int oldIndex;
    protected EObject owner;
    protected EList ownerList;
    protected Object value;
}

```

Hierarchy: Object → AbstractCommand(Command) → AbstractOverrideableCommand(OverrideableCommand(Command)) → MoveCommand

OverrideableCommand

org.eclipse.emf.edit.command

The `OverrideableCommand` interface provides an orthogonal (to subclassing) dimension of overrideability for commands. Used as intended, an overrideable command should implement this interface and call `EditingDomain.createOverrideCommand()` in its constructor to set up the override command. All of its `Command` methods should then be delegated to the override command if `getOverride()` is not null, or to corresponding `do()` methods otherwise. The contract with the overriding command is that the overrideable command will implement all its function in the `do()` methods—for example, `execute()` is implemented in `doExecute()`—so that the overriding command can call back to the overrideable command's `do()` methods if it wants to extend rather than replace the original implementation.

`AbstractOverrideableCommand` provides a convenient base implementation of this interface. The overrideable command mechanism is implemented by most of the built-in EMF commands and it is used by the EMF mapping framework.¹ Clients should not typically use this mechanism for their own command specializations.

```
public interface OverrideableCommand extends Command {
    // Property Accessor Methods (by property name)
    public abstract Command getOverride();
    public abstract void setOverride(Command overrideCommand);
    // Public Instance Methods
    public abstract boolean doCanExecute();
    public abstract boolean doCanUndo();
    public abstract void doDispose();
    public abstract void doExecute();
    public abstract Collection doGetAffectedObjects();
    public abstract String doGetDescription();
    public abstract String doGetLabel();
    public abstract Collection doGetResult();
    public abstract void doRedo();
    public abstract void doUndo();
}
```

Hierarchy: (OverrideableCommand(Command))

PasteFromClipboardCommand

org.eclipse.emf.edit.command

`PasteFromClipboardCommand` delegates to and works exactly like an `AddCommand` but the object(s) to be added are first copied from the `EditingDomain` clipboard using a `CopyCommand`. The implementation uses a `StrictCompoundCommand` to compose the copy command with the add command, and a `CommandWrapper` to delay the construction of the add command until the copy command has executed and its result is available.

¹The mapping framework is a recent addition to EMF that supports the viewing and editing of mappings between any two EMF models. It uses the overrideable command mechanism to specialize the generic EMF editing commands with mapping-specific extensions. The mapping framework is beyond the scope of this book.

```

public class PasteFromClipboardCommand extends AbstractOverrideableCommand {
    // Public Constructors
    public PasteFromClipboardCommand(EditingDomain domain, Object owner, Object feature, int index);
    public PasteFromClipboardCommand(EditingDomain domain, Object owner, Object feature, int index, boolean optimize);
    // Protected Constants
    protected static final String DESCRIPTION;
    protected static final String LABEL;
    // Public Class Methods
    public static Command create(EditingDomain domain, Object owner, Object feature);
    public static Command create(EditingDomain domain, Object owner, Object feature, int index);
    // Property Accessor Methods (by property name)
    public Object getFeature();
    public int getIndex();
    public Object getOwner();
    // Public Methods Overriding AbstractOverrideableCommand
    public void doDispose();
    public void doExecute();
    public Collection doGetAffectedObjects();
    public Collection doGetResult();
    public void doRedo();
    public void doUndo();
    public String toString();
    // Protected Instance Methods
    protected boolean optimizedCanExecute();
    protected boolean prepare();                                empty
    // Overrides:AbstractCommand
    // Protected Instance Fields
    protected StrictCompoundCommand command;
    protected Object feature;
    protected int index;
    protected boolean optimize;
    protected Object owner;
}

```

Hierarchy: Object → AbstractCommand(Command) → AbstractOverrideableCommand(OverrideableCommand(Command)) → PasteFromClipboardCommand

RemoveCommand

org.eclipse.emf.edit.command

A RemoveCommand is used to remove one or more objects from a many-valued attribute or reference of an owner object, or from a specified list (for example, the contents list of a resource).

The canExecute() method, via prepare(), returns false if any of the objects being removed are not in the list. The getAffectedObjects() method returns the owner object after execute() and redo(), and the collection of previously removed objects after undo(). The getResult() method always returns the collection of removed objects.

Like all the commands in this package, a RemoveCommand is usually created using one of its static create() methods, which delegates to an EditingDomain for the actual command creation.

```

public class RemoveCommand extends AbstractOverrideableCommand {
    // Public Constructors
    public RemoveCommand(EditingDomain domain, EObject owner, EStructuralFeature feature, Object value);
    public RemoveCommand(EditingDomain domain, EObject owner, EStructuralFeature feature, Collection collection);
    public RemoveCommand(EditingDomain domain, EList list, Object value);
    public RemoveCommand(EditingDomain domain, EList list, Collection collection);
    // Protected Constants
    protected static final String DESCRIPTION;
    protected static final String DESCRIPTION_FOR_LIST;
    protected static final String LABEL;
}

```

```

// Public Class Methods
    public static Command create(EditingDomain domain, Object value);
    public static Command create(EditingDomain domain, Object owner, Object feature, Object ←
        value);
    public static Command create(EditingDomain domain, Collection collection);
    public static Command create(EditingDomain domain, Object owner, Object feature, Collection ←
        collection);
// Property Accessor Methods (by property name)
    public Collection getCollection();
    public EStructuralFeature getFeature();
    public int[] getIndices();
    public EObject getOwner();
    public EList getOwnerList();
// Public Methods Overriding AbstractOverrideableCommand
    public void doExecute();
    public Collection doGetAffectedObjects();
    public Collection doGetResult();
    public void doRedo();
    public void doUndo();
    public String toString();
// Protected Instance Methods
    protected boolean prepare();
// Protected Instance Fields
    protected Collection affectedObjects;
    protected Collection collection;
    protected EStructuralFeature feature;
    protected int[] indices;
    protected EObject owner;
    protected EList ownerList;
}

```

Overrides:AbstractCommand

Hierarchy: Object → AbstractCommand(Command) → AbstractOverrideableCommand(OverrideableCommand(Command)) → RemoveCommand

ReplaceCommand

org.eclipse.emf.edit.command

A ReplaceCommand is used to replace an object in a many-valued attribute or reference of an owner object, or in a specified list (for example, the contents list of a resource), with one or more other objects.

The `canExecute()` method, via `prepare()`, returns `false` if the value being replaced is not in the list or if any of the replacement objects are not of the correct type. The `getAffectedObjects()` method returns the collection of replacement objects after `execute()` and `redo()`, and the previously replaced object after `undo()`. The `getResult()` method always returns the collection of replacement objects.

Like all the commands in this package, a ReplaceCommand is usually created using one of its static `create()` methods, which delegates to an EditingDomain for the actual command creation.

```

public class ReplaceCommand extends AbstractOverrideableCommand {
// Public Constructors
    public ReplaceCommand(EditingDomain domain, EObject owner, EStructuralFeature feature, ←
        Object value, Object replacement);
    public ReplaceCommand(EditingDomain domain, EObject owner, EStructuralFeature feature, ←
        Object value, Collection collection);
    public ReplaceCommand(EditingDomain domain, EList list, Object value, Object replacement);
    public ReplaceCommand(EditingDomain domain, EList list, Object value, Collection ←
        collection);
// Protected Constants
    protected static final String DESCRIPTION;
    protected static final String LABEL;
// Public Class Methods
    public static Command create(EditingDomain domain, Object value, Collection collection);

```

```

    public static Command create(EditingDomain domain, Object owner, Object feature, Object ←
    value, Collection collection);
    // Property Accessor Methods (by property name)
    public Collection getCollection();
    public EStructuralFeature getFeature();
    public int getIndex();
    public EObject getOwner();
    public EList getOwnerList();
    public Object getValue();
    // Public Methods Overriding AbstractOverrideableCommand
    public void doExecute();
    public Collection doGetAffectedObjects();
    public Collection doGetResult();
    public void doRedo();
    public void doUndo();
    public String toString();
    // Protected Instance Methods
    protected boolean prepare();
    Overrides:AbstractCommand
    // Protected Instance Fields
    protected Collection affectedObjects;
    protected Collection collection;
    protected EStructuralFeature feature;
    protected int index;
    protected EObject owner;
    protected EList ownerList;
    protected Object value;
}

```

Hierarchy: Object → AbstractCommand(Command) → AbstractOverrideableCommand(OverrideableCommand(Command)) → ReplaceCommand

SetCommand

org.eclipse.emf.edit.command

A SetCommand is used to set the value of a single-valued attribute or reference of an owner object. The canExecute() method, via prepare(), returns false if the value is not of the correct type for the feature being set. The getAffectedObjects() and getResult() methods both return the owner object.

Like all the commands in this package, a SetCommand is usually created using its static create() method, which delegates to an EditingDomain for the actual command creation.

A SetCommand on a bidirectional reference with a multiplicity-many reverse, or with a multiplicity-1 reverse that is already set (on the value object), is not undoable; canUndo() returns false in these cases. The SetCommand's static create() method overcomes this problem: Instead of returning an instance of this class, it returns a CompoundCommand containing a RemoveCommand followed by an AddCommand for the other end of the association, which can be undone.

```

public class SetCommand extends AbstractOverrideableCommand {
    // Public Constructors
    public SetCommand(EditingDomain domain, EObject owner, EStructuralFeature feature, Object ←
    value);
    // Protected Constants
    protected static final String DESCRIPTION;
    protected static final String LABEL;
    protected static final EcorePackage ecorePackage;
    // Public Class Methods
    public static Command create(EditingDomain domain, Object owner, Object feature, Object ←
    value);
    // Property Accessor Methods (by property name)
    public EStructuralFeature getFeature();
    public Object getOldValue();
}

```

```

public EObject getOwner();
public Object getValue();
// Public Methods Overriding AbstractOverrideableCommand
public boolean doCanUndo();
public void doExecute();
public Collection doGetAffectedObjects();
public Collection doGetResult();
public void doRedo();
public void doUndo();
public String toString();
// Protected Instance Methods
protected boolean prepare();
// Protected Instance Fields
protected boolean canUndo;
protected EStructuralFeature feature;
protected Object oldValue;
protected EObject owner;
protected Object value;
}

Overrides:AbstractCommand

```

Hierarchy: Object → AbstractCommand(Command) → AbstractOverrideableCommand(OverrideableCommand(Command)) → SetCommand

The org.eclipse.emf.edit.domain Package

The `org.eclipse.emf.edit.domain` package includes the `EditingDomain` interface, which acts as the centralized control mechanism for managing a set of resources and the editing commands that modify them. It also provides the default adapter-delegating editing domain implementation class and other support classes for using it.

`AdapterFactoryEditingDomain`

`org.eclipse.emf.edit.domain`

`AdapterFactoryEditingDomain` is the default implementation of the `EditingDomain` interface used by the EMF.Edit framework. It wraps an `AdapterFactory`, which it uses to access item providers implementing the `IEditionDomainItemProvider` interface, to which it delegates much of its implementation. It maintains a command stack, resource set, and clipboard in the data fields `commandStack`, `resourceSet`, and `clipboard`, respectively.

With the exception of the clipboard commands, which it creates itself, the `createCommand()` method delegates its implementation to the owner object specified in the `commandParameter` argument. For commands being created without an owner specified, it deduces one first. For a `RemoveCommand` involving a collection of objects, not all of which are children of the same parent, a `CompoundCommand` is created containing several remove commands, one per unique parent, each of which is created by delegation to the parent's (owner's) item provider.

The two static `getEditingDomainFor()` methods provide global access to the domain to use when editing a given object. If the object is itself an `IEditionDomainProvider`, then its corresponding domain is returned. Otherwise, if it is an `EObject`, the editing domain is accessed from its resource set, if it implements the `IEditionDomainProvider` interface. The third static method, `getEditingDomainItemProviderFor()`, simply calls `getEditingDomainFor()` and then, using the returned domain's `adapterFactory`, adapts the specified object with the `IEditionDomainProvider` adapter type.

An adapter factory editing domain can optionally be passed a resource set when constructed, which should implement the `IErasingDomainProvider` interface. If no resource set is specified, a new instance of the inner class `AdapterFactoryEditingDomainResourceSet` is used.

```
public class AdapterFactoryEditingDomain implements EditingDomain {
    // Public Constructors
    public AdapterFactoryEditingDomain(AdapterFactory adapterFactory, CommandStack commandStack);
    public AdapterFactoryEditingDomain(AdapterFactory adapterFactory, CommandStack commandStack, ResourceSet resourceSet);
    // Public Inner Classes
    public static class DomainTreeIterator;
    // Protected Inner Classes
    protected class AdapterFactoryEditingDomainResourceSet;
    // Public Class Methods
    public static EditingDomain getEditingDomainFor(EObject object);
    public static EditingDomain getEditingDomainFor(Object object);
    public static IErasingDomainItemProvider getEditingDomainItemProviderFor(Object object);
    // Property Accessor Methods (by property name)
    public AdapterFactory getAdapterFactory();
    public void setAdapterFactory(AdapterFactory adapterFactory);
    public Collection getClipboard();                                Implements:EditingDomain
    public void setClipboard(Collection clipboard);                  Implements:EditingDomain
    public CommandStack getCommandStack();                          Implements:EditingDomain
    public boolean getOptimizeCopy();                             Implements:EditingDomain
    public void setOptimizeCopy(boolean optimizeCopy);           Implements:EditingDomain
    public ResourceSet getResourceSet();                           Implements:EditingDomain
    // Methods Implementing EditingDomain
    public Command createCommand(Class commandClass, CommandParameter commandParameter);
    public Command createOverrideCommand(OverrideableCommand command);          constant
    public Resource createResource(String fileNameURI);
    public Collection getChildren(Object object);
    public Collection getChildDescriptors(Object object, Object sibling);
    public Object getParent(Object object);
    public Object getRoot(Object object);
    public List getTreePath(Object object);
    public Resource loadResource(String fileNameURI);
    public TreeIterator treeIterator(Object object);
    // Protected Instance Fields
    protected AdapterFactory adapterFactory;
    protected Collection clipboard;
    protected CommandStack commandStack;
    protected boolean optimizeCopy;
    protected ResourceSet resourceSet;
}
```

Hierarchy: Object → AdapterFactoryEditingDomain (EditingDomain)

AdapterFactoryEditingDomain.AdapterFactoryEditingDomainResourceSet

org.eclipse.emf.edit.domain notifier

`AdapterFactoryEditingDomain.AdapterFactoryEditingDomainResourceSet` is a resource set that also implements `IErasingDomainProvider` to provide access to its associated editing domain.

```
protected class AdapterFactoryEditingDomain.AdapterFactoryEditingDomainResourceSet extends ResourceSetImpl implements IErasingDomainProvider {
    // Public Constructors
    public AdapterFactoryEditingDomainResourceSet();
    // Property Accessor Methods (by property name)
    public EditingDomain getEditingDomain();                         Implements:IErasingDomainProvider
}
```

Hierarchy: Object → NotifierImpl(Notifier) → ResourceSetImpl(ResourceSet(Notifier)) → AdapterFactoryEditingDomain.AdapterFactoryEditingDomainResourceSet(IEditingDomainProvider)

AdapterFactoryEditingDomain.DomainTreeIterator

org.eclipse.emf.edit.domain collection

AdapterFactoryEditingDomain.DomainTreeIterator implements a tree iterator using the getChildren() method of an EditingDomain to access the children.

```
public static class AdapterFactoryEditingDomain.DomainTreeIterator extends ↪
AbstractTreeIterator {
// Public Constructors
    public DomainTreeIterator(EditingDomain domain, Object object);
    public DomainTreeIterator(EditingDomain domain, Object object, boolean includeRoot);
// Protected Instance Methods
    protected Iterator getChildren(Object o);
    Overrides:AbstractTreeIterator
// Protected Instance Fields
    protected EditingDomain domain;
}
```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → AbstractTreeIterator(TreeIterator(Iterator)) → AdapterFactoryEditingDomain.DomainTreeIterator

EditingDomain

org.eclipse.emf.edit.domain

An EditingDomain manages a set of concurrently edited EMF resources and the commands that modify them. Commands that modify the model are typically created through the domain and are executed using its command stack.

The domain imposes a hierarchical structure on the models via the results of the getChildren() and getParent() methods. This is used for implementing commands such as RemoveCommand, which often needs to deduce the parent from which to remove a particular object, and CopyCommand, which often needs to deduce all the children to copy recursively. The getRoot() method walks the parent chain, via getParent(), until null is returned. The getTreePath() method returns a list of objects on the path from the root object to the specified object in the tree.

Command creation is done by calling createCommand() on the domain. Command implementation classes typically provide static create() convenience methods that construct a CommandParameter and then call createCommand() indirectly for the client. The createOverrideCommand() method is called by overrideable commands to create their override, if there is one. Ordinary editing domain implementations should typically return null (that is, no commands are overridden), because this mechanism is reserved for framework use. Execution of commands in the domain uses the CommandStack returned by getCommandStack().

The getNewChildDescriptors() method returns a collection of objects describing the different children that can be added, using CreateChildCommand, under the specified object. If a sibling is specified (non-null), the children should be as close to immediately following that sibling as possible. The createResource() and loadResource() methods are convenience methods for

creating and loading a resource, respectively, using the domain's resource set. The `getResouceSet()` method returns the resource set itself. The `getClipboard()` and `setClipboard()` methods are used to get and set the contents of the domain's clipboard, respectively. The `getOptimizeCopy()` method is called by the copy command to determine if certain implementation optimizations are safe in this domain. An implementation typically returns `true`.

```
public interface EditingDomain {
    // Property Accessor Methods (by property name)
    public abstract Collection getClipboard();
    public abstract void setClipboard(Collection clipboard);
    public abstract CommandStack getCommandStack();
    public abstract boolean getOptimizeCopy();
    public abstract ResourceSet getResourceSet();
    // Public Instance Methods
    public abstract Command createCommand(Class commandClass, CommandParameter commandParameter);
    public abstract Command createOverrideCommand(OverrideableCommand command);
    public abstract Resource createResource(String fileNameURI);
    public abstract Collection getChildren(Object object);
    public abstract Collection getNewChildDescriptors(Object object, Object sibling);
    public abstract Object getParent(Object object);
    public abstract Object getRoot(Object object);
    public abstract List getTreePath(Object object);
    public abstract Resource loadResource(String fileNameURI);
    public abstract TreeIterator treeIterator(Object object);
}
```

IEditingDomainProvider

org.eclipse.emf.edit.domain

`IEditingDomainProvider` is used to access an `EditingDomain`, usually from a resource set.

```
public interface IEditingDomainProvider {
    // Property Accessor Methods (by property name)
    public abstract EditingDomain getEditingDomain();
}
```

The org.eclipse.emf.edit.provider Package

The `org.eclipse.emf.edit.provider` package includes all the interfaces implemented by item providers. It also includes the highly functional convenience base class used for the implementation of item providers (typically generated) that are also EMF adapters. Another convenience class is provided for implementing item providers for nonmodeled objects. Property sheet support classes, and a number of interfaces and classes for composing adapter factories and working with item providers, are also included.

AdapterFactoryItemDelegator

org.eclipse.emf.edit.provider

`AdapterFactoryItemDelegator` is a convenience class for calling basic item provider interface methods on an object. Instead of adapting the object, downcasting to the required provider interface, and then calling the method, a client can simply call the required method directly on a single instance of this class. It, in turn, adapts the object and delegates the call to the appropriate item provider.

This class is conceptually similar to the `AdapterFactoryContentProvider` and `AdapterFactoryLabelProvider` classes in the `org.eclipse.emf.edit.ui` plug-in, only it is independent of the Eclipse UI framework.

