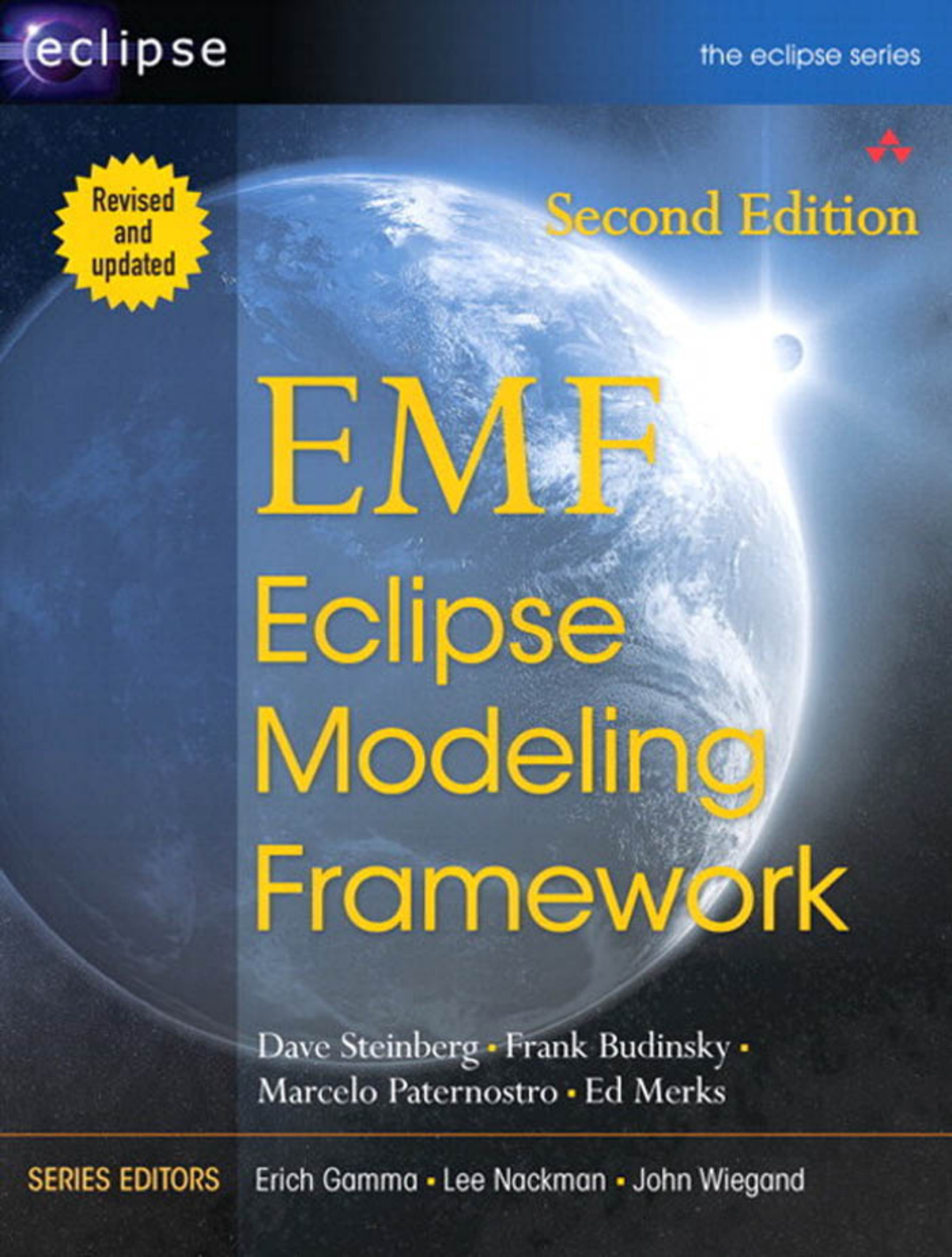
eclipse
the eclipse series
Revised and updated
Second Edition
EMF
Eclipse Modeling Framework
Dave Steinberg • Frank Budinsky •
Marcelo Paternostro • Ed Merks
SERIES EDITORS
Erich Gamma • Lee Nackman • John Wiegand

Library of Congress Cataloging-in-Publication Data:

EMF : Eclipse Modeling Framework / Dave Steinberg ... [et al.].
p. cm.
ISBN 0-321-33188-5 (pbk. : alk. paper) 1. Computer software--Development. 2. Java (Computer program language) I. Steinberg, Dave.

QA76.76.D47E55 2008
005.13'3--dc22

2007049160

ISBN-13: 978-0-321-33188-5
ISBN-10: 0-321-33188-5
Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan
First printing December 2008

Foreword

by Richard C. Gronback

Modeling can mean very different things to different people, even within the discipline of software engineering. Some will immediately think of the Unified Modeling Language (UML), others will think of Model-Driven Architecture (MDA), while others may remember the days of CASE tools. With increasing frequency, those familiar with the Eclipse community think of the Eclipse Modeling Framework (EMF), which provides a solid basis for application development through the use of pragmatic modeling and code generation facilities.

From its beginnings within the Tools Project at Eclipse, EMF's reputation for high quality and unparalleled community support quickly led to several complementary modeling projects forming at Eclipse. Code generators, graphical diagramming frameworks, model transformation, validation, and search are just a few that have built upon EMF and now are contained within the Eclipse Modeling Project. The growth and success of this top-level project is due in large part to the strength of its core component, EMF.

In many ways, the EMF project is a model for other Eclipse projects (pun intended). From the tireless efforts of its committers in answering questions on the project's newsgroup, to the professionalism and openness of its development in terms of API, features, performance, and documentation, EMF is a tough act to follow. The diversity of the Modeling community that has grown up around EMF makes it a poster child for collaboration, including individual contributors, commercial vendors, and academic institutions. Further evidence of EMF's value to Eclipse is its anticipated use in e4, the next Eclipse platform. At present, e4 developers have plans to leverage the capabilities of EMF to provide a consistent model-based foundation and runtime; clearly, a step forward for model-driven software development.

With so much technology built upon EMF, understanding its architecture and capabilities are essential to using it successfully. For years, the first edition of this book has been an important resource for many developers building their

applications with EMF and extending EMF's own capabilities. With the introduction of this second edition, the many enhancements made to EMF in the interim are now documented and able to be leveraged effectively. The API section has been replaced by new chapters covering EMF persistence, client programming, change recording, validation, and Rich Client Platform (RCP) development. In addition to updating the original material, this new edition covers the latest features of EMF versions 2.3 and 2.4, including generics, content types, and REST APIs. It is a most welcome second edition.

I hope you find this second edition as valuable as I do.

Richard C. Gronback
Chief Scientist, Borland Software Corporation
November 2008

Foreword

by Mike Milinkovich

The Eclipse Modeling Framework is an exemplary open source project in many respects. We at the Eclipse Foundation have a number of ways to value the contributions of any one project to our community. Our criteria include industry adoption, robust technology, stability as an extensible platform, and open and transparent project leadership. In all of these ways and more, EMF has for many years been one of the very best of Eclipse.

With respect to industry adoption, I have often marveled at the amazing success of EMF. A large part of my role at the Eclipse Foundation is to travel and speak with adopters of Eclipse technology. Whether I am speaking to startups, enterprises or established software vendors, everywhere I go EMF is on the list of key Eclipse technologies being used. Its ability to simplify the development of complex software applications and products has been widely recognized. Rarely has a single framework driven so much developer productivity in so many domains.

This adoption has largely been driven by a very simple value proposition: EMF is great technology. Careful attention has been paid to EMF's architecture, the completeness of its APIs, its flexibility, and its performance. And performance is key for any technology that is going to be used in real world applications.

As a platform, EMF has transformed the modeling tools industry. Leading model tools vendors such as Borland and IBM have based their products on EMF, making a strategic decision that EMF is their core modeling framework of the future. Almost every new modeling product that I have seen over the past four years has been based on EMF.

So, EMF has clearly seen enormous adoption in the industry, but that is only half the story. We look at EMF as a community success story as well. The EMF team has always done an excellent job of interacting with community. Ed Merks, the leader of the project and one of the authors of this book, is famous throughout the Eclipse community for his prompt responses to any adopter's inquiry

about EMF. That leadership-by-example has resulted in the entire EMF team being one of the most community-focused at Eclipse.

EMF is an enormous Eclipse community success story and I am certain that this book will help further that success.