```

public class AdapterFactoryItemDelegator implements IEditingDomainItemProvider, ←
IItemLabelProvider, IItemPropertySource, IStructuredItemContentProvider, ←
ITableItemLabelProvider, ITreeItemContentProvider {
// Public Constructors
    public AdapterFactoryItemDelegator(AdapterFactory adapterFactory);
// Property Accessor Methods (by property name)
    public AdapterFactory getAdapterFactory();
// Methods Implementing IEditingDomainItemProvider
    public Command createCommand(Object object, EditingDomain editingDomain, Class ←
commandClass, CommandParameter commandParameter);
    public Collection getChildren(Object object);
    public Collection getNewChildDescriptors(Object object, EditingDomain editingDomain, Object ←
sibling);
    public Object getParent(Object object);
// Methods Implementing IItemLabelProvider
    public Object getImage(Object object);
    public String getText(Object object);
// Methods Implementing IItemPropertySource
    public Object getEditableValue(Object object);
    public IPropertyDescriptor getPropertyDescriptor(Object object, Object propertyId);
    public List getPropertyDescriptors(Object object);
// Methods Implementing IStructuredItemContentProvider
    public Collection getElements(Object object);
// Methods Implementing ITableItemLabelProvider
    public Object getColumnImage(Object object, int columnIndex);
    public String getColumnText(Object object, int columnIndex);
// Methods Implementing ITreeItemContentProvider
    public boolean hasChildren(Object object);
// Protected Instance Fields
    protected AdapterFactory adapterFactory;
}

```

Hierarchy: Object → AdapterFactoryItemDelegator(IEditingDomainItemProvider, IItemLabelProvider, IItemPropertySource, IStructuredItemContentProvider, ITableItemLabelProvider, ITreeItemContentProvider (IStructuredItemContentProvider))

AdapterFactoryTreeIterator

org.eclipse.emf.edit.provider collection

AdapterFactoryTreeIterator implements a tree iterator using an AdapterFactory to adapt the objects and access their children. The objects are adapted with the ITreeItemContentProvider interface.

```

public class AdapterFactoryTreeIterator extends AbstractTreeIterator {
// Public Constructors
    public AdapterFactoryTreeIterator(AdapterFactory adapterFactory, Object object);
    public AdapterFactoryTreeIterator(AdapterFactory adapterFactory, Object object, boolean ←
includeRoot);
// Protected Instance Methods
    protected Iterator getChildren(Object o);
Overrides:AbstractTreeIterator
// Protected Instance Fields
    protected AdapterFactory adapterFactory;
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → BasicEList(EList(List), Cloneable, Serializable) → AbstractTreeIterator(TreeIterator(Iterator)) → AdapterFactoryTreeIterator

ChangeNotifier

org.eclipse.emf.edit.provider collection

Class `ChangeNotifier` is a simple implementation of the `IChangeNotifier` interface, which maintains its listeners using the `ArrayList` that it extends.

```
public class ChangeNotifier extends ArrayList implements IChangeNotifier {
    // Public Constructors
    public ChangeNotifier();
    // Methods Implementing IChangeNotifier
    public void addListener(INotifyChangedListener notifyChangedListener);
    public void fireNotifyChanged(Notification notification);
    public void removeListener(INotifyChangedListener notifyChangedListener);
}
```

empty

Hierarchy: Object → AbstractCollection(Collection) → AbstractList(List(Collection)) → ArrayList(List, Cloneable, Serializable) → ChangeNotifier(IChangeNotifier)

ComposeableAdapterFactory

org.eclipse.emf.edit.provider

The `ComposeableAdapterFactory` interface allows an adapter factory to be composed into a `ComposedAdapterFactory` that serves as the adapter factory for the union of the model objects from different packages. Item provider adapter factories typically implement this interface.

The `setParentAdapterFactory()` and `getRootAdapterFactory()` methods are used to set a parent adapter factory and to navigate to the root of the composition hierarchy, respectively.

```
public interface ComposeableAdapterFactory extends AdapterFactory {
    // Property Accessor Methods (by property name)
    public abstract void setParentAdapterFactory(ComposedAdapterFactory parentAdapterFactory);
    public abstract ComposeableAdapterFactory getRootAdapterFactory();
}
```

Hierarchy: (ComposeableAdapterFactory(AdapterFactory))

ComposedAdapterFactory

org.eclipse.emf.edit.provider

Class `ComposedAdapterFactory` is used to compose several adapter factories for different models into a single factory serving the union of the model objects. The nested adapter factories need not implement `ComposeableAdapterFactory`, but typically do. `ComposedAdapterFactory` is itself composable, and thereby supports an arbitrarily deep composition structure.

The `adapt()` method delegates to the first nested adapter factory for which `isFactoryForType()` returns true for both the specified type and for the package of the object being adapted. It uses the `getFactoryForTypes()` method to locate the factory. The `isFactoryForType()` method returns true if any of the nested factories returns true for the same method.

```
public class ComposedAdapterFactory implements AdapterFactory, ComposeableAdapterFactory, ←
IChangeNotifier, IDisposable {
    // Public Constructors
    public ComposedAdapterFactory(AdapterFactory adapterFactory);
```

```

public ComposedAdapterFactory(AdapterFactory[] adapterFactories);
public ComposedAdapterFactory(Collection adapterFactories);
// Property Accessor Methods (by property name)
public void setParentAdapterFactory(ComposedAdapterFactory parentAdapterFactory);
public ComposeableAdapterFactory getRootAdapterFactory();
                                         Implements:ComposeableAdapterFactory
// Public Instance Methods
public void addAdapterFactory(AdapterFactory adapterFactory);
public AdapterFactory getFactoryForType(Object type);
public AdapterFactory getFactoryForTypes(Collection types);
public void removeAdapterFactory(AdapterFactory adapterFactory);
// Methods Implementing AdapterFactory
public Object adapt(Object target, Object type);
public Adapter adapt(Notifier target, Object type);
public void adaptAllNew(Notifier target);
public Adapter adaptNew(Notifier target, Object type);
public boolean isFactoryForType(Object type);
// Methods Implementing IChangeNotifier
public void addListener(INotifyChangedListener notifyChangedListener);
public void fireNotifyChanged(Notification notification);
public void removeListener(INotifyChangedListener notifyChangedListener);
// Methods Implementing IDisposable
public void dispose();
// Protected Instance Methods
protected Adapter adapt(Notifier target, Object type, Collection failedPackages, Class ←
javaClass);
// Protected Instance Fields
protected Collection adapterFactories;
protected ChangeNotifier changeNotifier;
protected ComposedAdapterFactory parentAdapterFactory;
}

```

Hierarchy: Object → ComposedAdapterFactory(AdapterFactory, ComposeableAdapterFactory(AdapterFactory), IChangeNotifier, IDisposable)

ComposedImage

org.eclipse.emf.edit.provider

Class ComposedImage is used to compose multiple images into a single one. By default, the images are overlayed at position (0, 0) and the composed size is sufficient to contain the maximum of all the image heights and widths.

```

public class ComposedImage {
// Public Constructors
public ComposedImage(Collection images);
// Public Inner Classes
public static class Point;
public static class Size;
// Property Accessor Methods (by property name)
public List getImages();
// Public Instance Methods
public List getDrawPoints(ComposedImage.Size size);
public ComposedImage.Size getSize(Collection imageSizes);
// Public Methods Overriding Object
public boolean equals(Object that);
public int hashCode();
// Protected Instance Fields
protected List imageSizes;
protected List images;
}

```

ComposedImage.Point

org.eclipse.emf.edit.provider

ComposedImage.Point is used by a ComposedImage to represent the position of an image.

```
public static class ComposedImage.Point {
    // Public Constructors
    public Point();
    // Public Instance Fields
    public int x;
    public int y;
}
```

empty

ComposedImage.Size

org.eclipse.emf.edit.provider

ComposedImage.Size is used by a ComposedImage to represent the size of an image.

```
public static class ComposedImage.Size {
    // Public Constructors
    public Size();
    // Public Instance Fields
    public int height;
    public int width;
}
```

empty

DecoratorAdapterFactory

org.eclipse.emf.edit.provider

The DecoratorAdapterFactory abstract class provides support for creating IItemProviderDecorators for the adapters created by another adapter factory, represented by the decoratedAdapterFactory field. The adapt() method delegates to adapt() on its decoratedAdapterFactory, and then returns a corresponding implementation of IItemProviderDecorator, creating one if necessary. Subclasses override the createItemProviderDecorator() abstract method to return a decorator of the appropriate type, usually a subclass of ItemProviderDecorator.

As with most (usually generated) item provider adapter factories, DecoratorAdapterFactory is composeable, a change notifier, and disposable.

```
public abstract class DecoratorAdapterFactory implements AdapterFactory, ↵
    ComposeableAdapterFactory, IChangeNotifier, IDisposable {
    // Public Constructors
    public DecoratorAdapterFactory(AdapterFactory decoratedAdapterFactory);
    // Property Accessor Methods (by property name)
    public AdapterFactory getDecoratedAdapterFactory();
    public void setDecoratedAdapterFactory(AdapterFactory decoratedAdapterFactory);
    public void setParentAdapterFactory(ComposeableAdapterFactory parentAdapterFactory);
    // Methods Implementing AdapterFactory
    public Object adapt(Object target, Object type);
    public Adapter adapt(Notifier target, Object type);
    public void adaptAllNew(Notifier target);
    public Adapter adaptNew(Notifier target, Object type);
    public boolean isFactoryForType(Object type);
    // Methods Implementing IChangeNotifier
    public void addListener(INotifyChangedListener notifyChangedListener);
```

Implements:ComposeableAdapterFactory
Implements:ComposeableAdapterFactory

```

public void fireNotifyChanged(Notification notification);
public void removeListener(IChangeListener notifyChangedListener);
// Methods Implementing IDisposable
public void dispose();
// Protected Instance Methods
protected abstract IItemProviderDecorator createItemProviderDecorator(Object target, Object < Type>);
// Protected Instance Fields
protected ChangeNotifier changeNotifier;
protected AdapterFactory decoratedAdapterFactory;
protected HashMap itemProviderDecorators;
protected ComposedAdapterFactory parentAdapterFactory;
}

```

Hierarchy: Object → DecoratorAdapterFactory(AdapterFactory, ComposeableAdapterFactory(AdapterFactory), IChangeNotifier, IDisposable)

Disposable

org.eclipse.emf.edit.provider.collection

Class Disposable implements a set of disposable objects that can, in turn, be disposed. Objects added to the set must implement the IDisposable interface.

```

public class Disposable extends HashSet implements IDisposable {
// Public Constructors
    public Disposable();
    public Disposable(Collection disposables);                                         empty
// Public Methods Overriding HashSet
    public boolean add(Object object);
// Public Methods Overriding AbstractCollection
    public boolean addAll(Collection collection);
// Methods Implementing IDisposable
    public void dispose();
}

```

Hierarchy: Object → AbstractCollection(Collection) → AbstractSet(Set(Collection)) → HashSet(Set, Cloneable, Serializable) → Disposable(IDisposable)

IChangeNotifier

org.eclipse.emf.edit.provider

Interface IChangeNotifier provides the EMF.Edit notification mechanism that is most commonly used to centralize and pass object-level notifications from item providers on to views. As described in Chapter 3, content providers of viewers are added as listeners to item provider adapter factories, which implement this interface.

```

public interface IChangeNotifier {
// Public Instance Methods
    public abstract void addListener(IChangeListener notifyChangedListener);
    public abstract void fireNotifyChanged(Notification notification);
    public abstract void removeListener(IChangeListener notifyChangedListener);
}

```

IDisposable**org.eclipse.emf.edit.provider**

`IDisposable` is implemented by objects that need to be disposed (that is, require some sort of cleanup) after they are no longer needed.

```
public interface IDisposable {
    // Public Instance Methods
    public abstract void dispose();
}
```

IEditingDomainItemProvider**org.eclipse.emf.edit.provider**

Interface `IEditingDomainItemProvider` is implemented by item providers to support editing of an object. It includes the subset of the `EditingDomain` methods that are delegated to item providers, essentially those supporting command creation and initialization.

```
public interface IEditingDomainItemProvider {
    // Public Instance Methods
    public abstract Command createCommand(Object object, EditingDomain editingDomain, Class <?
    commandClass, CommandParameter commandParameter);
    public abstract Collection getChildren(Object object);
    public abstract Collection getNewChildDescriptors(Object object, EditingDomain <?
    editingDomain, Object sibling);
    public abstract Object getParent(Object object);
}
```

IItemLabelProvider**org.eclipse.emf.edit.provider**

Interface `IItemLabelProvider` is implemented by item providers to provide label text and an icon for an object. It is the EMF.Edit “item” equivalent of the Eclipse UI framework’s `ILabelProvider` interface, as described in Chapter 3.

```
public interface IItemLabelProvider {
    // Public Instance Methods
    public abstract Object getImage(Object object);
    public abstract String getText(Object object);
}
```

IItemPropertyDescriptor**org.eclipse.emf.edit.provider**

The `IItemPropertyDescriptor` interface is used to represent a property of an object. It is the EMF.Edit “item” equivalent of the Eclipse UI framework’s `IPropertyDescriptor` interface, as described in Chapter 3. It also provides additional methods that allow some of an `IPropertySource` implementation to be delegated to the descriptor, so that the descriptor implementation class can completely encapsulate the work associated with supporting a particular property sheet property.

```
public interface IItemPropertyDescriptor {
    // Public Instance Methods
    public abstract boolean canSetProperty(Object object);
    public abstract String getCategory(Object object);
```

```

public abstract Collection getChoiceOfValues(Object object);
public abstract String getDescription(Object object);
public abstract String getDisplayName(Object object);
public abstract Object getFeature(Object object);
public abstract String[] getFilterFlags(Object object);
public abstract Object getHelpContextIds(Object object);
public abstract String getId(Object object);
public abstract IItemLabelProvider getLabelProvider(Object object);
public abstract Object getPropertyValue(Object object);
public abstract boolean isCompatibleWith(Object object, Object anotherObject, ↵
IItemPropertyDescriptor anotherPropertyDescriptor);
public abstract boolean isPropertySet(Object object);
public abstract void resetPropertyValue(Object object);
public abstract void setPropertyValue(Object object, Object value);
}

```

IItemPropertySource

`org.eclipse.emf.edit.provider`

An `IItemPropertySource` is used to populate property sheet items. It is the EMF.Edit “item” equivalent of the Eclipse UI framework’s `IPropertySource` interface, as described in Chapter 3. It includes only a subset of the `IPropertySource` methods; the rest are supported by `IItemPropertyDescriptor`.

```

public interface IItemPropertySource {
// Public Instance Methods
    public abstract Object getEditableValue(Object object);
    public abstract IItemPropertyDescriptor getPropertyDescriptor(Object object, Object ↵
propertyID);
    public abstract List getPropertyDescriptors(Object object);
}

```

IItemProviderDecorator

`org.eclipse.emf.edit.provider`

The `IItemProviderDecorator` interface is implemented by item providers that decorate another item provider. Implementations of it typically extend from the convenience base class `ItemProviderDecorator`.

```

public interface IItemProviderDecorator {
// Property Accessor Methods (by property name)
    public abstract IChangeNotifier getDecoratedItemProvider();
    public abstract void setDecoratedItemProvider(IChangeNotifier decoratedItemProvider);
}

```

INotifyChangedListener

`org.eclipse.emf.edit.provider`

Interface `INotifyChangedListener` is implemented by listeners of an `IChangeNotifier`.

```

public interface INotifyChangedListener {
// Public Instance Methods
    public abstract void notifyChanged(Notification notification);
}

```

IStructuredItemContentProvider**org.eclipse.emf.edit.provider**

Interface `IStructuredItemContentProvider` is implemented by item providers to identify the objects with which to populate the top-level items in a tree viewer, the items of a list viewer, or the rows of a table viewer. It is the EMF.Edit “item” equivalent of the Eclipse UI framework’s `IstructuredContentProvider` interface, as described in Chapter 3.

```
public interface IStructuredItemContentProvider {  
    // Public Instance Methods  
    public abstract Collection getElements(Object object);  
}
```

ITableItemLabelProvider**org.eclipse.emf.edit.provider**

The `ITableItemLabelProvider` interface is used to provide labels for items in a table viewer. Its interface is similar to `IIItemLabelProvider`, but it passes an additional column index argument.

```
public interface ITableItemLabelProvider {  
    // Public Instance Methods  
    public abstract Object getColumnImage(Object object, int columnIndex);  
    public abstract String getColumnText(Object object, int columnIndex);  
}
```

ItemPropertyDescriptor**org.eclipse.emf.edit.provider**

Class `ItemPropertyDescriptor` provides a default implementation of the `IIItemPropertyDescriptor` interface. It maintains the data needed to map a feature of an `EObject` to a property sheet property. A property descriptor can be read-only, indicated by `isSettable` equal to `false`.

The `getPropertyValue()` and `isPropertySet()` methods use the reflective `EObject` API to generically access the property’s feature. The value returned by `getPropertyValue()` is wrapped in an `ItemPropertyDescriptor.PropertyValueWrapper`, returned by the `createPropertyValueWrapper()` method. The `setPropertyValue()` method uses the `getEditingDomain()` method to access the object’s domain and then sets the property using a `SetCommand`, thereby allowing property changes to be undone. The `getEditingDomain()` method calls the `AdapterFactoryEditingDomain.getEditingDomainFor()` static method for the object whose feature is being set.

The `getComboBoxObjects()` method is used by `getChoiceOfValues()` to populate a list of choices for the property’s value. If the property’s feature is a reference, it calls the static `getReachableObjectsOfType()` method, which works in conjunction with `collectReachableObjectsOfType()`, to exhaustively search the object’s resource set for all the potential targets of suitable type. Subclasses often override this default `getComboBoxObjects()` implementation to do this more efficiently and/or specifically.

The `getLabelProvider()` method returns the descriptor's `itemDelegator`, an instance of class `ItemPropertyDescriptor.ItemDelegator`, which is used to render the value of the property, and any objects returned by `getComboBoxObjects()`. If a static image is provided to the descriptor, the item delegator's `getImage()` method returns it. For properties whose type is a basic data type, one of the five constant fields (`BOOLEAN_VALUE_IMAGE`, `GENERIC_VALUE_CONSTANT`, and so on) is typically used for this purpose.

```

public class ItemPropertyDescriptor implements IItemPropertyDescriptor {
    // Public Constructors
    public ItemPropertyDescriptor(AdapterFactory adapterFactory, String displayName, String description, EStructuralFeature feature);
    public ItemPropertyDescriptor(AdapterFactory adapterFactory, String displayName, String description, EStructuralFeature feature, boolean isSettable);
    public ItemPropertyDescriptor(AdapterFactory adapterFactory, String displayName, String description, EStructuralFeature feature, boolean isSettable, String category);
    public ItemPropertyDescriptor(AdapterFactory adapterFactory, String displayName, String description, EReference[] parentReferences);
    public ItemPropertyDescriptor(AdapterFactory adapterFactory, String displayName, String description, EReference[] parentReferences, boolean isSettable);
    public ItemPropertyDescriptor(AdapterFactory adapterFactory, String displayName, String description, EStructuralFeature feature, boolean isSettable, Object staticImage);
    public ItemPropertyDescriptor(AdapterFactory adapterFactory, String displayName, String description, EStructuralFeature feature, boolean isSettable, Object staticImage, String category);
    // Public Constants
    public static final Object BOOLEAN_VALUE_IMAGE;
    public static final Object GENERIC_VALUE_IMAGE;
    public static final Object INTEGRAL_VALUE_IMAGE;
    public static final Object REAL_VALUE_IMAGE;
    public static final Object TEXT_VALUE_IMAGE;
    // Protected Constants
    protected static final EcorePackage ecorePackage;
    // Public Inner Classes
    public static class PropertyValueWrapper;
    // Protected Inner Classes
    protected class ItemDelegator;
    // Public Class Methods
    public static void collectReachableObjectsOfType(Collection visited, Collection result, EObject object, EClassifier type);
    public static Object getDefaultValue(EClassifier eType);
    public static Collection getReachableObjectsOfType(EObject object, EClassifier type);
    // Public Instance Methods
    public EditingDomain getEditingDomain(Object object);
    // Methods Implementing IItemPropertyDescriptor
    public boolean canSetProperty(Object object);
    public String getCategory(Object object);
    public Collection getChoiceOfValues(Object object);
    public String getDescription(Object object);
    public String getDisplayName(Object object);
    public Object getFeature(Object object);
    public String[] getFilterFlags(Object object);
    public Object getHelpContextIds(Object object);
    public String getId(Object object);
    public IItemLabelProvider getLabelProvider(Object object);
    public Object getPropertyValue(Object object);
    public boolean isCompatibleWith(Object object, Object anotherObject, IItemPropertyDescriptor anotherItemPropertyDescriptor);
    // Protected Instance Methods
    public boolean isPropertySet(Object object);
    public void resetPropertyValue(Object object);
    public void setPropertyValue(Object object, Object value);
    // Protected Instance Fields
    protected AdapterFactory adapterFactory;
    protected String category;
    protected String description;
}

```

```

protected String displayName;
protected EStructuralFeature feature;
protected String[] filterFlags;
protected boolean isSettable;
protected AdapterFactoryItemDelegator itemDelegator;
protected EReference[] parentReferences;
protected Object staticImage;
}

```

Hierarchy: Object → ItemPropertyDescriptor (IItemPropertyDescriptor)

ItemPropertyDescriptor.ItemDelegator

org.eclipse.emf.edit.provider

`ItemPropertyDescriptor.ItemDelegator` is the simple subclass of `AdapterFactoryItemDelegator` used by an item property descriptor to render the value of its property. The `getImage()` method returns a static image associated with the descriptor, if it has one. Otherwise, it delegates to the object's `IItemLabelProvider` via `super.getImage()`. The `getText()` method overrides the base implementation if the specified object is an attribute. In this case, it calls the attribute's data type `convertToString()` method, instead of delegating to the `IItemLabelProvider`.

```

protected class ItemPropertyDescriptor.ItemDelegator extends AdapterFactoryItemDelegator {
    // Public Constructors
    public ItemDelegator(AdapterFactory adapterFactory);
    // Public Methods Overriding AdapterFactoryItemDelegator
    public Object getImage(Object object);
    public String getText(Object object);
}

```

Hierarchy: Object → AdapterFactoryItemDelegator (IErasingDomainItemProvider, IItemLabelProvider, IItemPropertyDescriptor, IStructuredItemContentProvider, ITableItemLabelProvider, ITreeItemContentProvider (IStructuredItemContentProvider)) → ItemPropertyDescriptor.ItemDelegator

ItemPropertyDescriptor.PropertyValueWrapper

org.eclipse.emf.edit.provider

This class is used by class `ItemPropertyDescriptor` to wrap the value returned from the `get PropertyValue()` method. Its primary purpose is to produce nested property descriptors for the value, if it is compound. Its methods delegate to an `AdapterFactoryItemDelegator`, which is created using the adapter factory passed to its constructor. The `nestedPropertySource` is null for simple property values that have no nested properties.

```

public static class ItemPropertyDescriptor.PropertyValueWrapper implements IItemLabelProvider, ↵
    IItemPropertyDescriptor {
    // Public Constructors
    public PropertyValueWrapper(AdapterFactory adapterFactory, Object object, Object ↵
        propertyValue, Object nestedPropertySource);
    // Methods Implementing IItemLabelProvider
    public Object getImage(Object thisObject);
    public String getText(Object thisObject);
    // Methods Implementing IItemPropertyDescriptor
    public Object getEditableValue(Object thisObject);
    public IItemPropertyDescriptor getPropertyDescriptor(Object thisObject, Object propertyId);
    public List getPropertyDescriptors(Object thisObject);
    // Protected Instance Methods
    protected IItemPropertyDescriptor createPropertyDescriptorDecorator(Object object, ↵

```

```

IItemPropertyDescriptor itemPropertyDescriptor);
// Protected Instance Fields
protected AdapterFactoryItemDelegator itemDelegator;
protected Object nestedPropertySource;
protected Object object;
protected Object propertyValue;
}

```

Hierarchy: Object → ItemPropertyDescriptor.PropertyValueWrapper(IItemLabelProvider, IItemPropertySource)

ItemPropertyDescriptorDecorator

org.eclipse.emf.edit.provider

ItemPropertyDescriptorDecorator wraps another property descriptor and delegates all of its methods to it. It's constructed with two arguments, the itemPropertyDescriptor to which it delegates, and an object that it passes to the delegate, instead of the argument (thisObject), which is passed to it.

This class is commonly used to wrap property descriptors of an object to make them appear to belong to (that is, be displayed in the property sheet of) a different object.

```

public class ItemPropertyDescriptorDecorator implements IItemPropertyDescriptor {
// Public Constructors
    public ItemPropertyDescriptorDecorator(Object object, IItemPropertyDescriptor itemPropertyDescriptor);
// Methods Implementing IItemPropertyDescriptor
    public boolean canSetProperty(Object thisObject);
    public String getCategory(Object thisObject);
    public Collection getChoiceOfValues(Object thisObject);
    public String getDescription(Object thisObject);
    public String getDisplayName(Object thisObject);
    public Object getFeature(Object thisObject);
    public String[] getFilterFlags(Object thisObject);
    public Object getHelpContextIds(Object thisObject);
    public String getId(Object thisObject);
    public IItemLabelProvider getLabelProvider(Object thisObject);
    public Object getPropertyValue(Object thisObject);
    public boolean isCompatibleWith(Object object, Object anotherObject, ↪
        IItemPropertyDescriptor anotherItemPropertyDescriptor);
    public boolean isPropertySet(Object thisObject);
    public void resetPropertyValue(Object thisObject);
    public void setPropertyValue(Object thisObject, Object value);
// Protected Instance Fields
    protected IItemPropertyDescriptor itemPropertyDescriptor;
    protected Object object;
}

```

Hierarchy: Object → ItemPropertyDescriptorDecorator(IItemPropertyDescriptor)

ItemProvider

`org.eclipse.emf.edit.provider`

Class `ItemProvider` is a convenient item provider implementation base class for item providers that aren't adapters for EMF objects. It maintains a label text and image, a parent, collection of children objects, and optionally an adapter factory. It acts as its own `IChangeNotifier` (via its `changeNotifier` field) but also sends change notifications to its adapter factory, if it has one that is also an `IChangeNotifier`. If an item provider's parent or children are EMF objects, their adapter factories are typically used for this purpose.

The `children` list is implemented using `ItemProviderNotifyingArrayList` so that modification of the collection (using the standard `EList` interface) properly fires notifications to the `INotifyChangedListeners`. The `inverseAdd()` and `inverseRemove()` methods also update the parent for objects that are added to or removed from the list using the `IUpdateableItemParent.setParent()` method, if the `adapterFactory` is not null and the `IUpdateableParent.adapter` type is available for the object.

Most of the methods in this class are simple `get()` and `set()` methods for their corresponding fields. The `setText()` and `setImage()` methods additionally call `fireNotifyChanged()`, which sends notification to the `changeNotifier` and `adapterFactory`, if there is one. Some of the accessors have two versions, one of which includes the standard item provider argument and one that does not—for example, `getText()` and `getText(Object object)`. In these cases the nonobject version simply delegates to the object version, passing this as the object.

The `IChangeNotifier` methods, `addListener()`, `removeListener()`, and `fireNotifyChanged()`, delegate their implementations to the `changeNotifier` field, which is a lazily created instance of class `ChangeNotifier` (that is, a simple `ArrayList`-based `IChangeNotifier` implementation class), by default. The `getElements()` method, via a call to `getChildren()`, also returns the children of the item provider. The `getUpdateableText()` method simply delegates to `getText()`. As commands are not supported by default, the implementations of `createCommand()` and `getNewChildDescriptors()` return `UnexecutableCommand.INSTANCE` and `Collections.EMPTY_LIST`, respectively.

This class is often used as a convenient wrapper object to act as the input to a viewer. It lets you take a mixed collection of model objects and/or item providers, and show them as the elements of a structured viewer—for example, as the visible roots of a tree viewer. The structured viewer will not show this input object itself. However, if the viewer is displayed in a pane, it is typically used to provide the pane's title.

```
public class ItemProvider implements IChangeNotifier, IDisposable, IItemLabelProvider, ↵
IStructuredItemContentProvider, ITreelistContentProvider, IUpdateableItemParent {
// Public Constructors
    public ItemProvider();
    public ItemProvider(Collection children);
    public ItemProvider(String text);
    public ItemProvider(String text, Collection children);
    public ItemProvider(String text, Object image);
    public ItemProvider(String text, Object image, Collection children);
    public ItemProvider(String text, Object image, Object parent);
    public ItemProvider(String text, Object image, Object parent, Collection children);
    public ItemProvider(AdapterFactory adapterFactory);
    public ItemProvider(AdapterFactory adapterFactory, String text);
    public ItemProvider(AdapterFactory adapterFactory, String text, Object image);
    public ItemProvider(AdapterFactory adapterFactory, String text, Object image, Object ↵
parent);
    public ItemProvider(AdapterFactory adapterFactory, Collection children);
    public ItemProvider(AdapterFactory adapterFactory, String text, Collection children);
```

```

    public ItemProvider(AdapterFactory adapterFactory, String text, Object image, Collection ←
children);
    public ItemProvider(AdapterFactory adapterFactory, String text, Object image, Object parent, ←
Collection children);
// Public Inner Classes
    public class ItemProviderNotification;
    public class ItemProviderNotifyingArrayList;
// Property Accessor Methods (by property name)
    public AdapterFactory getAdapterFactory();
    public void setAdapterFactory(AdapterFactory adapterFactory);
    public EList getChildren();
    public EList getElements();
    public Object getImage();
    public void setImage(Object image);
    public Object getParent();
    public void setParent(Object parent);
    public String getText();
    public void setText(String text);
// Public Instance Methods
    public Command createCommand(Object object, EditingDomain editingDomain, Class ←
commandClass, CommandParameter commandParameter);
    public Collection getNewChildDescriptors(Object object, EditingDomain editingDomain, Object ←
sibling);
        public String getUpdateableText(Object object);
        public boolean hasChildren();
        public void setImage(Object object, Object image);
        public void setText(Object object, String text);
// Public Methods Overriding Object
    public String toString();
// Methods Implementing IChangeNotifier
    public void addListener(INotifyChangedListener listener);
    public void fireNotifyChanged(Notification notification);
    public void removeListener(INotifyChangedListener listener);
// Methods Implementing IDisposable
    public void dispose();
// Methods Implementing IIItemLabelProvider
    public Object getImage(Object object);
    public String getText(Object object);
// Methods Implementing IStructuredItemContentProvider
    public Collection getElements(Object object);
// Methods Implementing ITreelistContentProvider
    public Collection getChildren(Object object);
    public Object getParent(Object object);
    public boolean hasChildren(Object object);
// Methods Implementing IUpdateableItemParent
    public void setParent(Object object, Object parent);
// Protected Instance Fields
    protected AdapterFactory adapterFactory;
    protected IChangeNotifier changeNotifier;
    protected ItemProvider.ItemProviderNotifyingArrayList children;
    protected Object image;
    protected Object parent;
    protected String text;
}

```

empty

Hierarchy: Object → ItemProvider(IChangeNotifier, IDisposable, IIItemLabelProvider, IStructuredItemContentProvider, ITreelistContentProvider(IstructuredItemContentProvider), IUpdateableItemParent)

ItemProvider.ItemProviderNotification

org.eclipse.emf.edit.provider

This class implements notifications sent by an item provider. An `ItemProvider` is returned by `getNotifier()`, rather than an `EObject`, which we would normally expect. The `dispatch()` method sends the notification via the item provider's `fireNotifyChanged()` method, instead of `Notifier.eNotify()`.

```
public class ItemProvider.ItemProviderNotification extends NotificationImpl {
    // Public Constructors
    public ItemProviderNotification(int eventType, Object oldValue, Object newValue, int position);
    // Property Accessor Methods (by property name)
    public Object getNotifier();
    Overrides:NotificationImpl
    // Public Methods Overriding NotificationImpl
    public void dispatch();
}
```

Hierarchy: Object → `NotificationImpl(Notification, NotificationChain)` → `ItemProvider.ItemProviderNotification`

ItemProvider.ItemProviderNotifyingArrayList

org.eclipse.emf.edit.provider collection

This class is used to implement the `children list` for class `ItemProvider`. It overrides the notification methods to send instances of `ItemProviderNotification`, and it uses the inverse handshaking methods to maintain referential integrity by calling `IUpdateableItemParent.setParent()` on the target object's adapter.

```
public class ItemProvider.ItemProviderNotifyingArrayList extends NotifyingListImpl {
    // Public Constructors
    public ItemProviderNotifyingArrayList();
    empty
    public ItemProviderNotifyingArrayList(int initialCapacity);
    public ItemProviderNotifyingArrayList(Collection collection);
    // Protected Instance Methods
    protected NotificationImpl createNotification(int eventType, Object oldObject, Object newObject, int index);
    Overrides:NotifyingListImpl
    protected void dispatchNotification(Notification notification);
    Overrides:NotifyingListImpl
    protected boolean hasInverse();
    Overrides:NotifyingListImpl ← constant
    protected NotificationChain inverseAdd(Object object, NotificationChain notifications);
    Overrides:NotifyingListImpl
    protected NotificationChain inverseRemove(Object object, NotificationChain notifications);
    Overrides:NotifyingListImpl
    protected boolean isNotificationRequired();
    Overrides:NotifyingListImpl ← constant
}
```

Hierarchy: Object → `AbstractCollection(Collection)` → `AbstractList(List(Collection))` → `BasicEList(EList(List), Cloneable, Serializable)` → `Notifyin-gListImpl(NotifyingList(EList))` → `ItemProvider.ItemProviderNotifyingArrayList`

ItemProviderAdapter

`org.eclipse.emf.edit.provider.adapter`

`ItemProviderAdapter` is a convenient reusable base class for adapters that will be used as item providers. Although it does not directly implement the standard item provider interfaces itself, it provides base implementations of their methods for derived item providers that do. Default implementations are provided for the following interfaces: `IItemLabelProvider`, `IItemPropertySource`, `IStructuredItemContentProvider`, `ITreeItemContentProvider`, and `IEditDomainItemProvider`. A default implementation of `IUpdateableItemText.getText()`, which simply delegates to `getUpdateableText()`, is also provided, because editable text is often just the text itself.

Default implementations are provided for many of these methods by calling other methods, only a few of which are typically overridden. For example, the `getElements()` method returns the result of `getChildren()` and `hasChildren()` returns true if `getChildren().isEmpty()` is false. The `getChildren()` method returns the targets of all the references returned from the `getChildrenReferences()` method, which is one of the few methods that subclasses must override; it must return the collection of references whose targets are considered children of the specified object. The `getText()` method calls `toString()` on the specified object, although the `getText()` and `getImage()` methods are both usually overridden in subclasses.

The `createCommand()` method handles all the built-in EMF.Edit commands by delegating to one of the command-specific protected `create()` methods or returns `UnexecutableCommand.INSTANCE` for any other kind of command. Each of the protected command creation methods returns a new instance of its corresponding command. The `factorAddCommand()`, `factorRemoveCommand()`, and `factorMoveCommand()` methods are called by the `createCommand()` method if the feature has not been specified for an add, remove, or move command, respectively. These methods deduce the feature, and, in the case where add or remove involves more than one feature, create a `CompoundCommand` to compose several primitive commands, one per feature.

`ItemProviderAdapter` also maintains the set of property descriptors for the adapted object in the `itemPropertyDescriptor` field. The list is created by the `getPropertyDescriptors()` method, which is typically overridden in a subclass to add the appropriate descriptors. The `getPropertyValue()`, `setPropertyValue()`, `resetPropertyValue()`, and `isSetPropertyValue()` methods locate a given item property descriptor by calling `getPropertyDescriptor()`, and delegate to the same method on it.

The `IChangeNotifier` methods, `addListener()`, `removeListener()`, and `fireNotifyChanged()`, delegate their implementations to the `changeNotifier` field, which is a lazily created instance of class `ChangeNotifier` (that is, a simple `ArrayList`-based `IChangeNotifier` implementation class) by default.