Mike Milinkovich
Executive Director, Eclipse Foundation
November 2008

Preface

This book is a comprehensive introduction and developer's guide to 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 or an XML Schema, or even by specifying simple annotations on Java interfaces. In this last case, the developer writes just a subset of abstract interfaces that describe the model, and the rest of the code is generated automatically and merged back in.

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 a great deal of coding can be automated, given an appropriate tool. This book shows how EMF is such a tool. It also shows how using EMF lets you do more with your models that you might have thought possible.

EMF provides a runtime framework that allows any modeled data to be easily validated, persisted, and edited in a UI. Change notification and recording are supported automatically. Metadata is available to enable generic processing of any data using a uniform, reflective API. With all of these features and more, EMF is the foundation for data sharing and fine-grained interoperability among tools and applications in Eclipse, in much the same way that Eclipse is itself a platform for integration at the component and UI level. Numerous organizations are currently using Eclipse, EMF, and the growing number of EMF-based technologies in the Eclipse Modeling Project as the foundation for their own commercial and open source offerings.

This book assumes the reader is familiar with object-oriented programming concepts and specifically with the Java programming language. Previous exposure to modeling technologies such as UML, 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 9) presents a thorough overview of EMF's metamodel, Ecore, followed by details of the mappings between Ecore and the other supported model-definition forms: UML, annotated Java, and XML Schema. Part III (Chapters 10 to 13) includes detailed analyses of EMF's code generator patterns and tools, followed by an end-to-end example of a non-trivial EMF application. Finally, Part IV (Chapters 14 to 21) takes a close look at EMF's runtime framework and discusses important EMF programming techniques.

The bulk of this book is based on EMF 2.2, the last version to support the venerable Java 1.4 language. In version 2.3, EMF adopted key language features of Java 5.0, making it incompatible with previous Java runtimes. EMF 2.2, which was current while much of this book was written, is therefore still popular and an excellent base for learning about the framework. The code in Chapters 1 to 20 is based on that version, but due to EMF's backward compatibility, all examples run without change on version 2.4, the latest at the time of this book's release. Chapter 21 focuses specifically on changes in EMF 2.3 and 2.4 and, as such, uses 2.4 as the basis for its examples.

Conventions Used in This Book

The following formatting conventions are used throughout this book:

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 URIs, as well as for placeholder text that is meant to be replaced by a particular name. New terms are often italicized for emphasis. Also, in Chapter 9's example mappings, items shown purely to provide context appear in italics.

`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, command lines, 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.

Online Examples

The Web site for this book is located at *http://www.informit.com/title/9780321331885*. 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.

Eclipse and EMF are required to use the examples. You can download one of several Eclipse packages (we recommend Eclipse Classic) at *http://www.eclipse.org/downloads/* and the all-in-one EMF SDK at *http://www.eclipse.org/modeling/emf/downloads/*.

CHAPTER 2

Introducing EMF

Simply put, the Eclipse Modeling Framework (EMF) is a modeling framework that exploits the facilities provided by 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, because 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 Unified Modeling Language (UML), 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.

2.1 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 getItems(); // List of Item
}

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

  int getQuantity();
  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 added value," you say. Maybe a class diagram or two would fill out the documentation a bit, but other than that it

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

simply doesn't help. So, to appease the boss, you produce the UML diagram shown in Figure 2.1.[2]

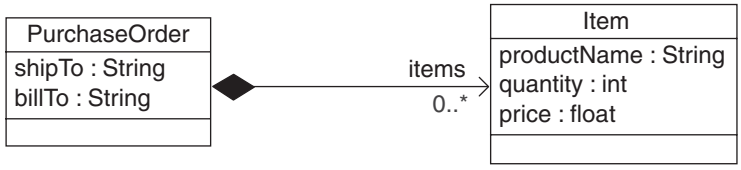

Figure 2.1 UML diagram of interfaces.

Then you tell the boss to go away so you can get down to business. (As you'll see later, 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"
            xmlns:po="http://www.example.com/SimplePO"
            targetNamespace="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>
```

2. If 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.

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 (i.e., 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.

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.

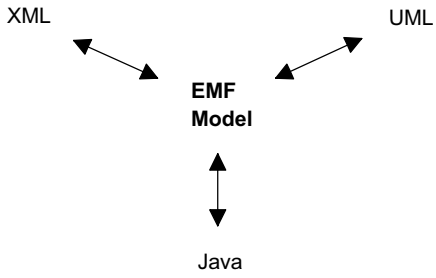

Figure 2.2 EMF unifies Java, XML, and UML.

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 might 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.

2.2 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 and 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 Model Driven Architecture (MDA) 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 because we're going to need to work with (e.g., 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.

3. MDA is described in Section 2.6.4.

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, as the designers of EMF, we 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 EMF 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 portion 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 (e.g., a debugger), 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-grained data integration between applications; next to the productivity gain resulting from code generation, this is one of the most important benefits of modeling.

2.3 Defining the Model

Let's put aside the philosophy for now and take a closer look at what we're really describing with an EMF model. We saw 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.

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 (e.g., UI views) 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 to describe them. More important, 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.

2.3.1 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 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, don't worry about it, as it's really just an academic issue anyway.

A simplified subset of the Ecore metamodel 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 metamodel the classes **EClass**, **EAttribute**, and **EReference** share a common base class, **ENamedElement**, which defines the **name** attribute that here we've shown explicitly in the classes themselves.

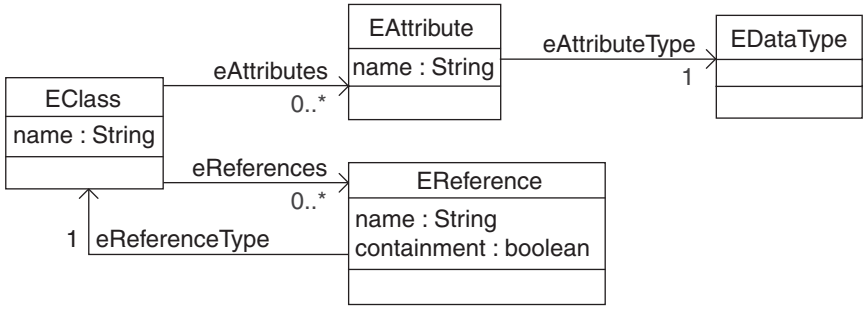

Figure 2.3 A simplified subset of the Ecore metamodel.

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

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`.

Notice that the names of the classes correspond most closely to the UML terms. This is not surprising because 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.

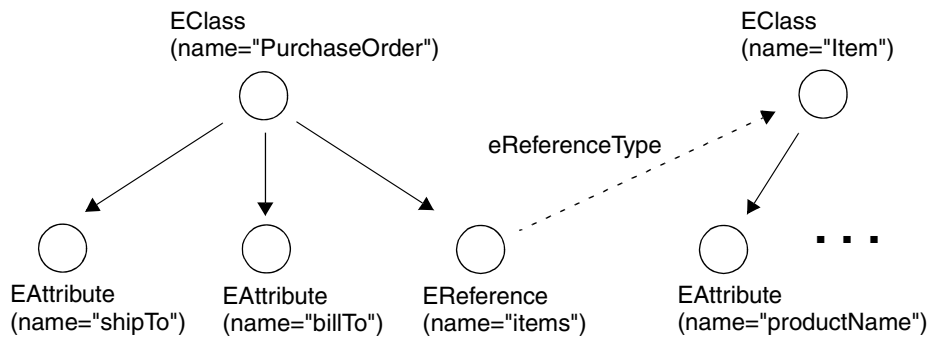

Figure 2.4 The purchase order Ecore instances.

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

2.3.2 Creating and Editing the Model

Now that we have these Ecore objects to represent a model in memory, EMF can read from them to, among other things, generate implementation code. You might be wondering, though, how do you create the model in the first place? The answer is that you need to build it from whatever input form you start with. If you start with Java interfaces, EMF will introspect them and build the Ecore 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 includes a simple tree-based sample editor for Ecore. If you'd rather use a graphical tool, the Ecore Tools project[5] provides a graphical Ecore editor based on UML notation. Third-party options are also available, including Topcased's Ecore Editor (*http://www.topcased.org/*), Omondo's EclipseUML (*http://www.omondo.com/*) and Soyatec's eUML (*http://www.soyatec.com/*).
2. **Import from UML.** The EMF Project and EMF Model wizards provide an extensible framework, into which model importers can be plugged, supporting different model formats. EMF provides support for Rational Rose (*.mdl* files) only. The reason Rose has this special status is because it's the tool that was used to "bootstrap" the implementation of EMF itself. The UML2 project[6] also provides a model importer for standard UML 2.x models.
3. **Export from UML.** This is similar to the second option, but the conversion support is provided exclusively by the UML tool. It is invoked from within the UML tool, instead of from an EMF wizard.

As you might imagine, the first option 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 Ecore model being out of sync with the tool's own native model. The other

5. Ecore Tools is a component of the EMF Technology (EMFT) project, which is itself a subproject of the Eclipse Modeling Project. EMFT is an incubator project for new technologies that extend or complement EMF. The Web site for Ecore Tools is *http://www.eclipse.org/modeling/emft/?project=ecoretools*.

6. UML2 is another component in the Eclipse Modeling Project. It provides an EMF-based implementation of the UML 2.x metamodel and can be found at *http://www.eclipse.org/modeling/mdt/?project=uml2*.

two approaches require an explicit reimport or reexport step whenever the UML model changes.

The advantage of the second and third options 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 Ecore model, and also to both define and generate other parts of your application. As long as a mechanism for conversion to Ecore is provided, that tool will also be usable as an input source for EMF and its generator.

2.3.3 XMI Serialization

By now you might be wondering what the serialized form of an Ecore model is. 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 (i.e., a fourth) persistent form that we use as the canonical representation: XML Metadata Interchange (XMI). Why did we need another one? We weren't exactly short of ways to represent the model persistently.

For starters, Java code, XML Schema, and UML all carry additional information beyond what is captured in an Ecore model. Moreover, none of these forms is required in every scenario in which EMF can be used. Java code was the only one required in our running example, but as we will soon see, even it is optional in some cases. So, what we need is a direct serialization of Ecore, which doesn't add any extra information. XMI fits the bill here, as it is a standard for serializing metadata concisely using XML.

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