The `getNewChildDescriptors()` method uses `collectNewChildDescriptors()` to populate the list that it returns. The implementation of the latter method, which is empty, should be overridden in subclasses. The methods from the `CreateChildCommand.Helper` interface rely on those from `ResourceLocator` to provide text and icon resources. This class implements these methods by delegating to the resource locator returned by the no-argument `getResourceLocator()` method, which is typically overridden in a subclass to return an appropriate EMF plug-in class. The second form of `getResourceLocator()` is a convenience method that attempts to find an appropriate item provider to use as a resource locator for the given object: It adapts the object with the `IItemLabelProvider` type and, if the result implements `ResourceLocator`, returns it.

```

public class ItemProviderAdapter extends AdapterImpl implements IChangeNotifier, IDisposable, ↵
CreateChildCommand.Helper, ResourceLocator {
// Public Constructors
    public ItemProviderAdapter(AdapterFactory adapterFactory);
// Property Accessor Methods (by property name)
    public AdapterFactory getAdapterFactory();
    public URL getBaseURL();                                Implements:ResourceLocator
    public void setTarget(Notifier target);                  Overrides:AdapterImpl
// Public Instance Methods
    public Command createCommand(Object object, EditingDomain domain, Class commandClass, ↵
CommandParameter commandParameter);
    public Collection getChildren(Object object);
    public Object getEditableValue(Object object);
    public Collection getElements(Object object);
    public Object getImage(Object object);                  constant
    public Collection getNewChildDescriptors(Object object, EditingDomain editingDomain, Object ←
sibling);
    public Object getParent(Object object);
    public IItemPropertyDescriptor getPropertyDescriptor(Object object, Object propertyId);
    public List getPropertyDescriptors(Object object);
    public Object getPropertyValue(Object object, String property);
    public String getText(Object object);
    public String getUpdateableText(Object object);
    public boolean hasChildren(Object object);
    public boolean isPropertySet(Object object, String property);
    public void resetPropertyValue(Object object, String property);
    public void setPropertyValue(Object object, String property, Object value);
// Public Methods Overriding AdapterImpl
    public boolean isAdapterForType(Object type);
// Methods Implementing IChangeNotifier
    public void addListener(INotifyChangedListener listener);
    public void fireNotifyChanged(Notification notification);
    public void removeListener(INotifyChangedListener listener);
// Methods Implementing IDisposable
    public void dispose();
// Methods Implementing CreateChildCommand.Helper
    public String getCreateChildDescription(Object owner, Object feature, Object child, ←
Collection selection);
    public Object getCreateChildImage(Object owner, Object feature, Object child, Collection ←
selection);
    public Collection getCreateChildResult(Object child);
    public String getCreateChildText(Object owner, Object feature, Object child, Collection ←
selection);
    public String getCreateChildToolTipText(Object owner, Object feature, Object child, ←
Collection selection);
// Methods Implementing ResourceLocator
    public Object getImage(String key);
    public String getString(String key);
    public String getString(String key, Object[] substitutions);
// Protected Instance Methods
    protected void collectNewChildDescriptors(Collection newChildDescriptors, Object object);
    protected Command createAddCommand(EditingDomain domain, EObject owner, EReference feature, ←
Collection collection, int index);
    protected CommandParameter createChildParameter(Object feature, Object child);
    protected Command createCopyCommand(EditingDomain domain, EObject owner, CopyCommand.Helper ←
helper);
    protected Command createCreateChildCommand(EditingDomain domain, EObject owner, EReference ←
feature, EObject value, int index, Collection collection);
    protected Command createCreateCopyCommand(EditingDomain domain, EObject owner, ←
CopyCommand.Helper helper);
    protected Command createDragAndDropCommand(EditingDomain domain, Object owner, float ←
location, int operations, int operation, Collection collection);
    protected Command createInitializeCopyCommand(EditingDomain domain, EObject owner, ←
CopyCommand.Helper helper);
    protected Command createMoveCommand(EditingDomain domain, EObject owner, EReference ←
feature, EObject value, int index);
    protected Command createRemoveCommand(EditingDomain domain, EObject owner, EReference ←
feature, Collection collection);
    protected Command createReplaceCommand(EditingDomain domain, EObject owner, EReference ←
feature, EObject value, Collection collection);
    protected Command createSetCommand(EditingDomain domain, EObject owner, EStructuralFeature ←

```

```

feature, Object value);
protected Command factorAddCommand(EditingDomain domain, CommandParameter commandParameter);
protected Command factorMoveCommand(EditingDomain domain, CommandParameter ←
commandParameter);
protected Command factorRemoveCommand(EditingDomain domain, CommandParameter ←
commandParameter);
protected EReference getChildReference(Object object, Object child);
protected Collection getChildrenReferences(Object object);
protected String getFeatureText(Object feature);
protected Object getReferenceValue(EObject object, EReference reference);
protected ResourceLocator getResourceLocator();
protected ResourceLocator getResourceLocator(Object anyObject);
protected EStructuralFeature getSetFeature(Object object, Object value);
protected Collection getSetFeatures(Object object);
protected String getString(String key, String s0);
protected String getString(String key, String s0, String s1);
protected String getTypeText(Object object);
// Protected Instance Fields
protected AdapterFactory adapterFactory;
protected IChangeNotifier changeNotifier;
protected List childrenReferences;
protected List itemPropertyDescriptors;
protected Collection targets;
}

```

Hierarchy: Object → AdapterImpl(Adapter) → ItemProviderAdapter(IChangeNotifier, IDisposable, CreateChildCommand.Helper, ResourceLocator)

ItemProviderDecorator

org.eclipse.emf.edit.provider

ItemProviderDecorator provides a convenient reusable base for item providers that will be used as decorators of other item providers. It provides default implementations for the IEditionDomainItemProvider, IItemLabelProvider, IItemPropertySource, IStructuredItemContentProvider, ITableItemLabelProvider, and ITreeItemContentProvider interfaces, and for the `getUpdateableText()` method of the IUpdateableItemText interface. All the implementations delegate to the decorated item provider.

It also adds itself as a listener of the decorated item provider, which must be an IChangeNotifier, so that it can forward the notifications.

```

public class ItemProviderDecorator implements INotifyChangedListener, IItemProviderDecorator, ←
IChangeNotifier, IDisposable {
// Public Constructors
    public ItemProviderDecorator(AdapterFactory adapterFactory);
// Property Accessor Methods (by property name)
    public AdapterFactory getAdapterFactory();
    public IChangeNotifier getDecoratedItemProvider();
Implements:IItemProviderDecorator
    public void setDecoratedItemProvider(IChangeNotifier decoratedItemProvider);
                                            Implements:IItemProviderDecorator
// Public Instance Methods
    public Command createCommand(Object object, EditingDomain domain, Class commandClass, ←
CommandParameter commandParameter);
    public Collection getChildren(Object object);
    public Object getColumnImage(Object object, int columnIndex);
    public String getColumnText(Object object, int columnIndex);
    public Object getEditableValue(Object object);
    public Collection getElements(Object object);
    public Object getImage(Object object);
    public Collection getNewChildDescriptors(Object object, EditingDomain editingDomain, Object ←
sibling);
    public Object getParent(Object object);
    public IIItemPropertyDescriptor getPropertyDescriptor(Object object, Object propertyId);

```

```

public List getPropertyDescriptors(Object object);
public String getText(Object object);
public String getUpdateableText(Object object);
public boolean hasChildren(Object object);
public boolean isAdapterForType(Object type);
// Public Methods Overriding Object
public String toString();
// Methods Implementing INotifyChangedListener
public void notifyChanged(Notification notification);
// Methods Implementing IChangeNotifier
public void addListener(INotifyChangedListener listener);
public void fireNotifyChanged(Notification notification);
public void removeListener(INotifyChangedListener listener);
// Methods Implementing IDisposable
public void dispose();
// Protected Instance Fields
protected AdapterFactory adapterFactory;
protected IChangeNotifier changeNotifier;
protected IChangeNotifier decoratedItemProvider;
}

```

Hierarchy: Object → ItemProviderDecorator(INotifyChangedListener, IItemProviderDecorator, IChangeNotifier, IDisposable)

ITreeItemContentProvider

org.eclipse.emf.edit.provider

Interface ITreeItemContentProvider is implemented by item providers to identify the objects to display in a tree viewer. It is the EMF.Edit “item” equivalent of the Eclipse UI framework’s ITreeContentProvider interface, as described in Chapter 3.

```

public interface ITreeItemContentProvider extends IStructuredItemContentProvider {
// Public Instance Methods
    public abstract Collection getChildren(Object object);
    public abstract Object getParent(Object object);
    public abstract boolean hasChildren(Object object);
}

```

Hierarchy: (ITreeItemContentProvider(IStructuredItemContentProvider))

IUpdateableItemParent

org.eclipse.emf.edit.provider

The IUpdateableItemParent interface is implemented by an item provider if it supports a generically updateable parent relation.

```

public interface IUpdateableItemParent {
// Public Instance Methods
    public abstract void setParent(Object object, Object parent);
}

```

IUpdateableItemText

org.eclipse.emf.edit.provider

The `IUpdateableItemText` interface is implemented by an item provider if it supports a generically updateable label. It is used to support edit-in-place tree items. The `getUpdateableText()` method returns the text to be displayed when editing begins, and `setText()` sets the label text.

```
public interface IUpdateableItemText {
    // Public Instance Methods
    public abstract String getUpdateableText(Object object);
    public abstract void setText(Object object, String text);
}
```

ReflectiveItemProvider

org.eclipse.emf.edit.provider.adapter

Class `ReflectiveItemProvider` provides a reflective implementation of an item provider that can be used to generically “provide for” any model object. The implementation uses the reflective `EObject` API to emulate the behavior of a generated item provider with default settings.

```
public class ReflectiveItemProvider extends ItemProviderAdapter implements ↪
IErasingDomainItemProvider, IStructuredItemContentProvider, ITreeItemContentProvider, ↪
IItemLabelProvider, IItemPropertySource {
    // Public Constructors
    public ReflectiveItemProvider(AdapterFactory adapterFactory);
    // Public Methods Overriding ItemProviderAdapter
    public Object getCreateChildImage(Object owner, Object feature, Object child, Collection ↪
selection);
    public Object getImage(Object object);
    public List getPropertyDescriptors(Object object);
    public String getText(Object object);
    // Public Methods Overriding AdapterImpl
    public void notifyChanged(Notification notification);
    // Protected Instance Methods
    protected void collectNewChildDescriptors(Collection newChildDescriptors, Object object);
                                                Overrides:ItemProviderAdapter
    protected List getAllConcreteSubclasses(EClass eClass);
    protected List getAllEClasses(EClass eClass);
    protected Collection getChildrenReferences(Object object);
    Overrides:ItemProviderAdapter
    protected String getFeatureText(Object feature);
    Overrides:ItemProviderAdapter
    protected String getTypeText(Object object);
    Overrides:ItemProviderAdapter
    // Protected Instance Fields
    protected List allEClasses;
    protected List allRoots;
}
```

Hierarchy: Object → AdapterImpl(Adapter) → ItemProviderAdapter(IChangeNotifier, IDisposable, CreateChildCommand.Helper, ResourceLocator) → ReflectiveItemProvider(IErasingDomainItemProvider, IStructuredItemContentProvider, ITreeItemContentProvider(IStructuredItemContentProvider), IItemLabelProvider, IItemPropertySource)

ReflectiveItemProviderAdapterFactory

org.eclipse.emf.edit.provider

Class `ReflectiveItemProviderAdapterFactory` can be used to adapt objects from any model. The `createAdapter()` method returns a singleton instance of class `ReflectiveItemProvider`, which is used for all adapted objects.

```
public class ReflectiveItemProviderAdapterFactory extends AdapterFactoryImpl implements ↪
    ComposeableAdapterFactory, IChangeNotifier {
    // Public Constructors
    public ReflectiveItemProviderAdapterFactory();
    // Property Accessor Methods (by property name)
    public void setParentAdapterFactory(ComposedAdapterFactory parentAdapterFactory);
    // Methods Overriding AdapterFactoryImpl
    public Adapter adapt(Notifier notifier, Object type);
    public Object adapt(Object object, Object type);
    public Adapter createAdapter(Notifier target);
    public boolean isFactoryForType(Object type);
    // Methods Implementing IChangeNotifier
    public void addListener(INotifyChangedListener notifyChangedListener);
    public void fireNotifyChanged(Notification notification);
    public void removeListener(INotifyChangedListener notifyChangedListener);
    // Protected Instance Fields
    protected IChangeNotifier changeNotifier;
    protected ComposedAdapterFactory parentAdapterFactory;
    protected ReflectiveItemProvider reflectiveItemProviderAdapter;
    protected Collection supportedTypes;
}
```

Hierarchy: Object → AdapterFactoryImpl(AdapterFactory) → ReflectiveItemProviderAdapterFactory(CheckableAdapterFactory(AdapterFactory), IChangeNotifier)

The org.eclipse.emf.edit.provider.resource Package

The `org.eclipse.emf.edit.provider.resource` package contains the default item providers and item provider adapter factory for resources and resource sets. Unlike most item providers, the ones in this package are not generated adapters for model objects. Although resources and resource sets aren't based on Ecore-modeled classes, they do behave much like `EObjects`. In particular, they are notifiers (that is, their interfaces extend `Notifier`), enabling their item providers to follow a pattern very similar to the one used in generated item providers.

ResourceItemProvider

org.eclipse.emf.edit.provider.resource adapter

`ResourceItemProvider` is the default item provider for a resource. The `getParent()` and `getChildren()` methods return the resource set and the content of the resource, respectively. The `getText()` method returns the resource's URI. No properties are provided and creation commands are disabled.

```

public class ResourceItemProvider extends ItemProviderAdapter implements ←
IErasingDomainItemProvider, IStructuredItemContentProvider, ITreeItemContentProvider, ←
IItemLabelProvider, IItemPropertySource {
// Public Constructors
    public ResourceItemProvider(AdapterFactory adapterFactory);
// Property Accessor Methods (by property name)
    public ResourceLocator getResourceLocator();
Overrides:ItemProviderAdapter
// Public Methods Overriding ItemProviderAdapter
    public Collection getChildren(Object object);
    public Collection getChildrenReferences(Object object);
    public Object getImage(Object object);
    public Collection getNewChildDescriptors(Object object, EditingDomain editingDomain, Object ←
sibling);
    public Object getParent(Object object);
    public List getPropertyDescriptors(Object object);
    public String getText(Object object);
// Public Methods Overriding AdapterImpl
    public void notifyChanged(Notification notification);
// Protected Instance Methods
    protected void collectNewChildDescriptors(Collection newChildDescriptors, Object object);
Overrides:ItemProviderAdapter
}

```

Hierarchy: Object → AdapterImpl(Adapter) → ItemProviderAdapter(IChangeNotifier, IDisposable, CreateChildCommand.Helper, ResourceLocator) → ResourceItemProvider(IErasingDomainItemProvider, IStructuredItemContentProvider, ITreeItemContentProvider(StructuredItemContentProvider), IItemLabelProvider, IItemPropertySource)

ResourceItemProviderAdapterFactory

org.eclipse.emf.edit.provider.resource

Class ResourceItemProviderAdapterFactory is the default item provider adapter factory for resources and resource sets. Although it is not generated, its implementation follows the same Stateful pattern that can be used in a generated adapter factory.

Depending on the type of object being adapted, Resource or ResourceSet, the createAdapter() method delegates to either createResourceAdapter(), which returns a new instance of ResourceItemProvider, or to createResourceSetAdapter(), which returns a new instance of ResourceSetItemProvider.

```

public class ResourceItemProviderAdapterFactory extends AdapterFactoryImpl implements ←
ComposeableAdapterFactory, IChangeNotifier {
// Public Constructors
    public ResourceItemProviderAdapterFactory();
// Property Accessor Methods (by property name)
    public void setParentAdapterFactory(ComposedAdapterFactory parentAdapterFactory);
    Implements:ComposeableAdapterFactory
    public ComposeableAdapterFactory getRootAdapterFactory();
    Implements:ComposeableAdapterFactory
// Public Instance Methods
    public Adapter createResourceAdapter();
    public Adapter createResourceSetAdapter();
// Public Methods Overriding AdapterFactoryImpl
    public Adapter adapt(Notifier notifier, Object type);
    public Object adapt(Object object, Object type);
    public Adapter createAdapter(Notifier target);
    public boolean isFactoryForType(Object type);
// Methods Implementing IChangeNotifier
    public void addListener(INotifyChangedListener notifyChangedListener);
    public void fireNotifyChanged(Notification notification);
    public void removeListener(INotifyChangedListener notifyChangedListener);

```

```

// Protected Class Fields
protected static Package resourcePackage;
// Protected Instance Fields
protected IChangeNotifier changeNotifier;
protected ComposedAdapterFactory parentAdapterFactory;
protected Collection supportedTypes;
}

```

Hierarchy: Object → AdapterFactoryImpl(AdapterFactory) → ResourceItemProviderAdapterFactory(ComposeableAdapterFactory(AdapterFactory), IChangeNotifier)

ResourceSetItemProvider

org.eclipse.emf.edit.provider.resource adapter

ResourceSetItemProvider is the default item provider for a resource set. The getParent() method returns null and getChildren() returns the resources in the set. The getText() method returns a constant label. No properties are provided and creation commands are disabled.

```

public class ResourceSetItemProvider extends ItemProviderAdapter implements ↪
IErasingDomainItemProvider, IStructuredItemContentProvider, ITreeItemContentProvider, ↪
IItemLabelProvider, IItemPropertySource {
// Public Constructors
    public ResourceSetItemProvider(AdapterFactory adapterFactory);
// Property Accessor Methods (by property name)
    public ResourceLocator getResourceLocator();
Overrides:ItemProviderAdapter
// Public Methods Overriding ItemProviderAdapter
    public Collection getChildren(Object object);
    public Collection getChildrenReferences(Object object);
    public Object getImage(Object object);
    public Collection getNewChildDescriptors(Object object, EditingDomain editingDomain, Object ↪
sibling);
    public Object getParent(Object object);                                constant
    public List getPropertyDescriptors(Object object);
    public String getText(Object object);
// Public Methods Overriding AdapterImpl
    public void notifyChanged(Notification notification);
// Protected Instance Methods
    protected void collectNewChildDescriptors(Collection newChildDescriptors, Object object);
                                                     Overrides:ItemProviderAdapter
}

```

Hierarchy: Object → AdapterImpl(Adapter) → ItemProviderAdapter(IChangeNotifier, IDisposable, CreateChildCommand.Helper, ResourceLocator) → ResourceSetItemProvider(IErasingDomainItemProvider, IStructuredItemContentProvider, ITreeItemContentProvider(ISTructuredItemContentProvider), IItemLabelProvider, IItemPropertySource)

The org.eclipse.emf.edit.tree Package

The org.eclipse.emf.edit.tree package is the generated interface package for the Tree model. It contains the standard generated interfaces, as described in Chapter 9.

A Tree model is a simple one-class model that can be used to create a tree structure fronting another data model. It allows you to impose a different hierarchical structure on an underlying set of data objects with a structure of their own.

TreeFactory

```
org.eclipse.emf.edit.tree
eobject
```

TreeFactory is a standard generated factory interface for the Tree model. The `createTreeNode()` method creates an instance of the one and only class in the model.

```
public interface TreeFactory extends EFactory {
    // Public Constants
    public static final TreeFactory eINSTANCE;
    // Property Accessor Methods (by property name)
    public abstract TreePackage getTreePackage();
    // Public Instance Methods
    public abstract TreeNode createTreeNode();
}
```

Hierarchy: (TreeFactory(EFactory(EModelElement(EObject(Notifier)))))

TreeNode

```
org.eclipse.emf.edit.tree
eobject
```

TreeNode is the generated interface for the one and only class (**TreeNode**) in the Tree model. The `getChildren()` and `getParent()` methods are used to access the two ends of the bidirectional association between a node and its children, which are also `TreeNodes`.

A `TreeNode` is only used to represent the tree hierarchy. The actual data of each node is stored in a referenced data object, which is accessed using the `getData()` method.

```
public interface TreeNode extends EObject {
    // Property Accessor Methods (by property name)
    public abstract EList getChildren();
    public abstract EObject getData();
    public abstract void setData(EObject value);
    public abstract TreeNode getParent();
    public abstract void setParent(TreeNode value);
}
```

Hierarchy: (TreeNode(EObject(Notifier)))

TreePackage

```
org.eclipse.emf.edit.tree
eobject
```

TreePackage is a standard generated package interface for the Tree model.

```
public interface TreePackage extends EPackage {
    // Public Constants
    public static final int TREE_NODE; =0
    public static final int TREE_NODE_FEATURE_COUNT; =3
    public static final int TREE_NODE_CHILDREN; =1
    public static final int TREE_NODE_DATA; =2
    public static final int TREE_NODE_PARENT; =0
    public static final TreePackage eINSTANCE;
    public static final String eNAME; ="tree"
    public static final String eNS_PREFIX; ="tree"
    public static final String eNS_URI; ="http://www.eclipse.org/emf/2002/
Tree"
```

```
// Property Accessor Methods (by property name)
public abstract TreeFactory getTreeFactory();
public abstract EClass getNode();
public abstract EReference getNode_Children();
public abstract EReference getNode_Data();
public abstract EReference getNode_Parent();
}
```

Hierarchy: (TreePackage (EPackage (ENamedElement (EModelElement (EObject (Notifier)))))))

The org.eclipse.emf.edit.tree.provider Package

The `org.eclipse.emf.edit.tree.provider` package is the generated item provider package for the Tree model. It contains an item provider adapter factory and an item provider for **TreeNode**, the one and only class in the model. It provides the default view of a Tree model: a view of the tree's **data** objects, structured according to the tree nodes.

TreeItemProviderAdapterFactory

org.eclipse.emf.edit.tree.provider

`TreeItemProviderAdapterFactory` is a standard generated item provider adapter factory for the Tree model that uses the Stateful pattern described in Chapter 10. The `createTreeNodeAdapter()` method returns a new instance of class `TreeNodeItemProvider`.

```
public class TreeItemProviderAdapterFactory extends TreeAdapterFactory implements ↪
ComposeableAdapterFactory, IChangeNotifier, IDisposable {
// Public Constructors
    public TreeItemProviderAdapterFactory();
// Property Accessor Methods (by property name)
    public void setParentAdapterFactory(ComposedAdapterFactory parentAdapterFactory);
                                            Implements:ComposeableAdapterFactory
    public ComposeableAdapterFactory getRootAdapterFactory();                      Implements:ComposeableAdapterFactory
// Public Methods Overriding TreeAdapterFactory
    public Adapter createTreeNodeAdapter();
    public boolean isFactoryForType(Object type);
// Public Methods Overriding AdapterFactoryImpl
    public Adapter adapt(Notifier notifier, Object type);
    public Object adapt(Object object, Object type);
    public Adapter adaptNew(Notifier object, Object type);
// Methods Implementing IChangeNotifier
    public void addListener(INotifyChangedListener notifyChangedListener);
    public void fireNotifyChanged(Notification notification);
    public void removeListener(INotifyChangedListener notifyChangedListener);
// Methods Implementing IDisposable
    public void dispose();
// Protected Instance Fields
    protected IChangeNotifier changeNotifier;
    protected Disposable disposable;
    protected ComposedAdapterFactory parentAdapterFactory;
    protected Collection supportedTypes;
}
```

Hierarchy: Object → AdapterFactoryImpl(AdapterFactory) → TreeAdapterFactory → TreeItemProviderAdapterFactory(ComposeableAdapterFactory(AdapterFactory), IChangeNotifier, IDisposable)

TreeNodeItemProvider

org.eclipse.emf.edit.tree.provider.adapter

Class `TreeNodeItemProvider` is the generated item provider for the `TreeNode` model class. The children and parent reflect those of the `TreeNode`, but the `getText()`, `getImage()`, and `getProperties()` methods have been hand-modified to return values from the `data` reference of the target `TreeNode`, instead of from the target itself. It delegates these methods to the data object's item provider using an `AdapterFactoryItemDelegator`.

This item provider also registers itself as a listener of the item provider for the tree node's data object and implements the `notifyChanged()` method to forward the data object's notifications as if they were its own, that is, by changing the notification's notifier to be the tree node.

The `collectNewChildDescriptors()` method returns an empty list, disabling child creation.

```
public class TreeNodeItemProvider extends ItemProviderAdapter implements ↪
IErasingDomainItemProvider, INotifyChangedListener, IStructuredItemContentProvider, ↪
ITreeItemContentProvider, IItemLabelProvider, IItemPropertySource {
// Public Constructors
    public TreeNodeItemProvider(AdapterFactory adapterFactory);
// Property Accessor Methods (by property name)
    public void setTarget(Notifier target);
Overrides:ItemProviderAdapter
// Public Methods Overriding ItemProviderAdapter
    public void dispose();
    public Collection getChildrenReferences(Object object);
    public Object getImage(Object object);
    public Object getParent(Object object);
    public List getPropertyDescriptors(Object object);
    public String getText(Object object);
// Public Methods Overriding AdapterImpl
    public void notifyChanged(Notification notification);
// Protected Instance Methods
    protected void collectNewChildDescriptors(Collection newChildDescriptors, Object object);
    Overrides:ItemProviderAdapter empty
// Protected Instance Fields
    protected IChangeNotifier delegateItemProvider;
    protected AdapterFactoryItemDelegator itemDelegator;
}
```

Hierarchy: Object → AdapterImpl(Adapter) → ItemProviderAdapter(IChangeNotifier, IDisposable, CreateChildCommand.Helper, ResourceLocator) → TreeNodeItemProvider(IErasingDomainItemProvider, INotifyChangedListener, IStructuredItemContentProvider, ITreeItemContentProvider(StructuredItemContentProvider), IItemLabelProvider, IItemPropertySource)

The org.eclipse.emf.edit.tree.util Package

The `org.eclipse.emf.edit.tree.util` package contains the Tree model's generated adapter factory and switch classes, as described in Chapter 9.

TreeAdapterFactory

org.eclipse.emf.edit.tree.util

`TreeAdapterFactory` is a standard generated adapter factory for the Tree model. It is the base class of `TreeItemProviderAdapterFactory`.

```

public class TreeAdapterFactory extends AdapterFactoryImpl {
    // Public Constructors
    public TreeAdapterFactory();
    // Public Instance Methods
    public Adapter createEObjectAdapter();
    public Adapter createTreeNodeAdapter();
    // Public Methods Overriding AdapterFactoryImpl
    public Adapter createAdapter(Notifier target);
    public boolean isFactoryForType(Object object);
    // Protected Class Fields
    protected static TreePackage modelPackage;
    // Protected Instance Fields
    protected TreeSwitch modelSwitch;
}

```

Hierarchy. Object → AdapterFactoryImpl (AdapterFactory) → TreeAdapterFactory

TreeSwitch

org.eclipse.emf.edit.tree.util

TreeSwitch is a standard generated switch class for the Tree model. It's used in the implementation of the generated TreeAdapterFactory class.

```

public class TreeSwitch {
    // Public Constructors
    public TreeSwitch();
    // Public Instance Methods
    public Object caseTreeNode(TreeNode object);
    public Object defaultCase(EObject object);
    public Object doSwitch(EObject theEObject);
    // Protected Class Fields
    protected static TreePackage modelPackage;
}

```

Chapter 20. The org.eclipse.emf.edit.ui Plug-In

The `org.eclipse.emf.edit.ui` plug-in contains the Eclipse UI-dependent portion of the EMF.Edit framework. The classes in it handle the connection of an EMF model to the UI, with an implementation based on the Eclipse UI Framework. The majority of these classes delegate much of their implementation to one of the two important UI-independent EMF.Edit mechanisms: item providers and commands.

Requires: `org.eclipse.emf.edit(org.eclipse.core.resources,`
`org.eclipse.emf.common(org.eclipse.core.runtime)→,`
`org.eclipse.emf.ecore(org.apache.xerces)→)→,` `org.eclipse.emf.com-`
`mon.ui(org.eclipse.ui)→)`

The `org.eclipse.emf.edit.ui` Package

The `org.eclipse.emf.edit.ui` package provides the plug-in class for the `org.eclipse.emf.edit.ui` plug-in.

`EMFEditUIPlugin`

`org.eclipse.emf.edit.ui`

`EMFEditUIPlugin` is the plug-in class for the `org.eclipse.emf.edit.ui` plug-in. Like all other EMF plug-in classes, it uses the Singleton design pattern, with its single instance available as the constant `INSTANCE` field, and extends `EMFPlugin` using the pattern described in Section 10.6.

When running within Eclipse, the Eclipse plug-in class of type `EMFEditUIPlugin`.`Implementation` is available from the static `getPlugin()` method. The same object is returned by `getPluginResourceLocator()` and is used as a delegate by the `EMFPlugin` implementations of `getBaseURL()`, `getImage()`, `getString()`, and `log()` to make use of the platform's plug-in support facilities. When running stand-alone, `getPluginResourceLocator()` returns null, and the base class uses its own implementations.

```
public final class EMFEditUIPlugin extends EMFPlugin {
    // No Constructor
    // Public Constants
    public static final EMFEditUIPlugin INSTANCE;
    // Public Inner Classes
    public static class Implementation;
    // Public Class Methods
    public static EMFEditUIPlugin.Implementation getPlugin();
    // Property Accessor Methods (by property name)
    public ResourceLocator getPluginResourceLocator();                                Overrides:EMFPlugin
}
```

Hierarchy: Object → `EMFPlugin(ResourceLocator, Logger)` → `EMFEditUIPlugin`

EMFEditUIPlugin.Implementation

org.eclipse.emf.edit.ui

EMFEditUIPlugin.Implementation extends org.eclipse.core.runtime.Plugin, the base class that represents a plug-in within Eclipse, via EMFPlugin.EclipsePlugin. It is instantiated by the platform, and then used to provide platform-based implementations for most of the methods of the EMFEditUIPlugin plug-in class. It is also through this class that other Eclipse facilities, including the preference store and plug-in descriptor, can be accessed.

```
public static class EMFEditUIPlugin.Implementation extends EMFPlugin.EclipsePlugin {
    // Public Constructors
    public Implementation(IPluginDescriptor descriptor);
}
```

Hierarchy: Object → Plugin → EMFPlugin.EclipsePlugin(ResourceLocator, Logger) → EMFEditUIPlugin.Implementation

The org.eclipse.emf.edit.ui.action Package

The org.eclipse.emf.edit.ui.action package provides several action implementations and base classes that delegate their function to EMF commands. Implementations of the standard actions (that is, **Undo**, **Redo**, **Cut**, **Copy**, **Paste**, and **Delete**) as well as actions for child and sibling creation are provided. An action bar contributor implementation base class is also included.

CommandAction

org.eclipse.emf.edit.ui.action

Class CommandAction is used to implement a selection-based JFace IAction, which might appear in a menu or on a toolbar, by delegating to an EMF Command. All possible aspects of the action are delegated to the command, namely the enablement state and, if it implements CommandActionDelegate, the text, icon, and tool tip text. A derived class must override the createActionCommand() method to return a command based on the specified EditingDomain and collection of selected objects.

```
public class CommandAction implements IEditorActionDelegate {
    // Public Constructors
    public CommandAction();
    // Methods Implementing IEditorActionDelegate
    public void setActiveEditor(IAction action, IEditorPart editorPart); empty
    // Methods Implementing IActionDelegate
    public void run(IAction action);
    public void selectionChanged(IAction action, ISelection selection);
    // Protected Instance Methods
    protected Command createActionCommand(EditingDomain editingDomain, Collection collection);
    protected ImageDescriptor getDefaultImageDescriptor(); constant
    protected ImageDescriptor objectToImageDescriptor(Object object);
    // Protected Instance Fields
    protected IAction action;
    protected Collection collection;
    protected Command command;
    protected EditingDomain editingDomain;
    protected IEditorPart editorPart;
}
```

Hierarchy: Object → CommandAction(IEditorActionDelegate(IActionDelegate))

CommandActionHandler

org.eclipse.emf.edit.ui.action

`CommandActionHandler` is the common base class for implementing the built-in selection-based actions (**Cut**, **Copy**, **Paste**, and **Delete**) by delegating to an EMF.Edit Command. Subclasses must override the `createCommand()` method to create a command for the specified selection using domain, the handler's `EditingDomain`.

```
public class CommandActionHandler extends SelectionListenerAction {
    // Public Constructors
    public CommandActionHandler(EditingDomain domain);
    public CommandActionHandler(EditingDomain domain, String label);
    // Property Accessor Methods (by property name)
    public EditingDomain getEditingDomain();
    public void setEditingDomain(EditingDomain domain);
    // Public Instance Methods
    public Command createCommand(Collection selection);
    // Public Methods Overriding SelectionListenerAction
    public boolean updateSelection(IStructuredSelection selection);
    // Public Methods Overriding Action
    public void run();
    // Protected Instance Fields
    protected Command command;
    protected EditingDomain domain;
}
```

Hierarchy: Object → Action(IAction) → SelectionListenerAction(ISelectionChangedListener) → CommandActionHandler

CopyAction

org.eclipse.emf.edit.ui.action

Class `CopyAction` is a command action handler that implements the standard **Copy** action. Its `createCommand()` method calls `CopyToClipboardCommand.create()`.

```
public class CopyAction extends CommandActionHandler {
    // Public Constructors
    public CopyAction(EditingDomain domain);
    public CopyAction();
    // Property Accessor Methods (by property name)
    public void setActiveEditor(IEditorPart editorPart);
    // Public Methods Overriding CommandActionHandler
    public Command createCommand(Collection selection);
}
```

Hierarchy: Object → Action(IAction) → SelectionListenerAction(ISelectionChangedListener) → CommandActionHandler → CopyAction

CreateChildAction

org.eclipse.emf.edit.ui.action

Class `CreateChildAction` is a static selection command action that adds a new object, specified in a descriptor, under a selected parent. Its `createCommand()` method calls `CreateChildCommand.create()`, passing the one and only object in `selection` as the `owner` argument.

```

public class CreateChildAction extends StaticSelectionCommandAction {
    // Public Constructors
    public CreateChildAction(IEditorPart editorPart, ISelection selection, Object descriptor);
    // Protected Instance Methods
    protected Command createActionCommand(EditingDomain editingDomain, Collection collection);
        Overrides:StaticSelectionCommandAction
    // Protected Instance Fields
    protected Object descriptor;
}

```

Hierarchy: Object → Action(IAction) → StaticSelectionCommandAction → Create-ChildAction

CreateSiblingAction

org.eclipse.emf.edit.ui.action

Class CreateSiblingAction is a static selection command action that adds a new object, specified in a descriptor, after a selected sibling object. Its createActionCommand() method calls CreateChildCommand.create(), passing null as the owner argument, to specify sibling, instead of child, creation.

```

public class CreateSiblingAction extends StaticSelectionCommandAction {
    // Public Constructors
    public CreateSiblingAction(IEditorPart editorPart, ISelection selection, Object descriptor);
    // Protected Instance Methods
    protected Command createActionCommand(EditingDomain editingDomain, Collection collection);
        Overrides:StaticSelectionCommandAction
    // Protected Instance Fields
    protected Object descriptor;
}

```

Hierarchy: Object → Action(IAction) → StaticSelectionCommandAction → Create-SiblingAction

CutAction

org.eclipse.emf.edit.ui.action

Class CutAction is a command action handler that implements the standard Cut action. Its createCommand() method calls CutToClipboardCommand.create().

```

public class CutAction extends CommandActionHandler {
    // Public Constructors
    public CutAction(EditingDomain domain);
    public CutAction();
    // Property Accessor Methods (by property name)
    public void setActiveEditor(IEditorPart editorPart);
    // Public Methods Overriding CommandActionHandler
    public Command createCommand(Collection selection);
}

```

Hierarchy: Object → Action(IAction) → SelectionListenerAction(ISelectionChangedListener) → CommandActionHandler → CutAction

DelegatingCommandAction

org.eclipse.emf.edit.ui.action

A DelegatingCommandAction is a JFace Action that wraps and delegates its implementation to an IEditorActionDelegate, usually a CommandAction.

```
public class DelegatingCommandAction extends Action implements ISelectionListener, ↵
ISelectionChangedListener {
// Public Constructors
    public DelegatingCommandAction(IEditorActionDelegate editorActionDelegate);
// Property Accessor Methods (by property name)
    public void setActiveEditor(IEditorPart editorPart);
// Public Methods Overriding Action
    public void run();
// Methods Implementing ISelectionListener
    public void selectionChanged(IWorkbenchPart part, ISelection selection);
// Methods Implementing ISelectionChangedListener
    public void selectionChanged(SelectionChangedEvent event);
// Protected Instance Methods
    protected void handleSelection(ISelection selection);
    protected void registerSelectionListener(IEditorPart editorPart);
    protected void selectionChanged(ISelection selection);
    protected void unregisterSelectionListener(IEditorPart editorPart);
// Protected Instance Fields
    protected IEditorActionDelegate editorActionDelegate;
    protected IEditorPart editorPart;
}
```

Hierarchy: Object → Action (IAction) → DelegatingCommandAction (ISelectionListener, ISelectionChangedListener)

DeleteAction

org.eclipse.emf.edit.ui.action

Class DeleteAction is a command action handler that implements the standard **Delete** action. Its createCommand() method calls RemoveCommand.create().

```
public class DeleteAction extends CommandActionHandler {
// Public Constructors
    public DeleteAction(EditDomain domain);
    public DeleteAction();
// Property Accessor Methods (by property name)
    public void setActiveEditor(IEditorPart editorPart);
// Public Methods Overriding CommandActionHandler
    public Command createCommand(Collection selection);
}
```

Hierarchy: Object → Action (IAction) → SelectionListenerAction (ISelectionChangedListener) → CommandActionHandler → DeleteAction

EditingDomainActionBarContributor

org.eclipse.emf.edit.ui.action

EditingDomainActionBarContributor is an action bar contributor base class for an editor, multipage or otherwise, that implements the IEditingDomainProvider interface. It automatically hooks up the **Edit** menu's **Undo**, **Redo**, **Cut**, **Copy**, **Paste**, and **Delete** actions to corresponding EMF actions, supported by the EditingDomain.

The **Cut**, **Copy**, **Paste**, and **Delete** actions are registered as listeners of the `ISelectionProvider` for the active editor's site (`IEditorSite`), which keeps them up to date. The actions are also refreshed in the `propertyChanged()` method, which is called whenever the editor fires change notification to its property listeners (`IPropertyListeners`), of which the contributor is one.

This contributor class also implements the `IMenuListener` interface, allowing it to be used by an editor to contribute the **Edit** menu actions to a pop-up menu by calling the `menuAboutToShow()` method.

```
public class EditingDomainActionBarContributor extends MultiPageEditorActionBarContributor {
    implements IMenuListener, IPropertyListener {
    // Public Constructors
    public EditingDomainActionBarContributor();
    // Property Accessor Methods (by property name)
    public void setActiveEditor(IEditorPart part);
    Overrides:MultiPageEditorActionBarContributor
    public void setActivePage(IEditorPart part);
    Overrides:MultiPageEditorActionBarContributor empty
    // Public Instance Methods
    public void activate();
    public void deactivate();
    public void shareGlobalActions(IPage page, IActionBars actionBars);
    public void update();
    // Public Methods Overriding EditorActionBarContributor
    public void contributeToMenu(IMenuManager menuManager);
    public void contributeToStatusLine(IStatuslineManager statusLineManager);
    public void contributeToToolBar(IToolBarManager toolBarManager);
    public void init(IActionBars actionBars);
    // Methods Implementing IMenuListener
    public void menuAboutToShow(IMenuManager menuManager);
    // Methods Implementing IPropertyListener
    public void propertyChanged(Object source, int id);
    // Protected Instance Fields
    protected IEditorPart activeEditor;
    protected CopyAction copyAction;
    protected CutAction cutAction;
    protected DeleteAction deleteAction;
    protected PasteAction pasteAction;
    protected RedoAction redoAction;
    protected UndoAction undoAction;
}
```

Hierarchy: Object → EditorActionBarContributor (IEditorActionBarContributor) → MultiPageEditorActionBarContributor → EditingDomainActionBarContributor (IMenuListener, IPropertyListener)

PasteAction

org.eclipse.emf.edit.ui.action

Class `PasteAction` is a command action handler that implements the standard **Paste** action. Its `createCommand()` method calls `PasteFromClipboardCommand.create()`.

```
public class PasteAction extends CommandActionHandler {
    // Public Constructors
    public PasteAction(EditingDomain domain);
    public PasteAction();
    // Property Accessor Methods (by property name)
    public void setActiveEditor(IEditorPart editorPart);
    // Public Methods Overriding CommandActionHandler
    public Command createCommand(Collection selection);
}
```

Hierarchy: Object → Action (IAction) → SelectionListenerAction (ISelectionChangedListener) → CommandActionHandler → PasteAction

RedoAction

org.eclipse.emf.edit.ui.action

Class RedoAction implements the standard **Redo** action by delegating to the CommandStack maintained by an EditingDomain. The action's enablement and execution are implemented with calls to the canRedo() and redo() command stack methods, respectively.

```
public class RedoAction extends Action {
    // Public Constructors
    public RedoAction(EditingDomain domain);
    public RedoAction();
    // Property Accessor Methods (by property name)
    public void setActiveEditor(IEditorPart editorPart);
    public EditingDomain getEditingDomain();
    public void setEditingDomain(EditingDomain domain);
    // Public Instance Methods
    public void update();
    // Public Methods Overriding Action
    public void run();
    // Protected Instance Fields
    protected EditingDomain domain;
}
```

Hierarchy: Object → Action (IAction) → RedoAction

StaticSelectionCommandAction

org.eclipse.emf.edit.ui.action

StaticSelectionCommandAction is a base class for implementing an IAction, which might appear in a menu or on a toolbar, by delegating to an EMF Command, only when it is guaranteed that the selection will not change during the life of the action. In other words, the action itself will be created based on the selection and destroyed when the selection changes. All possible aspects of the action are delegated to the command, namely the enablement state and, if it implements the CommandActionDelegate interface, the text, icon, and tool tip text. However, this need only be done once, at the time the action is created.