```
<?xml version="1.0" encoding="UTF-8"?>
<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://www.example.com/SimplePO" nsPrefix="po">
  <eClassifiers xsi:type="ecore:EClass" name="PurchaseOrder">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="shipTo"
        eType="ecore:EDataType
               http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="billTo"
        eType="ecore:EDataType
```

```
                http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="items"
        upperBound="-1" eType="#//Item" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Item">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
        name="productName"
        eType="ecore:EDataType
                http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="quantity"
        eType="ecore:EDataType
                http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="price"
        eType="ecore:EDataType
                http://www.eclipse.org/emf/2002/Ecore#//EFloat"/>
  </eClassifiers>
</ecore:EPackage>
```

Notice that the XML elements correspond directly to the Ecore instances back in Figure 2.4, which makes perfect sense because this is a serialization of exactly those objects. Here we've hit an important point: because Ecore metadata is not the same as, for example, UML metadata, XMI serializations of the two are not the same either.[7]

2.3.4 Java Annotations

Let's revisit the issue of defining an Ecore 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,[8] 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

7. When we spoke of exporting a model for use with EMF in the previous section, we were really talking about exporting to Ecore XMI, specifically.

8. EMF uses a subset of the JavaBeans simple property accessor naming patterns. For more information, see Section 7.1 of the specification at *http://java.sun.com/products/javabeans/docs/spec.html*.

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 **PurhaseOrder** 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 `getItems()` returns a `List`, we need to specify the target type of the reference as `type="Item"`.[9] 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.

9. Note that beginning with Java 5.0, generics can be used to specify a list's item type. Generics have only been supported in EMF since version 2.3. You'll see the older form, with a raw list type, throughout most of this book, both as the Java specification for a multiplicity-many reference and in the code generated from it. Chapter 21, which focuses specifically on EMF 2.3 and 2.4, details the new generics-based form.

2.3.5 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. An Ecore 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 an Ecore 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 instances.

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 an Ecore 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.

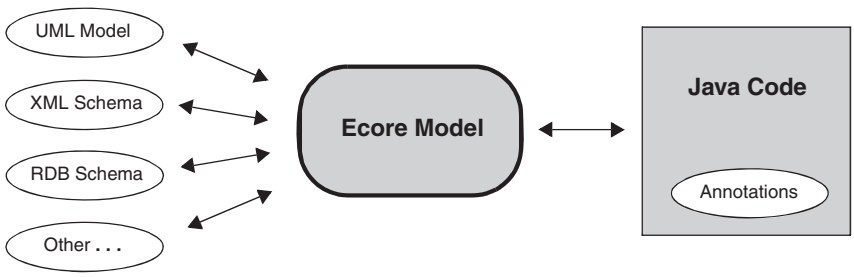

Figure 2.5 An Ecore model and its sources.

2.4 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 Ecore 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 (e.g., 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.

2.4.1 Generated Model Classes

So what kind of code does EMF generate? The first thing to notice is that an Ecore class (i.e., 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 favored by EMF. Why do we advocate this approach? The reason is simply that we believe it's the best pattern for any model-like API. For example, the Document Object Model (DOM) 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 `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 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 [3]. 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 (i.e., 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 might 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 10.

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. EMF's runtime 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. However, because it's generated, you know it's correct. It's a big time saver, especially for some of the more complicated bidirectional 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 (e.g., `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 (e.g., `POPackage`) provides convenient accessors for all the Ecore metadata for the model. You might already have noticed, in the implementation of `setShipTo()` shown earlier, 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.

2.4.2 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 factory[10] class (e.g., `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 (e.g., `POSwitch`) that implements a "switch statement"-like callback mechanism for dispatching based on an object's

10. Adapters and adapter factories are described in Section 2.5.1.

type (i.e., its **EClass**). The adapter factory class, as just described, uses this switch class in its implementation.

3. Plug-in manifest files and property files, so that the model can be used as an Eclipse plug-in.

If all you're interested in is generating a model, 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 EMF core, generate adapter classes that enable viewing and command-based, undoable editing of a model. It can even generate a complete 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.

2.4.3 Regeneration and Merge

The EMF generator produces files that are intended to be a combination of generated pieces and handwritten 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 (i.e., 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, 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, on 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 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 specify that they should be retained during regeneration. The single extends class of a generated class, however, will always be overwritten by the model's choice. We'll look at code merging in more detail in Chapter 10.

2.4.4 The Generator Model

Most of the data needed by the EMF generator is stored in the Ecore model. As we saw 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 Ecore 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 Ecore model. That is, generator model classes are decorators [3] 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 an Ecore model; it's actually a generator model editor.[11] When you use the generator, you'll be editing a generator model, which in turn indirectly accesses the Ecore 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 model resources (files) in the project: an *.ecore* file and a *.genmodel* file. The *.ecore* file is an XMI serialization of the Ecore 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.

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

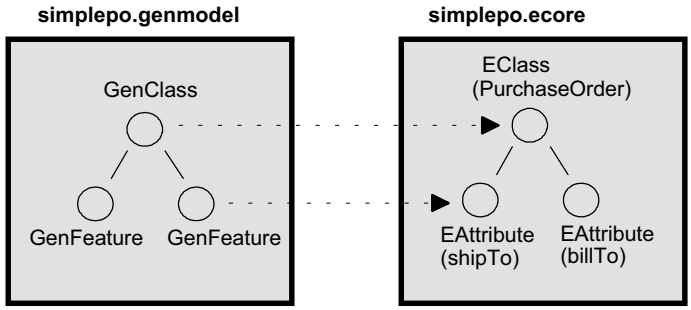

Figure 2.6 The .genmodel and .ecore files.

Separating the generator model from the Ecore 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 Ecore model is that a generator model might get out of sync if the referenced Ecore model changes. To handle this, the generator model elements are able to automatically reconcile themselves with changes to their corresponding Ecore elements. Users don't need to worry about it.

2.5 The Runtime 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 an efficient reflective API for manipulating EMF objects generically. Most importantly, EMF provides the foundation for interoperability with other EMF-based tools and applications.

2.5.1 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 (i.e., support additional interfaces without subclassing) of the object they're attached to. An adapter, as a simple observer, can be attached to any `EObject` (e.g., a `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 (e.g., if the `setBillTo()` method is called), as shown in Figure 2.7.

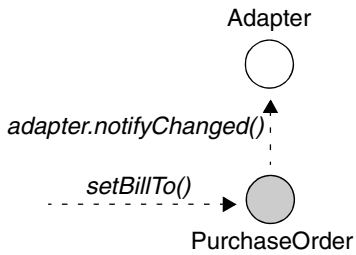

Figure 2.7 Calling the notifyChanged() method.

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, as different factories can return different 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 might want to be 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, EMF provides a very convenient adapter class, `EContentAdapter`, that 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 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 EMF.Edit, as we will see in Chapter 3. We'll also look at how they work in much more detail in Chapter 16.

2.5.2 Object Persistence

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

As we've seen earlier, Ecore 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 persistence 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`.[12] 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 (e.g., a file). To persist our objects, all we need to do is add the root object (i.e., 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 persistence framework: `ResourceSet`.

A `ResourceSet`, as its name implies, is a set of resources that are accessed together to allow for potential cross-document references among them. It's also

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

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 like this:[13]

```
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, after saving our simple resource, we'd get an XMI file, *mypo.xml*, that looks something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<po:PurchaseOrder xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:po="http://www.example.com/SimplePO"
    billTo="123 Maple Street">
  <items productName="Apples" quantity="12" price="0.5"/>
</po: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.