Subclasses must provide an implementation for the createActionCommand() method that creates the command to perform this action. They can also override getDefaultImageDescriptor() to provide a default icon and disable() to set the action's state when a command cannot be created.

```
public abstract class StaticSelectionCommandAction extends Action {
    // Public Constructors
    public StaticSelectionCommandAction(IEditorPart editorPart);
    public StaticSelectionCommandAction();
    // Public Instance Methods
    public void configureAction(ISelection selection);
    // Public Methods Overriding Action
    public void run();
    // Protected Instance Methods
    protected abstract Command createActionCommand(EditingDomain editingDomain, Collection<Object> collection);
    protected void disable();
    protected ImageDescriptor getDefaultImageDescriptor(); constant
    protected ImageDescriptor objectToImageDescriptor(Object object);
```

```
// Protected Instance Fields
protected Command command;
protected EditingDomain editingDomain;
}
```

Hierarchy: Object → Action (IAction) → StaticSelectionCommandAction

UndoAction

org.eclipse.emf.edit.ui.action

Class UndoAction implements the standard **Undo** action by delegating to the CommandStack maintained by an EditingDomain. The action's enablement and execution are implemented with calls to the canUndo() and undo() command stack methods, respectively.

```
public class UndoAction extends Action {
// Public Constructors
    public UndoAction(EditingDomain domain);
    public UndoAction();
// Property Accessor Methods (by property name)
    public void setActiveEditor(IEditorPart editorPart);
    public EditingDomain getEditingDomain();
    public void setEditingDomain(EditingDomain domain);
// Public Instance Methods
    public void update();
// Public Methods Overriding Action
    public void run();
// Protected Instance Fields
    protected EditingDomain domain;
}
```

Hierarchy: Object → Action (IAction) → UndoAction

The org.eclipse.emf.edit.ui.celleditor Package

This org.eclipse.emf.edit.ui.celleditor package provides support for in-place (cell) editing in a tree or table-tree viewer. It also includes an editor dialog for editing multivalued EMF attributes or references.

AdapterFactoryTableTreeEditor

org.eclipse.emf.edit.ui.celleditor

Class AdapterFactoryTableTreeEditor is a base class for implementing a TableTreeEditor that delegates to adapters produced by an AdapterFactory. It includes keyboard navigation support and scaffolding for customized cell editing. Subclasses must override the createDropDownEditor() method, and also the hasLaunchedEditor() and createLaunchedEditor() methods, if cell editor launching is supported.

```
public class AdapterFactoryTableTreeEditor extends ExtendedTableTreeEditor {
// Public Constructors
    public AdapterFactoryTableTreeEditor(TableTree tableTree, AdapterFactory adapterFactory);
// Property Accessor Methods (by property name)
    public AdapterFactory getAdapterFactory();
    public void setAdapterFactory(AdapterFactory adapterFactory);
// Public Instance Methods
    public void apply();
    public void cancel();
    public Control createDropDownEditor(Composite parent, Object object, int column);
    constant
```

```

    public void createLaunchedEditor(Composite parent, Object object, int ←
column);                                empty
    public IItemPropertyDescriptor getColumnPropertyDescriptor(Object object, int column);
constant
    public boolean hasDropDownEditor(Object object, int column);
    public boolean hasInPlaceEditor(Object object, int column);
    public boolean hasLaunchedEditor(Object object, int column);
constant
    public void paintControl(PaintEvent event);
// Public Methods Overriding ExtendedTableTreeEditor
    public void dismiss();
// Protected Instance Methods
    protected void activate();
    protected void arrowDown();
    protected void arrowLeft();
    protected void arrowRight();
    protected void arrowUp();
    protected Composite createComposite();
    protected void editItem(TableItem tableItem, TableTreeItem tableTreeItem, int column);
Overrides:ExtendedTableTreeEditor

    protected void getGradients();
    protected Image getLeftGradient();
    protected Image getRightGradient();
    protected boolean isDown();
    protected void setDown(boolean isDown);
// Protected Instance Fields
    protected Control activeEditor;
    protected AdapterFactory adapterFactory;
    protected Composite canvas;
    protected int currentColumn;
    protected TableItem currentTableItem;
    protected TableTreeItem currentTableTreeItem;
    protected Object currentTableTreeItemData;
    protected boolean hasDropDown;
    protected boolean hasLaunched;
    protected boolean isDown;
    protected AdapterFactoryItemDelegator itemDelegator;
    protected KeyListener keyListener;
    protected Image leftGradient;
    protected PaintListener paintListener;
    protected Image rightGradient;
}

```

Hierarchy: Object → ControlEditor → TableTreeEditor → ExtendedTableTreeEditor(KeyListener(SWTEventListener(EventListener)), MouseListener(SWTEventListener), SelectionListener(SWTEventListener)) → AdapterFactoryTableTreeEditor

AdapterFactoryTreeEditor

org.eclipse.emf.edit.ui.celleditor

AdapterFactoryTreeEditor implements an SWT TreeEditor that delegates to an IUpdateableItemText adapter. The editItem() method uses the getUpdateableText() and setText() methods to get and set the specified TreeItem's data value, respectively.

```

public class AdapterFactoryTreeEditor extends ExtendedTreeEditor {
// Public Constructors
    public AdapterFactoryTreeEditor(Tree tree, AdapterFactory adapterFactory);
// Property Accessor Methods (by property name)
    public AdapterFactory getAdapterFactory();
    public void setAdapterFactory(AdapterFactory adapterFactory);
// Protected Instance Methods
    protected void editItem(TreeItem treeItem);
Overrides:ExtendedTreeEditor

```

```
// Protected Instance Fields
protected AdapterFactory adapterFactory;
protected TreeItem currentTreeItem;
}
```

Hierarchy: Object → ControlEditor → TreeEditor → ExtendedTreeEditor (SelectionListener (SWTEventListener (EventListener)), MouseListener (SWTEventListener), KeyListener (SWTEventListener)) → AdapterFactoryTreeEditor

FeatureEditorDialog

org.eclipse.emf.edit.ui.celleditor

Class FeatureEditorDialog is a dialog window for editing a multiplicity-many attribute or reference. It provides an editor that allows one to add and remove the values of the eStructuralFeature in eObject, given choiceOfValues to choose from. After okPressed() is called, the selected set of values is provided in the list returned from the getResult() method.

This class is instantiated by the createPropertyEditor() method in class PropertyDescriptor in the case of editing multivalued features.

```
public class FeatureEditorDialog extends Dialog {
// Public Constructors
    public FeatureEditorDialog(Shell parent, ILabelProvider labelProvider, EObject eObject, ↵
EStructuralFeature eStructuralFeature, String displayName, List choiceOfValues);
// Property Accessor Methods (by property name)
    public EList getResult();
// Public Methods Overriding Dialog
    public boolean close();
// Protected Instance Methods
    protected void configureShell(Shell shell);
Overrides:Dialog
    protected Control createDialogArea(Composite parent);
Overrides:Dialog
    protected void okPressed();
Overrides:Dialog
// Protected Instance Fields
    protected List choiceOfValues;
    protected IContentProvider contentProvider;
    protected String displayName;
    protected EObject eObject;
    protected EStructuralFeature eStructuralFeature;
    protected ILabelProvider labelProvider;
    protected EList result;
    protected ItemProvider values;
}
```

Hierarchy: Object → Window → Dialog → FeatureEditorDialog

The org.eclipse.emf.edit.ui.dnd Package

The org.eclipse.emf.edit.ui.dnd package provides support for drag and drop by delegation to a DragAndDropCommand. Like all command-based modification in EMF.Edit, drag-and-drop operations are undoable.

EditingDomainViewerDropAdapter

org.eclipse.emf.edit.ui.dnd

`EditingDomainViewerDropAdapter` implements an SWT `DropTargetListener` that is designed to turn a drag-and-drop operation into a `Command` on the model objects of an `EditingDomain`. The underlying command is created by calling `DragAndDropCommand.create()`. It is designed to do early data transfer so the enablement and feedback of the drag-and-drop interaction can intimately depend on the state of the model objects involved.

The base implementation of this class should be sufficient for most applications. Any change in behavior is typically accomplished by overriding the `createDragAndDropCommand()` method in class `ItemProviderAdapter` to return a derived implementation of class `DragAndDropCommand`.

These adapters are typically registered with a viewer like this:

```
viewer.addDropSupport
(DND.DROP_COPY | DND.DROP_MOVE | DND.DROP_LINK,
new Transfer [] { LocalTransfer.getInstance() },
EditingDomainViewerDropAdapter(viewer));
```

This implementation prefers to use a `LocalTransfer`, which short-circuits the transfer process for simple transfers within the workbench, although the `getDragSource()` method can be overridden to change this behavior. The implementation also only handles an `IStructuredSelection`, but the `extractDragSource()` method can be overridden to change this. The `setHoverThreshold()` method can be called to set the amount of time, in milliseconds, to hover over an item before the `hover()` method is called; the default is 1,500 milliseconds.

```
public class EditingDomainViewerDropAdapter extends DropTargetAdapter {
// Public Constructors
    public EditingDomainViewerDropAdapter(EditingDomain domain, Viewer viewer);
// Protected Inner Classes
    protected static class DragAndDropCommandInformation;
// Protected Class Methods
    protected static float getLocation(DropTargetEvent event);
// Property Accessor Methods (by property name)
    public int getHoverThreshold();
    public void setHoverThreshold(int hoverThreshold);
// Public Methods Overriding DropTargetAdapter
    public void dragEnter(DropTargetEvent event);
    public void dragLeave(DropTargetEvent event);
    public void dragOperationChanged(DropTargetEvent event);
    public void dragOver(DropTargetEvent event);
    public void drop(DropTargetEvent event);
    public void dropAccept(DropTargetEvent event);                                empty
// Protected Instance Methods
    protected Collection extractDragSource(Object object);
    protected Collection getDragSource(DropTargetEvent event);
    protected void helper(DropTargetEvent event);
    protected void hover(Object target);
    protected boolean scrollIfNeeded(DropTargetEvent event);
// Protected Instance Fields
    protected Command command;
    protected Object commandTarget;
    protected EditingDomain domain;
    protected EditingDomainViewerDropAdapter.DragAndDropCommandInformation
dragAndDropCommandInformation;
    protected long hoverStart;
    protected int hoverThreshold;
    protected int originalOperation;
```

```

protected Widget previousItem;
protected Collection source;
protected Viewer viewer;
}

```

Hierarchy: Object → DropTargetAdapter(DropTargetListener(SWTEventListener(EventListener))) → EditingDomainViewerDropAdapter

EditingDomainViewerDropAdapter.DragAndDropCommandInformation

org.eclipse.emf.edit.ui.dnd

This class is used by EditingDomainViewerDropAdapter to keep track of the information that was last used to create or validate its associated DragAndDropCommand. The createCommand() method calls DragAndDropCommand.create() with the current information.

```

protected static class EditingDomainViewerDropAdapter.DragAndDropCommandInformation {
    // Public Constructors
    public DragAndDropCommandInformation(EditingDomain domain, Object target, float location, ←
    int operations, int operation, Collection source);
    // Public Instance Methods
    public Command createCommand();
    // Protected Instance Fields
    protected EditingDomain domain;
    protected float location;
    protected int operation;
    protected int operations;
    protected Collection source;
    protected Object target;
}

```

LocalTransfer

org.eclipse.emf.edit.ui.dnd

Class LocalTransfer is a derived implementation of SWT's ByteArrayTransfer that short-circuits the transfer process so that a local transfer does not serialize the object and can return the original object instead of a copy. It uses the Singleton design pattern, with its single instance available from the static getInstance() method.

The javaToNative() method records the specified object in its object field, but as an added guard saves the current time in the startTime field and serializes it to the TransferData by calling super.javaToNative(). When nativeToJava() is later called, it simply returns the object field, but only after verifying that the TransferData contains the correct recorded time.

```

public class LocalTransfer extends ByteArrayTransfer {
    // Protected Constructors
    protected LocalTransfer();                                         empty
    // Protected Constants
    protected static final int TYPE_ID;
    protected static final String TYPE_NAME;                           ="local-transfer-
format"
    // Public Class Methods
    public static LocalTransfer getInstance();
    // Property Accessor Methods (by property name)
    public String[] getTypeNames();
    Overrides:Transfer
    // Public Methods Overriding ByteArrayTransfer
    public void javaToNative(Object object, TransferData transferData);
    public Object nativeToJava(TransferData transferData);
}

```

```

// Protected Instance Methods
protected int[] getTypesIds();
Overrides:Transfer
// Protected Class Fields
protected static LocalTransfer instance;
// Protected Instance Fields
protected Object object;
protected long startTime;
}

```

Hierarchy: Object → Transfer → ByteArrayTransfer → LocalTransfer

ViewerDragAdapter

org.eclipse.emf.edit.ui.dnd

Class `ViewerDragAdapter` is an implementation of an SWT `DragSourceListener` that records the selection in effect at the start of a drag-and-drop interaction, so that an interaction within a single view can be optimized. It's typically used in conjunction with class `EditingDomainViewerDropAdapter` and class `LocalTransfer`, and registered with a viewer like this:

```

viewer.addDragSupport
(DND.DROP_COPY | DND.DROP_MOVE | DND.DROP_LINK,
 new Transfer [] { LocalTransfer.getInstance() },
ViewerDragAdapter(viewer));

```

This allows a drag operation to be initiated from the viewer, whereby the viewer's selection will be transferred directly to the drop target.

```

public class ViewerDragAdapter implements DragSourceListener {
// Public Constructors
    public ViewerDragAdapter(Viewer viewer);
// Methods Implementing DragSourceListener
    public void dragFinished(DragSourceEvent event);
    public void dragSetData(DragSourceEvent event);
    public void dragStart(DragSourceEvent event);
// Protected Instance Fields
    protected ISelection selection;
    protected Viewer viewer;
}

```

Hierarchy: Object → ViewerDragAdapter(DragSourceListener(SWTEventListener(EventListener)))

The org.eclipse.emf.edit.ui.provider Package

The `org.eclipse.emf.edit.ui.provider` package provides implementations of JFace content and label providers, and property source and descriptors that delegate to EMF.Edit item providers. It also provides an image registry class and a convenience class for optimized refreshing of standard JFace viewers, in response to EMF notifications.

AdapterFactoryContentProvider

org.eclipse.emf.edit.ui.provider

Class AdapterFactoryContentProvider wraps an AdapterFactory and delegates JFace provider interfaces to corresponding adapter-implemented item provider interfaces. All method calls to the various structured content provider interfaces are delegated to corresponding interfaces on the adapters: IStructuredContentProvider methods are delegated to IStructuredItemContentProvider, and ITreeContentProvider methods are delegated to ITreeItemContentProvider. IPropertySourceProvider returns a property source that delegates to IItemPropertySource.

An AdapterFactoryContentProvider also registers as a listener of its wrapped adapter factory, if it is an IChangeNotifier. It responds to notifications by calling NotifyChangedToViewerRefresh.handleNotifyChanged() to refresh its associated viewer.

```
public class AdapterFactoryContentProvider implements ITreeContentProvider, ↪
    IPropertySourceProvider, INotifyChangedListener {
    // Public Constructors
    public AdapterFactoryContentProvider(AdapterFactory adapterFactory);
    // Property Accessor Methods (by property name)
    public AdapterFactory getAdapterFactory();
    public void setAdapterFactory(AdapterFactory adapterFactory);
    // Methods Implementing ITreeContentProvider
    public Object[] getChildren(Object object);
    public Object getParent(Object object);
    public boolean hasChildren(Object object);
    // Methods Implementing IPropertySourceProvider
    public IPropertySource getPropertySource(Object object);
    // Methods Implementing INotifyChangedListener
    public void notifyChanged(Notification notification);
    // Methods Implementing IStructuredContentProvider
    public Object[] getElements(Object object);
    // Methods Implementing IContentProvider
    public void dispose();
    public void inputChanged(Viewer viewer, Object oldInput, Object newInput);
    // Protected Instance Methods
    protected IPropertySource createPropertySource(Object object, IItemPropertySource ↪
        itemPropertySource);
    // Protected Instance Fields
    protected AdapterFactory adapterFactory;
    protected Viewer viewer;
}
```

Hierarchy: Object → AdapterFactoryContentProvider(ITreeContentProvider(StructuredContentProvider(IContentProvider)), IPropertySourceProvider, INotifyChangedListener)

AdapterFactoryLabelProvider

org.eclipse.emf.edit.ui.provider

Class AdapterFactoryLabelProvider wraps an AdapterFactory and delegates JFace label provider interfaces to corresponding adapter-implemented item provider interfaces. All method calls to the label provider interfaces are delegated to corresponding interfaces on the adapters: ILabelProvider methods are delegated to IIItemLabelProvider and ITableLabelProvider methods are delegated to ITableItemLabelProvider, if supported by the adapter factory—otherwise, the columnIndex argument is ignored and they are also delegated to the IIItemLabelProvider interface.

An `AdapterFactoryLabelProvider` also registers as a listener of its wrapped adapter factory, if it is an `IChangeNotifier`. It also maintains a list of its own `ILabelProviderListeners`, but it never sends notifications to them by default, although subclasses can override this.

```
public class AdapterFactoryLabelProvider implements ILabelProvider, ITableLabelProvider, +  
INotifyChangedListener {  
    // Public Constructors  
    public AdapterFactoryLabelProvider(AdapterFactory adapterFactory);  
    // Property Accessor Methods (by property name)  
    public AdapterFactory getAdapterFactory();  
    public void setAdapterFactory(AdapterFactory adapterFactory);  
    // Public Instance Methods  
    public void fireLabelProviderChanged();  
    // Methods Implementing ILabelProvider  
    public Image getImage(Object object);  
    public String getText(Object object);  
    // Methods Implementing ITableLabelProvider  
    public Image getColumnImage(Object object, int columnIndex);  
    public String getColumnText(Object object, int columnIndex);  
    // Methods Implementing INotifyChangedListener  
    public void notifyChanged(Notification notification);  
    empty  
    // Methods Implementing IBaseLabelProvider  
    public void addListener(ILabelProviderListener listener);  
    public void dispose();  
    public boolean isLabelProperty(Object object, String id);  
    constant  
    public void removeListener(ILabelProviderListener listener);  
    // Protected Instance Methods  
    protected Image getDefaultImage(Object object);  
    protected Image getImageFromObject(Object object);  
    // Protected Instance Fields  
    protected AdapterFactory adapterFactory;  
    protected Collection labelProviderListeners;  
}
```

Hierarchy: Object → `AdapterFactoryLabelProvider` (`ILabelProvider` (`IBaseLabelProvider`), `ITableLabelProvider` (`IBaseLabelProvider`)), `INotifyChangedListener`)

ExtendedImageRegistry

org.eclipse.emf.edit.ui.provider

An `ExtendedImageRegistry` is, like JFace's `ImageRegistry`, a registry and cache for images. EMF.Edit's `ExtendedImageRegistry` however, allows images to be accessed more generally than just using a string key. The object (`java.lang.Object`) passed to the `getImage()` or `getImageDescriptor()` method can be an `ImageDescriptor`, a URL, a `ComposedImage`, or even an `Image` itself. The two methods return the corresponding image and image descriptor, respectively.