13. If 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 might serialize to use relative URIs, guaranteeing that our serialized document(s) will be location independent. URIs and cross-document referencing are described in detail in Chapter 14.

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, and they are both in the same resource. Imagine, however, that we had modeled **items** as a non-containment reference as shown in 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:

```
/**
 * @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. We also 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. In Figure 2.9, Resource 1 (which could contain our purchase order, for example) contains cross-document references to Resource 2 (e.g., containing our item). When we load Resource 1 by calling `getResource()` for "uri 1", any references to objects in Resource 2 (i.e., "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 (i.e., replaced with the target object).

Although, as we saw earlier, objects in containment references are implicitly included in their container's resource by default, it is also possible to enable cross-resource containment. In Chapters 10 and 15, we'll explore this topic, and look at demand loading, proxies, and proxy resolution in greater detail.

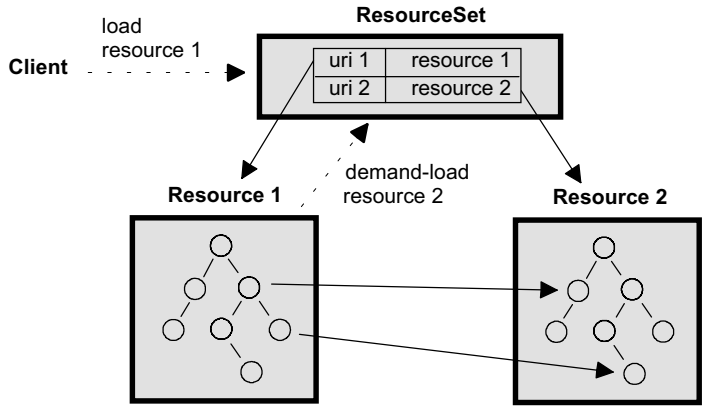

Figure 2.9 Resource set demand loading of resources.

2.5.3 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 fields like this:

```
POFactory poFactory = POFactory.eINSTANCE;
EClass purchaseOrderClass = POPackage.Literals.PURCHASE_ORDER;
EAttribute shipToAttribute =
  POPackage.Literals.PURCHASE_ORDER__SHIP_TO;
```

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 EMF.Edit to implement a full set of generic commands (e.g., `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, whereas `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 10.

2.5.4 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 could simply create the Ecore 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.getEStructuralFeatures().add(shipToAttribute);

// and so on ...
```

Here we have an in-memory Ecore 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.[14] `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

14. This is not entirely true. It could instantiate `EObjectImpl` directly, but instead it actually uses an instance of a simple subclass of `EObjectImpl`, `DynamicEObjectImpl`, which is tuned to provide better performance in the pure dynamic case.

`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. Another direct benefit of this approach is that any **SubPO** object is actually an instance of the Java interface `PurchaseOrder`, as reported by the `instanceof` operator.

The most important point of all of 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 Ecore 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 is the speed of the code.

2.5.5 Foundation for Data Integration

The last few sections have shown various features of the runtime 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 might be partially or completely dynamic (i.e., without generated implementation classes).

In addition to these features, the runtime 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 (e.g., to clean up references when deleting the object) and any unresolved proxies in a resource, among other things.

All these features, combined with an intrinsic property of models—that they are higher level descriptions that can more easily be shared—provide all the needed ingredients to foster fine-grained 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 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-grained data integration in Eclipse.

2.6 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.

2.6.1 Unified Modeling Language

UML 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 first standardized by the OMG in 1997. The standard's latest version is 2.1.2; it is available at *http://www.omg.org/spec/UML/2.1.2/*. The UML2 project, which like EMF belongs to the Eclipse Modeling Project, provides an EMF-based implementation of the UML metamodel.

2.6.2 Meta-Object Facility

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 first standardized in 1997, at the same time as UML. The standard, which is now at version 2.0, is available at *http://www.omg.org/spec/MOF/2.0/*.

Development experience from EMF has substantially influenced this latest version of the specification, in terms of the layering of the architecture and the structure of the semantic core. Essential Meta-Object Facility (EMOF) is the new, lightweight core of the metamodel that quite closely resembles Ecore. Because the two models are so similar, EMF is able to support EMOF directly as an alternate XMI serialization of Ecore.

2.6.3 XML Metadata Interchange

XMI is the standard that connects modeling with XML, defining a simple way to serialize models 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 instances of any EMF model, it is most appropriate for use with models representing metadata; that is, metamodels, like Ecore itself. We refer to an Ecore 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 latest XMI specification, version 2.1.1, is available at *http://www.omg.org/spec/XMI/2.1.1/*.

2.6.4 Model Driven Architecture

MDA is an industry architecture proposed by the OMG that addresses full lifecycle 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 and 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 *http://www.omg.org/mda/*.

CHAPTER 9

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 an Ecore 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.

At a high level, the mapping to Ecore is quite simple:

- A schema maps to an **EPackage**.
- A complex type definition maps to an **EClass**.
- A simple type definition maps to an **EDataType**.
- An attribute declaration or element declaration maps to an **EAttribute** if its type maps to an **EDataType**, or to an **EReference** if its type maps to an **EClass**.

From a modeling perspective, however, 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. To address this, EMF provides a set of extensions to XML Schema in the form of attributes from the Ecore namespace ("http://www.eclipse.org/emf/2002/Ecore"), which can be used to specify this missing information or to customize the mapping in other ways. These attributes are described in the following sections, which correspond to the components to which they apply, and are also summarized in Section 9.10.

Although not expressive enough from a modeling perspective, XML Schema is, at the same time, able to express many details, mostly serialization related, that are not representable in Ecore. Because XML Schema's primary purpose is to define the structure of XML instance documents, instances of the Ecore model should conform to the corresponding schema when they are serialized as XML. EMF records the extra information required to do this on the model using

extended metadata **EAnnotation**s. Recall from Chapter 8 that the **source** of such an **EAnnotation** is "http:///org/eclipse/emf/ecore/util/ExtendedMetaData". The details of all such annotations are described in this chapter, and their use in customizing the default EMF serialization is discussed in Chapter 15.

One important use of extended metadata **EAnnotation**s is to record the original name of an XML Schema component corresponding to an Ecore element whose name is adjusted while mapping to Ecore. Such adjustment is often required because XML Schema naming rules are less restrictive then Java's (and consequently Ecore's). The resulting Ecore names are always valid Java identifiers and conform to the naming conventions outlined in Chapter 6 of the *Java Language Specification* [6]. For example, camel case is used, and **EClassifier** names start with an uppercase letter while **EStructuralFeature** names begin with lowercase.

As we saw briefly in Chapter 8, a number of XML Schema constructs give rise to feature maps in the corresponding Ecore model, where they are used to maintain cross-feature order. Extended metadata annotations are also used capture details about these feature maps, such as what type of construct a feature map represents and, when needed, which other feature's values it can contain.

In the following sections, we'll look at the details of how the various schema components map to Ecore, and how their associated **EAnnotation**s are initialized. For each schema component, the corresponding Ecore representation is described along with any attributes and nested content that affect the resulting model. In some situations, the mapping rules, described later, might result in Ecore elements with conflicting names (e.g., two **EAttribute**s that are in the same **EClass** and have the same name). In such situations, the second and subsequent elements are made unique by appending a number to the end of their names (e.g., "foo1").

Note that an understanding of XML Schema is assumed in this discussion. Readers who are unfamiliar with this technology should first consult an introductory resource, such as the XML Schema primer. [2]

9.1 Schema

An `xsd:schema` maps to an **EPackage**. The **name**, **nsURI**, and **nsPrefix** of the **EPackage** depend on whether or not the schema has a `targetNamespace` attribute.

9.1.1 Schema without Target Namespace

An `xsd:schema` with no `targetNamespace` maps to an **EPackage** initialized as follows:

- **nsURI** = the URI of the schema document
- **nsPrefix** = the last segment of the URI (short file name), excluding the file extension
- **name** = same as **nsPrefix**
- **eAnnotations** = an extended metadata **EAnnotation**

The **details** map of the extended metadata **EAnnotation** contains the following entry:

- **key** = "qualified", **value** = "false"

in resource: file:/c:/myexample/po.xsd	**EPackage**
`<xsd:schema>`	**name**="po"
`...`	**nsPrefix**="po"
`</xsd:schema>`	**nsURI**="file:/c:/myexample/po.xsd"
	EAnnotation
	source="…/ExtendedMetaData"
	details="qualified"→"false"

9.1.2 Schema with Target Namespace

If a schema has a `targetNamespace` attribute, then it is used to initialize the corresponding **EPackage**, as well as to specify the fully qualified Java package name, via the **GenPackage** in the generator model that is created along with the Ecore model to control code generation for it.

In this case, the **EPackage** attributes are set as follows:

- **nsURI** = the `targetNamespace` value
- **nsPrefix** = the last segment of the Java package name (derived from the `targetNamespace`)
- **name** = same as **nsPrefix**

There is no extended metadata **EAnnotation** in this case.

The Java package name, and consequently the **nsPrefix**, is derived from the `targetNamespace` using the following algorithm:

1. Strip the URI scheme and leading slash ("/") characters (e.g., "http://www.example.com/library" → "www.example.com/library").
2. Remove "www" and then reverse the components of the URI authority, if present (e.g., "www.example.com/library" → "com.example/library").
3. Replace slash ("/") characters with dot (".") characters.
4. Split mixed-case names into dot-separated lowercase names.

The **nsPrefix** is then set to the last component of the Java package name.[1] The **basePackage** property in the **GenPackage** is set to the rest of the name.

`<xsd:schema` `  targetNamespace=` `  "http://www.example.com/PrimerPO">` `...` `</xsd:schema>`	**EPackage** **name**="po" **nsPrefix**="po" **nsURI**="http://www.example.com/PrimerPO" ... **GenPackage** **basePackage**="com.example.primer" ...

9.1.3 Global Element or Attribute Declaration

If there is one or more global element or attribute declaration in the schema, an **EClass**, representing the document root, is created in the schema's **EPackage**. The name of the document root class is "DocumentRoot" by default.

`<xsd:schema ... >` `<xsd:element ... />` `...` `</xsd:schema>`	**EPackage** **EClass** **name**="DocumentRoot" ...

A document root class contains one feature for every global attribute or element declaration in the schema (see Sections 9.4.6 and 9.5.7). A single instance of this class is used as the root object of an XML resource (i.e., a conforming XML document). This instance will have exactly one of its element features set: the one corresponding to the global element at the root of the XML document. The features corresponding to global attribute declarations will never be set, but can be used for setting values in attribute wildcard feature maps.

The document root **EClass** looks like one corresponding to a mixed complex type (see Section 9.3.4) including a "mixed" feature, and derived implementations for the other features in the class. This allows it to maintain comments and whitespace that appear in the document, before the root element. A document root class contains two additional features, both string-to-string maps, which are used to record special mappings needed in instance documents. One, named

1. A leading underscore is introduced if the **nsURI** would otherwise start with any case variations of "xml", yielding a valid prefix as defined by the Namespaces in XML recommendation. [5]

"xMLNSPrefixMap", records namespace to prefix mappings, and the other, "xSISchemaLocation", records `xsi:schemaLocation` mappings.

The name of a document root class, if there is one, can be changed from the default ("DocumentRoot") by including an `ecore:documentRoot` attribute on the schema.

`<xsd:schema` `  ecore:documentRoot="PORoot" ... >` `  <xsd:element ... />` `  ...` `</xsd:schema>`	**EPackage** **EClass** **name**="PORoot" ...

9.1.4 Element or Attribute Form Default

Whether qualification of local elements and attributes is required can be globally specified by a pair of attributes, `elementFormDefault` and `attributeFormDefault`, on the schema element, or can be specified separately for each local declaration using the `form` attribute. The value of any of these attributes can be "qualified" or "unqualified", to indicate whether or not locally declared elements and attributes must be qualified in conforming documents.

Neither `elementFormDefault` nor `attributeFormDefault` have any effect on the corresponding **EPackage** or "DocumentRoot" **EClass** (if it exists), but the Ecore model for any corresponding local declarations may include additional information. For details see Sections 9.4.5 and 9.5.6.

`<xsd:schema` `  elementFormDefault="qualified" ... >` `  ...` `</xsd:schema>`	**EPackage** ...

9.1.5 EMF Extensions

The initialization of an **EPackage** corresponding to a schema can be further customized through the use of additional schema attributes from the Ecore namespace.

An `ecore:nsPrefix` attribute can be used to explicitly set the **nsPrefix** attribute of the **EPackage**.

`<xsd:schema` `  ecore:nsPrefix="myprefix"  ... >` `  ...` `</xsd:schema>`	**EPackage** **nsPrefix**="myprefix" ...

An `ecore:package` attribute can be used to specify the fully qualified Java package name corresponding to the schema. It sets both the **name** of the corresponding **EPackage** and the **basePackage** of the **GenPackage** (in the generator model) based on the Java package name, as described in Section 9.1.2.

`<xsd:schema` `  ecore:package=` `  "org.basepackage.mypackage" ... >` `...` `</xsd:schema>`	**EPackage** **name**="mypackage" ... **GenPackage** **basePackage**="org.basepackage" ...

Finally, an `ecore:documentRoot` attribute can be used to specify a nondefault name for the document root class created in the presence of global element or attribute declarations, as discussed in Section 9.1.3.

9.2 Simple Type Definitions

Each simple type definition in a schema maps to an **EDataType** in the **eClassifiers** list of the schema's corresponding **EPackage**. The **name**, **instanceClassName**, and **eAnnotations** of the **EDataType** depend on the contents of the type definition.

In some cases, a single simple type actually maps to two **EDataType**s, where the second represents a wrapper for the first that allows it to be used in certain needed contexts. We'll see these situations in Sections 9.2.1 and 9.2.2.

9.2.1 Restriction

An **EDataType** corresponding to a simple type defined by restriction is initialized as follows:

- **name** = the `name` of the simple type converted, if necessary, to a proper Java class name
- **instanceClassName** = the **instanceClassName** of the **EDataType** corresponding to the base type
- **eAnnotations** = an extended metadata **EAnnotation**

The **details** map of the extended metadata **EAnnotation** contains the following entries:

- **key** = "name", **value** = the unaltered `name` of the simple type
- **key** = "baseType", **value** = the restriction's namespace-qualified `base` type

The restriction's facets, which represent constraints on the base type, are also captured in the **details** map. Each facet produces an additional entry as follows:

- **key** = the name of facet, **value** = the facet's `value`

`<xsd:simpleType name="zipCode">` `  <xsd:restriction base="xsd:int">` `  <xsd:minInclusive value="10000"/>` `  <xsd:maxInclusive value="99999"/>` `  </xsd:restriction>` `</xsd:simpleType>`	**EDataType** **name**="ZipCode" **instanceClassName**="int" **EAnnotation** **source**="…/ExtendedMetaData" **details**="name"→"zipCode", "baseType"→"…/XMLType#int", "minInclusive"→"10000", "maxInclusive"→"99999"

An `ecore:ignore` attribute can be specified on a facet to suppress it in the corresponding **EDataType**.