This class uses the Singleton design pattern, with its single instance available as the constant `INSTANCE` field or from the static `getInstance()` method.

```
public class ExtendedImageRegistry {  
    // Public Constructors  
    public ExtendedImageRegistry();  
    public ExtendedImageRegistry(Display display);  
    // Public Constants  
    public static final ExtendedImageRegistry INSTANCE;  
    // Public Class Methods  
    public static ExtendedImageRegistry getInstance();  
    // Public Instance Methods
```

```

public Image getImage(Object object);
public ImageDescriptor getImageDescriptor(Object object);
// Protected Instance Methods
protected void handleDisplayDispose();
protected void hookDisplayDispose(Display display);
// Protected Class Fields
protected static String createChildURLPrefix;
protected static String itemURLPrefix;
protected static String resourceURLPrefix;
// Protected Instance Fields
protected HashMap table;
}

```

NotifyChangedToViewerRefresh

org.eclipse.emf.edit.ui.provider

Class `NotifyChangedToViewerRefresh` implements optimized notification refresh for all the standard JFace viewer classes. The static `handleNotifyChanged()` method constructs an instance and invokes the `refresh()` method, which in turn delegates to one of the viewer-specific `refresh()` methods. Each of these methods performs the refresh optimally for the type of viewer and the notification `eventType`.

```

public class NotifyChangedToViewerRefresh {
// Public Constructors
public NotifyChangedToViewerRefresh();
empty
// Public Class Methods
public static void handleNotifyChanged(Viewer viewer, Object object, int eventType, Object ←
feature, Object oldValue, Object newValue, int index);
// Public Instance Methods
public void refresh(Viewer viewer, Object object, int eventType, Object feature, Object ←
oldValue, Object newValue, int index);
public void refreshAbstractTreeViewer(AbstractTreeViewer viewer, Object object, int ←
eventType, Object feature, Object oldValue, Object newValue, int index);
public void refreshListViewer(ListViewer viewer, Object object, int eventType, Object ←
feature, Object oldValue, Object newValue, int index);
public void refreshStructuredViewer(StructuredViewer viewer, Object object, int eventType, ←
Object feature, Object oldValue, Object newValue, int index);
public void refreshTableTreeViewer(TableTreeViewer viewer, Object object, int eventType, ←
Object feature, Object oldValue, Object newValue, int index);
public void refreshTableViewer(TableViewer viewer, Object object, int eventType, Object ←
feature, Object oldValue, Object newValue, int index);
public void refreshTreeViewer(TreeViewer viewer, Object object, int eventType, Object ←
feature, Object oldValue, Object newValue, int index);
public void refreshViewer(Viewer viewer, Object object, int eventType, Object feature, ←
Object oldValue, Object newValue, int index);
}

```

PropertyDescriptor

org.eclipse.emf.edit.ui.provider

Class `PropertyDescriptor` implements an Eclipse `IPropertyDescriptor` by delegating to an EMF.Edit `IIItemPropertyDescriptor`. It encapsulates an `IIItemPropertyDescriptor`, along with the object for which it is a property.

The `createPropertyEditor()` method returns the cell editor that will be used to edit the value of the property. It determines the kind of cell editor to use based on the type of the structural feature returned by `itemPropertyDescriptor.getFeature()`. In the case where a `ComboBoxCellEditor` is used, it populates the combo box with the values returned from `itemPropertyDescriptor.getChoiceOfValues()`.

```

public class PropertyDescriptor implements IPropertyDescriptor {
    // Public Constructors
    public PropertyDescriptor(Object object, IIItemPropertyDescriptor itemPropertyDescriptor);
    // Protected Constants
    protected static final EcorePackage ecorePackage;
    // Public Inner Classes
    public static class EDataTypeCellEditor;
    # public static class FloatCellEditor;
    # public static class IntegerCellEditor;
    // Property Accessor Methods (by property name)
    public String getCategory();
    Implements:IPropertyDescriptor
    public String getDescription();
    Implements:IPropertyDescriptor
    public String getDisplayName();
    Implements:IPropertyDescriptor
    public String[] getFilterFlags();
    Implements:IPropertyDescriptor
    public Object getHelpContextIds();
    Implements:IPropertyDescriptor
    public Object getId();
    Implements:IPropertyDescriptor
    public ILabelProvider getLabelProvider();
    Implements:IPropertyDescriptor
    // Methods Implementing IPropertyDescriptor
    public CellEditor createPropertyEditor(Composite composite);
    public boolean isCompatibleWith(IPropertyDescriptor anotherProperty);
    constant
    // Protected Instance Fields
    protected IIItemPropertyDescriptor itemPropertyDescriptor;
    protected Object object;
}

```

Hierarchy: Object → PropertyDescriptor (IPropertyDescriptor)

PropertyDescriptor.EDataTypeCellEditor

org.eclipse.emf.edit.ui.provider

A `PropertyDescriptor.EDataTypeCellEditor` is used to edit the values of single-valued attributes that have neither boolean nor enumeration type. It uses the attribute's data type `createFromString()` and `convertToString()` methods to get and set the editor value, respectively.

```

public static class PropertyDescriptor.EDataTypeCellEditor extends TextCellEditor {
    // Public Constructors
    public EDataTypeCellEditor(EDataType eDataType, Composite composite);
    // Public Methods Overriding TextCellEditor
    public Object doGetValue();
    public void doSetValue(Object value);
    // Protected Instance Fields
    protected EDataType eDataType;
}

```

Hierarchy: Object → CellEditor → TextCellEditor → PropertyDescriptor.EDataTypeCellEditor

PropertySource

`org.eclipse.emf.edit.ui.provider`

Class `PropertySource` implements an Eclipse `IPropertySource` by delegating to an EMF `Edit.IItemPropertySource`, usually the item provider adapter of the object for which it's the source. It encapsulates an `IItemPropertySource`, along with the object for which it is a property source.

The `createPropertyDescriptor()` method, which is called by `getPropertyDescriptors()` for each `IItemPropertyDescriptor` returned by `itemPropertySource.getPropertyDescriptors()`, creates and returns an instance of class `PropertyDescriptor`. The four `IPropertySource` methods—`getPropertyValue()`, `isPropertySet()`, `resetPropertyValue()`, and `setPropertyValue()`—are ultimately delegated to the `IItemPropertyDescriptor`, via `itemPropertySource.getPropertyDescriptor()`, for the specified `propertyId`.

```
public class PropertySource implements IPropertySource {
    // Public Constructors
    public PropertySource(Object object, IItemPropertySource itemPropertySource);
    // Property Accessor Methods (by property name)
    public Object getEditableValue();
    Implements:IPropertySource
    public IPropertyDescriptor[] getPropertyDescriptors();
    Implements:IPropertySource
    // Methods Implementing IPropertySource
    public Object getPropertyValue(Object propertyId);
    public boolean isPropertySet(Object propertyId);
    public void resetPropertyValue(Object propertyId);
    public void setPropertyValue(Object propertyId, Object value);
    // Protected Instance Methods
    protected IPropertyDescriptor createPropertyDescriptor(IItemPropertyDescriptor ←
    itemPropertyDescriptor);
    // Protected Instance Fields
    protected IItemPropertySource itemPropertySource;
    protected Object object;
}
```

Hierarchy: Object → PropertySource (IPropertySource)

Appendix A. UML Notation

UML class diagrams are the most concise way of describing EMF models. We use UML diagrams throughout the book to depict examples and to illustrate various EMF concepts. EMF uses only a subset of the complete UML notation along with some conventions of its own. Here we provide a quick summary of the subset UML notation used in the book. For a complete description of the mapping of UML to EMF, refer to Chapter 8.

Classes and Interfaces

Figure A.1 shows the UML notation for concrete and abstract classes and for interfaces. All three are denoted by a box containing three compartments, with the class or interface name in the top one. Concrete class names are displayed in regular font and the names of abstract classes are *italicized*. An interface name is also in regular font, but preceded by the stereotype `<<interface>>`.

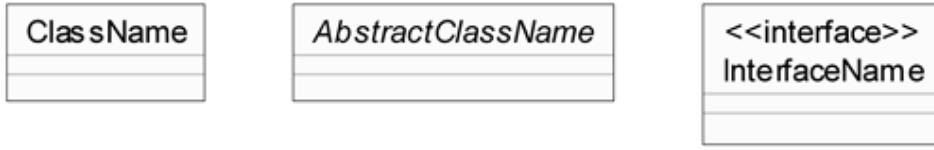


Figure A.1. Classes and interfaces.

A UML class, abstract or concrete, is used to represent an EMF class, not just a Java class. An EMF class maps to both a Java interface and a corresponding implementation class as described in Chapter 9. A UML/EMF interface, on the other hand, maps to just a Java interface with no corresponding implementation class.

The second and third compartments of a UML class or interface include its attributes and operations, respectively. Figure A.2 shows a class containing three attributes and two operations. Attributes represent the data elements that comprise the class.

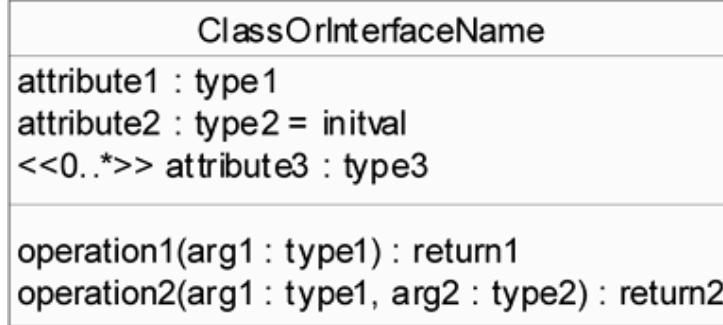


Figure A.2. Attributes and operations.

In a UML class, an attribute appears as a name followed by a colon (:) and then its type. The type of an attribute can be any Java primitive type (`boolean`, `int`, `long`, and so on), a basic Java data type from the `java.lang` package (`Boolean`, `Integer`, `Long`, `String`, and so on), or the name of a user-defined enumeration or data type as we'll describe next. An attribute declaration (like `attribute2`, in our example) can also optionally be followed by an equal sign (=) and then an initial value suitable for the type of the attribute. An attribute can also be preceded by a stereotype that indicates that the attribute is not a single value, but instead a list of values. For example, the stereotype on `attribute3` indicates an unlimited multiplicity. Multiplicity-many associations are much more common than multiplicity-many attributes. We'll describe multiplicities further in "Class Relationships" on page 647.

The operations in the bottom compartment of the UML class are declared in a similar way to attributes, only with the addition of a list of zero or more comma-separated parameters inside parentheses following the operation name and before the return type. The return type and parameter types of an operation can be any of the types that an attribute can be, described previously, but also can be the name of any class in the model.

Enumerations and Data Types

As stated earlier, the type of an attribute can be an enumeration or user-defined data type. Figure A.3 shows the UML notation for defining such types. Both are represented as UML classes, but with only two compartments instead of three.¹ As with classes and interfaces, the top compartment contains the name of the type, but in these cases, preceded by the stereotype `<<enumeration>>` or `<<datatype>>`.



Figure A.3. Enumerations and data types.

In an enumeration declaration, the second compartment (that is, the attribute compartment of the UML class) contains the set of names of the enumeration's literals. A literal name can optionally be followed by an equal sign (=) and then its corresponding integer value. Literals without an explicit supplied value are automatically assigned sequential values starting from zero.

The second compartment of a data type contains a single entry (attribute) that is the fully qualified name of an arbitrary Java class that represents the type. The name must also be preceded by the stereotype `<<javaclass>>`.

Class Relationships

An inheritance relationship in UML is represented by a line between the classes with a triangle pointing to the superclass. Figure A.4 shows two examples of a subclass (`ClassB`), one involving single inheritance, the other multiple.

¹By convention, we suppress the third (operation) compartment because it would always be empty anyway.

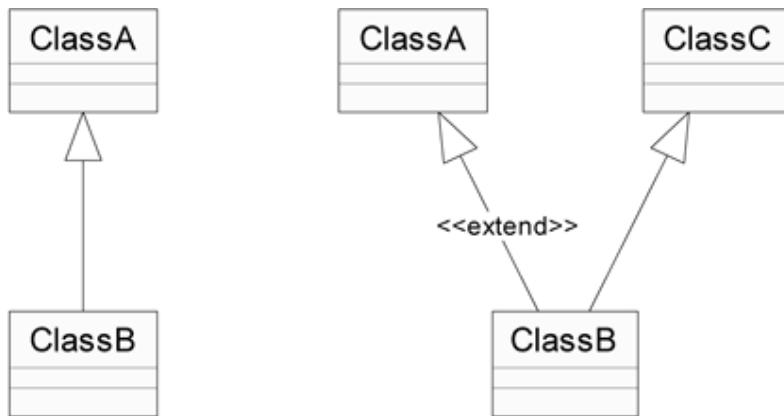


Figure A.4. Class inheritance.

Although EMF supports multiple inheritance, Java only supports single-implementation inheritance. Therefore, in the case of multiple inheritance, the `<<extend>>` stereotype is used to indicate which of the multiple superclasses is to be used as the base class for the corresponding Java implementation class. The others only involve interface inheritance in Java and their implementations are replicated in the subclass.

A one-way association, or reference, between two classes is depicted in UML by a line with an arrowhead on the target end. Figure A.5 shows three examples of one-way references from a source class, **ClassA**, to a target **ClassB**. At the target end appears the name of the reference (often referred to as the role name), and its multiplicity. The multiplicity is typically of the form `lower..upper`, where `lower` represents the lower bound and `upper` represents the upper bound, although the single value `1` is often used as a short form for `1..1`. An upper bound value of `*` represents unlimited multiplicity.

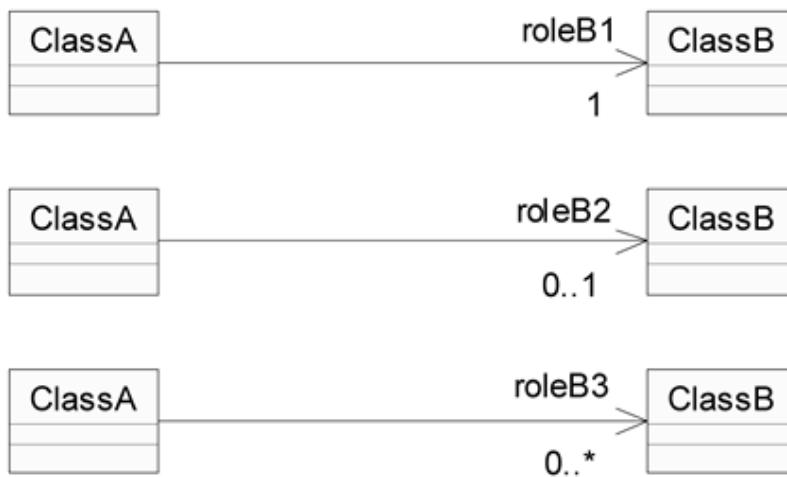


Figure A.5. One-way associations.

The difference between **roleB1** and **roleB2** in Figure A.5 is that **roleB1** (whose lower bound is implicitly `1`) is declared to be required, whereas **roleB2** is declared as optional; it might or might not be set in a valid model instance.

Associations can also be bidirectional. A bidirectional association is the same as two one-way associations in opposite directions, only the inverse is automatically updated when either end is set. As shown in Figure A.6, a bidirectional association is represented in UML by a line between the classes with no arrowhead on either end.



Figure A.6. Bidirectional associations.

The last, and arguably the most important, kind of association that EMF is concerned with is referred to as containment or by-value aggregation. As shown in Figure A.7, a containment reference is indicated by a black diamond on the source (that is, the container) end of the association. Containment associations imply physical location and ownership. Instances of **ClassB** in our example are owned by and persisted with an instance of **ClassA**.



Figure A.7. Containment associations.

Just like other associations, a containment association can be one-way or bidirectional.² The multiplicity of the container end of a containment association need not be specified. It is implicitly, and can only be, 1; an object can only exist in one container.

²In fact, all containment references are implicitly bidirectional in EMF; any EMF object can generically access its container. The benefit of a bidirectional containment association is that its access can be type-safe, and that notification will be provided for the contained object when its container changes.

Appendix B. Summary of Example Models

The code fragments and examples used throughout this book are based on five main models. These models, SimplePO, PrimerPO, ExtendedPO1, ExtendedPO2, and ExtendedPO3, define the data for purchase order management applications, each one progressively more complicated than the last. The models are shown here, along with a brief description of how each builds on the one before it. The first two models are provided in four forms: UML, XML Schema, annotated Java interfaces, and Ecore/XMI. For the more complicated extended models, we only show the most concise form: UML. Note that all of these models, along with the rest of the example code used throughout this book, can also be downloaded from the Addison-Wesley Web site at <http://www.aw.com/budinsky>.

SimplePO

This is the first and simplest example in the book. It includes two classes, **PurchaseOrder** and **Item**, with a containment association between them. The SimplePO model is used in Chapter 2 to introduce EMF. Elements of it are also used to illustrate mapping patterns in Chapter 7. The first example in Chapter 14 (Section 14.1) is also based on it. See Figure B.1.

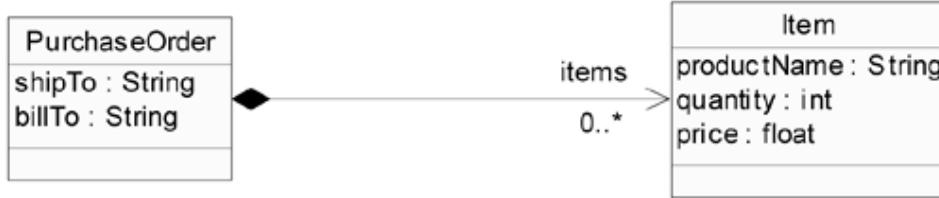


Figure B.1. The SimplePO model.