`<xsd:minInclusive value="10000"` `  ecore:ignore="true"/>`	*No minInclusive entry in **details** map*

When the **EDataType** represents a primitive type (i.e., when **instanceClassName** identifies a Java primitive type), a second **EDataType** must be created for the corresponding wrapper class. This is because the simple type may be used as the type of a nillable element (see Section 9.5.4), and a primitive would be unable to represent the `xsi:nill="true"` state. In this case, the wrapper **EDataType** is initialized as follows:

- **name** = the **name** of the primitive **EDataType**, with the suffix "Object" appended
- **instanceClassName** = the wrapper class for the instance class of the primitive **EDataType** (e.g., "java.lang.Integer" for "int")
- **eAnnotations** = an extended metadata **EAnnotation**

The **details** map of the extended metadata **EAnnotation** contains the following entries:

- **key** = "name", **value** = the `name` of the simple type, with the suffix ":Object" appended
- **key** = "baseType", **value** = the original `name` of the simple type

`<xsd:simpleType name="zipCode">` `  <xsd:restriction base="xsd:int">` `  ...` `  </xsd:restriction>` `</xsd:simpleType>`	***EDataType name**="ZipCode" ...* **EDataType** **name**="ZipCodeObject" **instanceClassName**="java.lang.Integer" **EAnnotation** **source**="…/ExtendedMetaData" **details**="name"→"zipCode:Object", "baseType"→"zipCode"

The wrapper **EDataType** is only used as the type of an **EAttribute** corresponding to a nillable element, as described in Section 9.5.4. The primitive **EDataType** is used in all other circumstances.

9.2.2 Restriction with Enumeration Facets

A restriction with enumeration facets maps to an **EEnum** and a wrapper **EDataType**. The **EEnum** is initialized as follows:

- **name** = the `name` of the simple type converted, if necessary, to a proper Java class name
- **eLiterals** = one **EEnumLiteral** for each enumeration facet in the restriction
- **eAnnotations** = an extended metadata **EAnnotation**

Each **EEnumLiteral** has the following attributes:

- **name** = the `value` of the enumeration facet converted, if necessary, to a valid Java identifier
- **literal** = the unaltered `value` of the enumeration facet
- **value** = an integer value sequentially assigned, starting at 0

The **details** map of the **EEnum**'s extended metadata **EAnnotation** contains the following entry:

- **key** = "name", **value** = the unaltered name of the simple type

`<xsd:simpleType name="USState">` `  <xsd:restriction base="xsd:string">` `  <xsd:enumeration value="A-K"/>` `  <xsd:enumeration value="A-L"/>` `  <!-- and so on ... -->` `  </xsd:restriction>` `</xsd:simpleType>`	**EEnum** **name**="USState" **EEnumLiteral** **name**="AK" **literal**="A-K" **value**=0 **EEnumLiteral** **name**="AL" **literal**="A-L" **value**=1 **EAnnotation** **source**="…/ExtendedMetaData" **details**="name"→ "USState"

If the simple type definition includes an `ecore:enum="false"` attribute, the type maps instead to an ordinary **EDataType** as described in Section 9.2.1. If the Java instance class of such an **EDataType** is primitive (e.g., `int`), **EAttribute**s of the type will have a default value set (see Sections 9.4.4 and 9.5.5).

An `ecore:name` attribute can be added to an enumeration facet to specify the **name** attribute of the corresponding **EEnumLiteral**.

`<xsd:enumeration value="A-K"` `  ecore:name="A_K"/>`	**EEnumLiteral** **name**="A_K" …

Likewise, an `ecore:value` attribute can be used to specify the **value** attribute of an **EEnumLiteral**.

`<xsd:enumeration value="A-K"` `  ecore:value="100"/>`	**EEnumLiteral** **value**="100" …

In Ecore, the only valid values of an **EEnum** are its literals; null is not allowed. As a result, **EEnum**s have the same limitation as primitive **EDataType**s: they cannot be used as the type of an attribute corresponding to a nillable element. So, a wrapper **EDataType** is needed for each **EEnum**, for use only in that context. The attributes of the **EDataType** are as follows:

- **name** = the **name** of the **EEnum**, with the suffix "Object" appended
- **instanceClassName** = "org.eclipse.emf.common.util.Enumerator"
- **eAnnotations** = an extended metadata **EAnnotation**

The **details** map of the extended metadata **EAnnotation** contains the following entries:

- **key** = "name", **value** = the `name` of the simple type, with the suffix ":Object" appended
- **key** = "baseType", **value** = the original `name` of the simple type

`<xsd:simpleType name="USState">` `  <xsd:restriction base="xsd:string">` `  <xsd:enumeration value="A-K"/>` `  ...` `  </xsd:restriction>` `</xsd:simpleType>`	***EEnum name**="USState" …* **EDataType** **name**="USStateObject" **instanceClassName**= "org.eclipse.emf.common.util.Enumerator" **EAnnotation** **source**="…/ExtendedMetaData" **details**="name"→"USState:Object", "baseType"→"USState"

9.2.3 List Type

An **EDataType** corresponding to a list simple type is initialized as follows:

- **name** = the `name` of the simple type converted, if necessary, to a proper Java class name **instanceClassName** = "java.util.List"
- **eAnnotations** = an extended metadata **EAnnotation**

The **details** map of the extended metadata **EAnnotation** contains the following entries:

- **key** = "name", **value** = the unaltered `name` of the simple type
- **key** = "itemType", **value** = the `itemType` of the list

`<xsd:simpleType name="nameList">` `  <xsd:list itemType="xsd:NCName"/>` `</xsd:simpleType>`	**EDataType** **name**="NameList" **instanceClassName**="java.util.List" **EAnnotation** **source**="…/ExtendedMetaData" **details**="name"→"nameList", "itemType"→"…/XMLType#NCName"

9.2.4 Union Type

An **EDataType** corresponding to a union simple type is initialized as follows:

- **name** = the `name` of the simple type converted, if necessary, to a proper Java class name **instanceClassName** = a common instance class of the members (if there is one) or "java.lang.Object"
- **eAnnotations** = an extended metadata **EAnnotation**

If the **EDataType**s corresponding to the union members share a common **instanceClassName**, then the **instanceClassName** of the union's **EDataType** is set to this common value. If they are not all the same, then "java.lang.Object" is used instead.

The **details** map of the extended metadata **EAnnotation** contains the following entries:

- **key** = "name", **value** = the unaltered `name` of the simple type
- **key** = "memberTypes", **value** = the space-separated list of `memberTypes` in the union

<xsd:simpleType name="zipUnion"> <xsd:union memberTypes="zipCode USState"/> </xsd:simpleType>	**EDataType** **name**=" ZipUnion" **instanceClassName**="java.lang.Object" **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"zipUnion", "memberTypes"→"zipCode USState"

9.2.5 Anonymous Type

Even when defined anonymously, a simple type still maps to an **EDataType** in the containing package's **eClassifiers** list. If an anonymous simple type is used in an element or attribute declaration, then the corresponding **EDataType**'s **name** is obtained by appending the suffix "Type" to the converted `name` of that enclosing element or attribute,. The "name" entry in the **details** map of the extended metadata **EAnnotation** has as its value the original `name` of the enclosing element or attribute, with the suffix "_._type" appended. Any additional entries that would appear in the **details** map for the simple type are unchanged.

<*xsd:element name="myElement*"> <xsd:simpleType> ... </xsd:simpleType> </*xsd:element*>	**EDataType** **name**="MyElementType" **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"myElement_._type", … …

If an anonymous simple type is used as the base type of a restriction, then the corresponding **EDataType**'s **name** is based on the enclosing type's converted `name` and carries the suffix "Base", instead of "Type". The "name" entry in the **details** map of the extended metadata **EAnnotation** has the suffix "_._base" in this case.

<*xsd:simpleType name="myType*"> <*xsd:restriction*> <xsd:simpleType> ... </xsd:simpleType> </*xsd:restriction*> </*xsd:simpleType*>	**EDataType** **name**="MyTypeBase" **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"myType_._base", … …

Similarly, if an anonymous simple type is used as the item type of a list, then the corresponding **EDataType**'s **name** is obtained by appending the suffix "Item" to the enclosing type's converted `name`. The "name" entry in the **details** map of the extended metadata **EAnnotation** has the suffix "_._item".

`<xsd:simpleType name="myType">` `  <xsd:list>` `  <xsd:simpleType>` `  ...` `  </xsd:simpleType>` `  </xsd:list>` `</xsd:simpleType>`	**EDataType** **name**="MyTypeItem" **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"myType_._item", … …

Finally, if an anonymous simple type is used as a member type of a union, then the corresponding **EDataType**'s **name** is formed from the enclosing type's converted `name` and the suffix "Member", but in this case, it ends with a number representing the position (starting from 0) of the member in the union. The "name" entry in the **details** map of the extended metadata **EAnnotation** has the suffix "_._member", also qualified with the position number.