In the form of an XML Schema, the SimplePO model looks like this:

SimplePO.xsd:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.com/SimplePO"
  xmlns:PO="http://www.example.com/SimplePO">
  <xsd:complexType name="PurchaseOrder">
    <xsd:sequence>
      <xsd:element name="shipTo" type="xsd:string"/>
      <xsd:element name="billTo" type="xsd:string"/>
      <xsd:element name="items" type="PO:Item"
        minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Item">
    <xsd:sequence>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="quantity" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>

```

```

<xsd:element name="price" type="xsd:float"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>

```

Using annotated Java, the SimplePO model can be described with two Java source files: *PurchaseOrder.java* and *Item.java*.

PurchaseOrder.java:

```

package com.example.po;

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

/**
 * @model
 */
public interface PurchaseOrder
{
    /**
     * @model
     */
    String getShipTo();

    /**
     * @model
     */
    String getBillTo();

    /**
     * @model type="Item" containment="true"
     */
    EList getItems();
}

```

Item.java:

```

package com.example.po;

/**
 * @model
 */
public interface Item
{
    /**
     * @model
     */
    String getProductName();

    /**
     * @model
     */
    int getQuantity();

    /**
     * @model
     */
    float getPrice();
}

```

The Ecore model for SimplePO looks like this when serialized using XML:

SimplePO.ecore:

```
<?xml version="1.0" encoding="ASCII"?>
<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="http://com/example/po.ecore"
    nsPrefix="com.example.po">
    <eClassifiers xsi:type="ecore:EClass" name="PurchaseOrder">
        <eReferences name="items"
            eType="#/Item" upperBound="-1" containment="true"/>
        <eAttributes name="shipTo"
            eType="ecore:EDataType
                http://www.eclipse.org/emf/2002/Ecore//EString"/>
        <eAttributes name="billTo"
            eType="ecore:EDataType
                http://www.eclipse.org/emf/2002/Ecore//EString"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="Item">
        <eAttributes name="productName"
            eType="ecore:EDataType
                http://www.eclipse.org/emf/2002/Ecore//EString"/>
        <eAttributes name="quantity"
            eType="ecore:EDataType
                http://www.eclipse.org/emf/2002/Ecore//EInt"/>
        <eAttributes name="price"
            eType="ecore:EDataType
                http://www.eclipse.org/emf/2002/Ecore//EFloat"/>
    </eClassifiers>
</ecore:EPackage>
```

PrimerPO

The PrimerPO model builds on the SimplePO model, making it closely resemble the purchase order example used in the W3C Recommendation for XML Schema Part 0: Primer [2]. It adds some new features to the **PurchaseOrder** and **Item** classes, a new class **USAddress**, and two data types, **Date** and **SKU**. The PrimerPO model is introduced in Chapter 4. It's also used as the main example for illustrating code generator patterns in Chapters 9 and 10. The second example in Chapter 14 (Section 14.2.1) is also based on it. See Figure B.2.

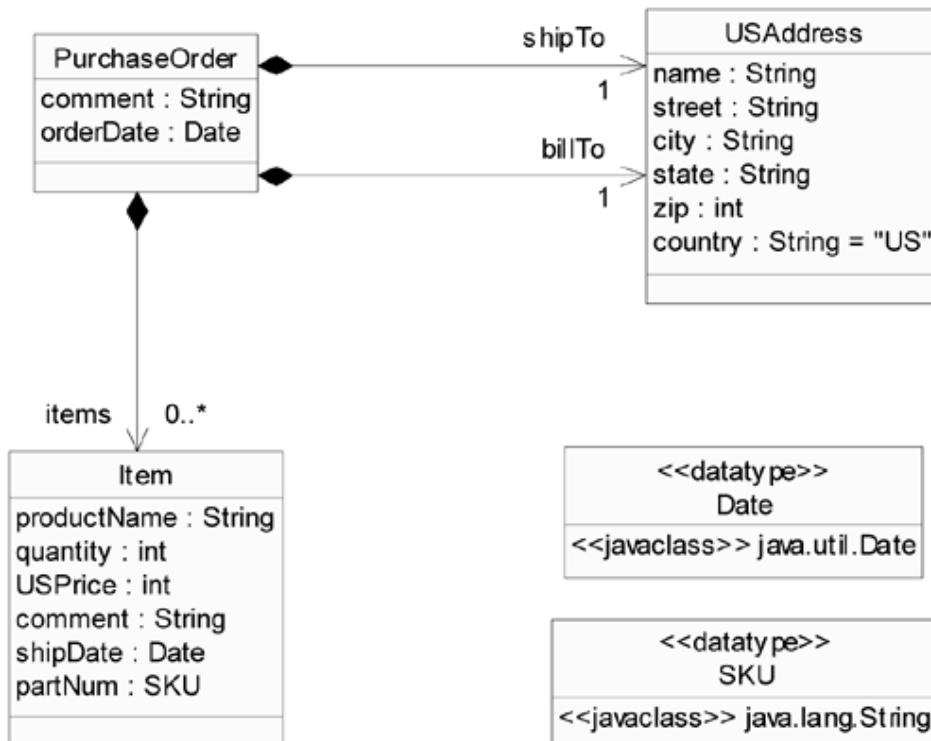


Figure B.2. The PrimerPO model.

In the form of an XML Schema, the PrimerPO model looks like this:

PrimerPO.xsd:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.com/PrimerPO"
  xmlns:ppo="http://www.example.com/PrimerPO">

  <xsd:element name="order" type="ppo:PurchaseOrder"/>

  <xsd:element name="comment" type="xsd:string"/>

  <xsd:complexType name="PurchaseOrder">
    <xsd:sequence>
      <xsd:element name="shipTo" minOccurs="0" type="ppo:USAddress"/>
      <xsd:element name="billTo" type="ppo:USAddress"/>
      <xsd:element ref="ppo:comment" minOccurs="0"/>
      <xsd:element name="items" type="ppo:Item"
        minOccurs="0" maxOccurs="unbounded"/>
    </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="Item">
  <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="ppo:comment" minOccurs="0"/>
    <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="partNum" type="ppo:SKU" use="required"/>
</xsd:complexType>

<xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-[A-Z]{2}" />
  </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

Using annotated Java, the PrimerPO model can be described with three Java source files: *PurchaseOrder.java*, *Item.java*, and *USAddress.java*.

PurchaseOrder.java:

```
package com.example.ppo;

import java.util.Date;
import java.util.List;

/**
 * @model
 */
public interface PurchaseOrder
{
  /**
   * @model
   */
  String getComment();

  /**
   * @model
   */
  Date getOrderDate();

  /**
   * @model type="Item" containment="true"
   */
  List getItems();

  /**
   * @model containment="true" required="true"
   */
  USAddress getBillTo();

  /**
   * @model containment="true" required="true"
   */
  USAddress getShipTo();
}
```

Item.java:

```
package com.example.ppo;

import java.util.Date;

/**
 * @model
 */
public interface Item
{
    /**
     * @model
     */
    String getProductName();

    /**
     * @model
     */
    int getQuantity();

    /**
     * @model
     */
    int getUSPrice();

    /**
     * @model
     */
    String getComment();

    /**
     * @model
     */
    Date getShipDate();

    /**
     * @model dataType="SKU"
     */
    String getPartNum();
}
```

USAddress.java:

```
package com.example.ppo;

/**
 * @model
 */
public interface USAddress
{
    /**
     * @model
     */
    String getName();

    /**
     * @model
     */
    String getStreet();

    /**
     * @model
     */
    String getCity();

    /**
     * @model
     */
}
```

```

String getState();

/**
 * @model
 */
int getZip();

/**
 * @model default="US" changeable="false"
 */
String getCountry();
}

```

The Ecore model for PrimerPO looks like this when serialized using XML:

PrimerPO.ecore:

```

<?xml version="1.0" encoding="ASCII"?>
<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="ppo" nsURI="http:///com/example/ppo.ecore"
    nsPrefix="com.example.ppo">

<eClassifiers xsi:type="ecore:EClass" name="Item">
    <eAttributes name="productName"
        eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#/EString"/>
    <eAttributes name="quantity"
        eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#/EInt"/>
    <eAttributes name="USPrice"
        eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#/EInt"/>
    <eAttributes name="comment"
        eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#/EString"/>
    <eAttributes name="shipDate" eType="#//Date"/>
    <eAttributes name="partNum" eType="#//SKU"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="USAddress">
    <eAttributes name="name"
        eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#/EString"/>
    <eAttributes name="street"
        eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#/EString"/>
    <eAttributes name="city"
        eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#/EString"/>
    <eAttributes name="state"
        eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#/EString"/>
    <eAttributes name="zip"
        eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#/EInt"/>
    <eAttributes name="country"
        eType="ecore:EDataType
            http://www.eclipse.org/emf/2002/Ecore#/EString"
            changeable="false" defaultValueLiteral="US"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EDataType" name="SKU"
    instanceClassName="java.lang.String"/>
<eClassifiers xsi:type="ecore:EDataType" name="Date"
    instanceClassName="java.util.Date"/>
<eClassifiers xsi:type="ecore:EClass" name="PurchaseOrder">
    <eReferences name="items"
        eType="#//Item" upperBound="-1" containment="true"/>
    <eReferences name="billTo"
        eType="#//USAddress" lowerBound="1" containment="true"/>

```

```

<eReferences name="shipTo"
  eType="#//USAddress" lowerBound="1" containment="true"/>
<eAttributes name="comment"
  eType="ecore:EDataType
  http://www.eclipse.org/emf/2002/Ecore//EString"/>
<eAttributes name="orderDate" eType="#//Date"/>
</eClassifiers>
</ecore:EPackage>

```

ExtendedPO1

This model extends PrimerPO, adding class **Customer** and a new container class for purchase orders and customers, **Supplier**. It also adds a new one-way reference, **previousOrder**, and an enumeration type **OrderStatus**. It's the last of the example models that can be generated and run without any hand modification. Elements of the ExtendedPO1 model are used for some of the examples in Chapters 7 and 9. The third and fourth examples in Chapter 14 (Sections 14.2.2 and 14.2.3) are also based on it. See Figure B.3.

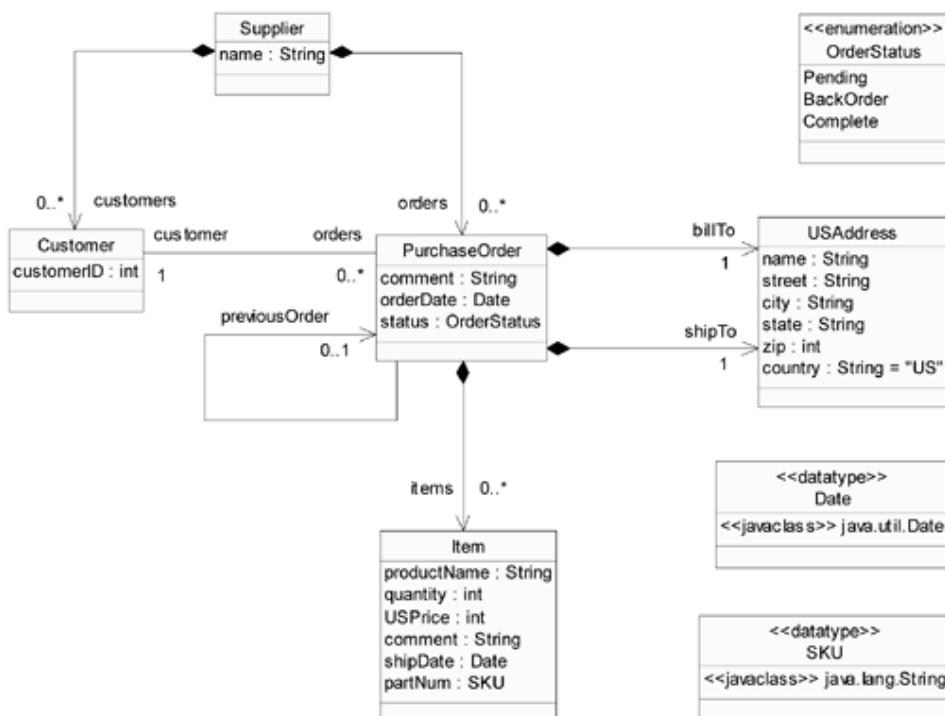


Figure B.3. The ExtendedPO1 model.

ExtendedPO2

This model builds on ExtendedPO1, adding an abstract class, **Address**, which **USAddress** and another new class, **GlobalAddress**, extend from. A second base class, **GlobalLocation**, of class **GlobalAddress** is another trivial class in the ExtendedPO2 model, whose only purpose is to serve as an example of the code generation pattern for multiple inheritance, described in Chapter 9. Two new volatile references, **pendingOrders** and **shippedOrders**, are added to class **Supplier**, and the volatile **totalAmount** attribute is added to class **PurchaseOrder**. The ExtendedPO2 model is introduced in Chapter 12. Some of its elements are also used to illustrate code generation patterns in Chapter 9 and for most of the examples in Chapter 13. The ExtendedPO2 model in UML is shown in Figure B.4.

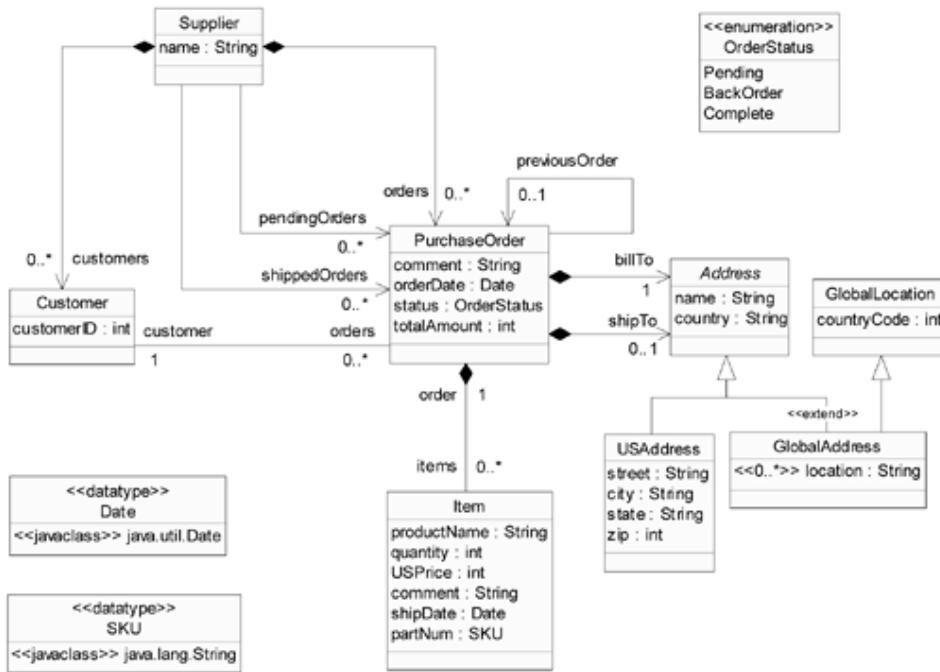


Figure B.4. The ExtendedPO2 model.

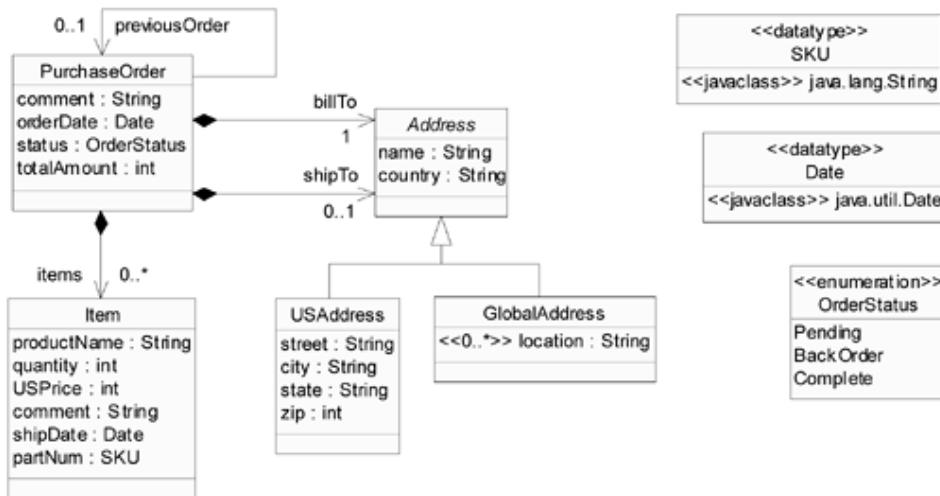
Some of the features in the ExtendedPO2 model have nondefault values for certain non-UML Ecore properties. They are summarized in Table B.1.

Table B.1. ExtendedPO2 Model Properties

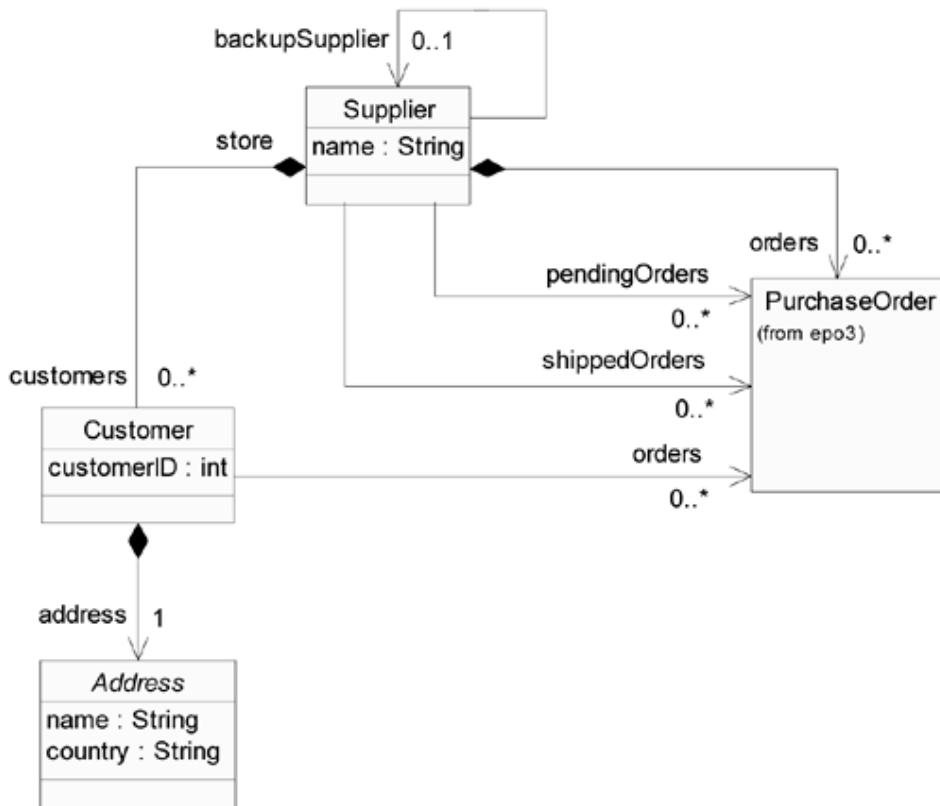
Feature	volatile	transient	changeable	resolveProxies
Supplier.pendingOrders	true	true	false	false
Supplier.shippedOrders	true	true	false	false
PurchaseOrder.totalAmount	true	true	false	
PurchaseOrder.previousOrder				false
PurchaseOrder.customer				false
Customer.orders				false

ExtendedPO3

The ExtendedPO3 model is essentially the same model as ExtendedPO2, only split into two separate packages, **epo3** and **supplier**. Like ExtendedPO2, it is also introduced in Chapter 12. In addition, the ExtendedPO3 model is used as the primary example for describing mappings to Ecore in Chapters 6 and 8. The **epo3** package in UML is shown in Figure B.5.

Figure B.5. The **epo3** package of the ExtendedPO3 model.

The rest of the ExtendedPO3 model is in the **supplier** package. It looks like Figure B.6 in UML.

Figure B.6. The **supplier** package of the ExtendedPO3 model.

Some of the features in the ExtendedPO3 model have nondefault values for certain non-UML Ecore properties. They are summarized in Table B.2.

Table B.2. ExtendedPO3 Model Properties

<i>Feature</i>	<i>volatile</i>	<i>transient</i>	<i>changeable</i>	<i>resolveProxies</i>
Supplier.pendingOrders	true	true	false	false
Supplier.shippedOrders	true	true	false	false
PurchaseOrder.totalAmount	true	true	false	