`<xsd:simpleType name="myType">` `  <xsd:union>` `  <xsd:simpleType>` `  ...` `  </xsd:simpleType>` `  ...` `  </xsd:union>` `</xsd:simpleType>`	**EDataType** **name**="MyTypeMember0" **EAnnotation** **source=**"…/ExtendedMetaData" **details**= "name"→"myType_._member0", … …

9.2.6 EMF Extensions

In addition to the `ecore:ignore`, `ecore:enum`, `ecore:name`, and `ecore:value` attributes described in Sections 9.2.1 and 9.2.2, there are several attributes from the Ecore namespace that are applicable to simple type declarations in general.

An `ecore:name` attribute can be used to set the **name** of the **EDataType**, for example, if the corresponding simple type is anonymous or if the default name conversion is unacceptable.

`<xsd:simpleType name="stName"` `  ecore:name="MyName">` `  ...` `</xsd:simpleType>`	**EDataType** **name**=“MyName” …

An `ecore:instanceClass` attribute can be used to set the **instanceClassName** attribute of the corresponding **EDataType**.

`<xsd:simpleType name="date"` `  ecore:instanceClass="java.util.Date">` `  ...` `</xsd:simpleType>`	**EDataType** **name**=“Date” **instanceClassName**=“java.util.Date” …

The “baseType” (see Section 9.2.1) is not recorded in the **details** map of the extended metadata **EAnnotation** in this case.

An `ecore:serializable` attribute can be used to set the **serializable** attribute of the corresponding **EDataType**.

`<xsd:simpleType name="date"` `  ecore:serializable="false">` `  ...` `</xsd:simpleType>`	**EDataType** **name**=“Date” **serializable**=“false” …

An `ecore:constraints` attribute can be used to declare named constraints by adding an Ecore-sourced **EAnnotation** to the corresponding **EDataType**. Constraints and validation are discussed in depth in Chapter 18.

`<xsd:simpleType name="date"` `  ecore:constraints="A B">` `  ...` `</xsd:simpleType>`	**EDataType** **name**=“Date” **EAnnotation** **source=**“ …/emf/2002/Ecore” **details**=“constraints”→“A B”

9.3 Complex Type Definitions

Each complex type definition of a schema maps to an **EClass** in the **eClassifiers** list of the schema’s corresponding **EPackage**. Such an **EClass** is initialized as follows:

- **name** = the `name` of the complex type converted, if necessary, to a proper Java class name
- **eAnnotations** = an extended metadata **EAnnotation**

The **details** map of the extended metadata **EAnnotation** contains the following entries:

- **key** = "name", **value** = the unaltered `name` of the simple type
- **key** = "kind", **value** = one of "empty", "simple", "elementOnly", or "mixed"

The value of the "kind" **details** entry depends on the content type of the complex type definition.

`<xsd:complexType name="globalAddress">` `  <xsd:complexContent>` `  ...` `  </xsd:complexContent>` `</xsd:complexType>`	**EClass** **name**="GlobalAddress" **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"globalAddress", "kind"→"elementOnly"

The complex type's attributes, elements, groups, and wildcards map to **EStructuralFeature**s of its corresponding **EClass**. These mappings will be discussed later, in Sections 9.4 through 9.7.

9.3.1 Extension and Restriction

If a complex type is an extension or restriction of another complex type, then the `base` type's corresponding **EClass** is added to the **eSuperTypes** of the **EClass**.

`<xsd:complexType name="globalAddress">` `  <xsd:complexContent>` `  <xsd:extension base="Address">` `  ...` `  </xsd:extension>` `  </xsd:complexContent>` `</xsd:complexType>`	**EClass** **name**="GlobalAddress" **eSuperTypes**="//Address" …

In the case of extension, attribute and element declarations within the body of the extension also produce features in the **EClass** as described in the following sections.

The mapping for the contents of a restriction, however, depends on whether or not the `base` type contains any wildcards (i.e., if its definition includes `xsd:any` or `xsd:anyAttribute` elements). If the restricted `base` type contains no wildcards, everything in the restriction body is ignored and the corresponding **EClass** contains no new features. In this case, the subclass is simply provided to

restrict the existing features, for example, to constrain their multiplicity or to make their types narrower. Because Ecore does not allow inherited features to be redeclared, such restrictions are not captured in the Ecore representation.

If, on the other hand, the `base` type contains wildcards for which the restricted complex type introduces new elements or attributes, the corresponding derived **EClass** includes features for them.

`<xsd:complexType name="MyBaseType">` `  <xsd:sequence>` `  <xsd:element name="element1" ... />` `  <xsd:any maxOccurs="unbounded" ... />` `  </xsd:sequence>` `</xsd:complexType>` `<xsd:complexType name="MyType">` `  <xsd:complexContent>` `  <xsd:restriction base="MyBaseType">` `  <xsd:sequence>` `  <xsd:element name="element1" ... />` `  <xsd:element name="element2" ... />` `  </xsd:sequence>` `  </xsd:restriction>` `  </xsd:complexContent>` `</xsd:complexType>`	**EClass** **name**="MyType" **eSuperTypes**="//MyBaseType" … **EAttribute** **name**="element2" **volatile**=true **transient**=true **derived**=true *(from "any")* …

These new features have derived implementations that delegate to the feature map for the `base` type's `xsd:any` or `xsd:anyAttribute` feature (see Section 9.7). This is similar to the way the features of a mixed complex type delegate to a "mixed" feature map, as we will see in Section 9.3.4.

9.3.2 Simple Content

A complex type with simple content is defined as an extension or restriction of a simple type. Instead of adding an **eSuperType** to the corresponding **EClass**, a single **EAttribute** is added to its **eAttributes** to represent the simple content. This **EAttribute** is initialized as follows:

- **name** = "value"
- **eType** = an **EDataType** corresponding to the simple `base` of the extension or restriction
- **eAnnotations** = an extended metadata **EAnnotation**

The **details** map of the extended metadata **EAnnotation** on the **EAttribute** contains the following entries:

- **key** = "name", **value** = ":0"
- **key** = "kind", **value** = "simple"

The "kind" of the **EClass** is also "simple" in this case.

`<xsd:complexType name="richInt">` `  <xsd:simpleContent>` `  <xsd:extension base="xsd:int">` `  ...` `  </xsd:extension>` `  </xsd:simpleContent>` `</xsd:complexType>`	**EClass** **name**="RichInt" **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"richInt", "kind"→"simple" **EAttribute** **name**="value" **eType**="…/XMLType#//Int" **EAnnotation** **source=**"…/ExtendedMetaData" **details**= "name"→":0", "kind"→"simple" …

9.3.3 Anonymous Type

If an anonymous complex type is used as the type of an element declaration, then the corresponding **EClass**'s **name** is obtained by appending the suffix "Type" to the enclosing element's converted name. The value of the "name" entry in the **details** map of the extended metadata **EAnnotation** is based on the original, uncoverted name of the enclosing element and carries the suffix "_._type".

<xsd:element name="myElement"> `  <xsd:complexType>` `  ...` `  </xsd:complexType>` *</xsd:element>*	**EClass** **name**="MyElementType" **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name-">"myElement_._type", …

9.3.4 Abstract Type

If a complex type definition includes an `abstract` attribute, it is used to set the **abstract** attribute of the corresponding **EClass**.

`<xsd:complexType abstract="true" ... >` `  ...` `</xsd:complexType>`	**EClass** **abstract**=true …

9.3.5 Mixed Type

A complex type with mixed content produces a feature map **EAttribute** named "mixed" in the corresponding **EClass**. This **EAttribute** includes the following entries in the **details** map of its extended metadata **EAnnotation**:

- **key** = "name", **value** = ":mixed"
- **key** = "kind", **value** = "elementWildcard"

In this case, the "kind" of the **EClass** is "mixed".

`<xsd:complexType name="MixedType"` `  mixed="true">` `  ...` `</xsd:complexType>`	**EClass** **name**="MixedType" **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"MixedType", "kind"→"mixed" **EAttribute** **name**="mixed" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→":mixed", "kind"→"elementWildcard" …

The **EAnnotation** specifying the special name ":mixed" identifies the attribute as the mixed feature for the class, of which there can only be one. All other features (**EReference**s and **EAttribute**s) mapped from element declarations in the schema are **derived**, with implementations that delegate to the mixed feature map.

`<xsd:complexType name="customersType"` *`  mixed="true">`* `  <xsd:sequence>` `  <xsd:element name="customer" ... />` `  </xsd:sequence>` *`</xsd:complexType>`*	***EClass name**="CustomersType" …* **EAttribute** **name**="customer" **volatile**=true **transient**=true **derived**=true *(from "mixed")* …

This structure allows values of the derived references to be mixed with values of the special features **XMLTypeDocumentRoot.text**, **XMLTypeDocumentRoot.cDATA**, and **XMLTypeDocumentRoot.comment**. These features, defined in EMF's XMLType model, represent simple text, character data sections, and XML comments, respectively.[2]

An `ecore:mixed` attribute can be added to a complex type that is not actually mixed in order to produce the same feature-map-based mapping described in this section. The complex type must have complex content and cannot be an extension or restriction of another complex type. This feature is typically used to provide support for adding and accessing comments and whitespace in an XML document,[3] as opposed to real mixed text. Adding any non-whitespace text to instances of such a type would produce an invalid document.

`<xsd:complexType ecore:mixed="true" ... >` `  ...` `</xsd:complexType>`	**EClass** ... **EAttribute** **name**="mixed" **eType**="…/Ecore#//EFeatureMapEntry" …

It is also possible to use a **name** other than "mixed" for the mixed feature. To do so, an `ecore:featureMap` attribute is added to the complex type definition, and the desired name is specified as its value. This works both for real mixed complex types and for other complex types on which the `ecore:mixed` attribute is specified.

`<xsd:complexType mixed="true"` `  ecore:featureMap="order" ... >` `  ...` `</xsd:complexType>`	**EClass** ... **EAttribute** **name**="order" **eType**="…/Ecore#//EFeatureMapEntry" …

An `ecore:featureMap` attribute can also be specified, without `ecore:mixed`, on a complex type to introduce another, subtly different feature-map-based structure in the corresponding **EClass**. Once again, this is only permitted on a complex type that has complex content and that is not an extension or restriction of another type. In this case, the value of the `ecore:featureMap`

2. Beginning in EMF 2.3, processing instructions are also represented using the special feature `XMLTypeDocumentRoot.processingInstruction`.

3. For comments and whitespace to be read from XML documents, the `OPTION_USE_LEXICAL_HANDLER` resource option must be enabled, as described in Chapter 15.

attribute still determines the **name** of the feature map **EAttribute**; however, its **EAnnotation** contains the following **details** entries:

- **key** = "name", **value** = ":group"
- **key** = "kind", **value** = "group"

`<xsd:complexType` `  ecore:featureMap="myMap" ... >` `  ...` `</xsd:complexType>`	**EClass** ... **Eattribute** **name**="myMap" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* **EAnnotation** **source**="…/ExtendedMetaData" **details**="name"→":group", "kind"→"group" …

This structure closely resembles the one used to handle a repeating model group, which is described in Section 9.6.1. All other features mapped from elements in the complex type are still derived from the feature map, as described in that section. The feature map is strictly limited to values of those features, however, and does not allow text, character data or comments, unlike in the case of a mixed type.

9.3.6 EMF Extensions

In addition to the `ecore:mixed` and `ecore:featureMap` attributes described in the previous section, there are several Ecore-namespace attributes that are applicable to complex type declarations in general.

An `ecore:name` attribute can be used to set the **name** of the **EClass**, for example, if the corresponding complex type is anonymous or if the default name conversion is unacceptable.

`<xsd:complexType name="ctName"` `  ecore:name="MyName"` `  ...` `</xsd:complexType>`	**EClass** **name**="MyName" …

An `ecore:instanceClass` attribute can be used to set the **instanceClassName** attribute of the corresponding **EClass**.

`<xsd:complexType` `  ecore:instanceClass="java.io.Serializable">` `  ...` `</xsd:complexType>`	**EClass** **instanceClassName**="java.io.Serializable" ...

An `ecore:interface` attribute can be used to set the **interface** attribute of the corresponding **EClass**.

`<xsd:complexType ecore:interface="true">` `  ...` `</xsd:complexType>`	**EClass** **interface**="true" ...

An `ecore:implements` attribute can be used to specify additional **eSuperTypes** for the corresponding **EClass**. The value of the attribute must be a space-separated list of qualified names, each of which resolves to corresponding complex type.

`<xsd:complexType` `  ecore:implements="MyOtherType">` `  ...` `</xsd:complexType>`	**EClass** **eSuperTypes**="... //MyOtherType" ...

An `ecore:constraints` attribute can be used to declare named constraints by adding an Ecore-sourced **EAnnotation** to the corresponding **EClass**. Constraints and validation are discussed in depth in Chapter 18.

`<xsd:complexType ecore:constraints="A B">` `  ...` `</xsd:complexType>`	**EClass** **EAnnotation** **source**="…/emf/2002/Ecore" **details**="constraints"→"A B"

9.3.7 Operations

The **EOperation**s of a complex type's corresponding **EClass** can be specified directly in the schema using a specialized appinfo annotation (see Section 9.8.2). To be recognized as defining **EOperation**s, the `xsd:appinfo` element must have the following two attributes: a `source` whose value is "http://www.eclipse.org/emf/2002/Ecore", and an `ecore:key` whose value is "operations".

Each **EOperation** of the **EClass** is represented by an `operation` element within the `xsd:appinfo`. The features of the **EOperation** are specified by attributes and nested elements of the `operation` as follows:

- **name** = the value of the `name` attribute
- **eType** = an **EDataType** or **EClass** corresponding to the simple or complex type specified as the `type` attribute, or null if that attribute is absent
- **eParameters** = a list of **EParameters**, one per nested `parameter` element
- **eExceptions =** a list of **EDataTypes** and **EClasses** corresponding to the space-separated list of simple and complex types in the `exceptions` attribute
- **lowerBound** = the value of the `lowerBound` attribute
- **upperBound** = the value of the `upperBound` attribute
- **ordered** = the value of the `ordered` attribute
- **unique** = the value of the `unique` value
- **eAnnotations** = a list of **EAnnotations**, one per nested `annotation` element

```
<xsd:complexType ... >
  <xsd:annotation>
    <xsd:appinfo source=
        "http://www.eclipse.org/emf/2002/Ecore"
        ecore:key="operations">
      <operation name="foo" type="xsd:string"
          lowerBound="1" upperBound="-1"
          exceptions="Exception">
        ...
      </operation>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:complexType>
```

EClass …
  EOperation
    name=“foo”
    eType=“…/XMLType#//String”
    lowerBound=1
    upperBound=-1 *(unbounded)*
    eExceptions=“//Exception”
    …

Each **EParameter** of the **EOperation** is initialized from the corresponding `parameter` element as follows:

- **name** = the value of the `name` attribute
- **eType** = an **EDataType** or **EClass** corresponding to the value of the `type` attribute
- **lowerBound** = the value of the `lowerBound` attribute
- **upperBound** = the value of the `upperBound` attribute
- **ordered** = the value of the `ordered` attribute
- **unique** = the value of the `unique` attribute
- **eAnnotations** = list of **EAnnotations**, one per nested `annotation` element

`<operation name="foo" ... >` `  <parameter name="x"` `  type="xsd:string"` `  lowerBound="1"` `  upperBound="-1"/>` `</operation>`	**EOperation name**="foo" … **EParameter** **name**="x" **eType**="…/XMLType#//String" **lowerBound**=1 **upperBound=-1** (*unbounded*)

If the `operation` element contains a nested `body` element, the corresponding **EOperation** includes an **EAnnotation** with source "http://www.eclipse.org/emf/2002/GenModel" and the following entry in its **details** map:

- **key** = "body", **value** = the text content of the `body` element

As discussed in Section 5.7.1, this value should be the Java code that implements the **EOperation**. The code generator will make use of it, including it in the generated method.

`<operation name="foo" ... >` `  ...` `  <body>return x;</body>` `</operation>`	**EOperation name**="foo" … **EAnnotation** **source=**"…/emf/2002/GenModel" **details=**"body"→"return x;"

Arbitrary **EAnnotation**s on an **EOperation** or **EParameter** can be specified by nesting `annotation` elements in the corresponding `operation` or `parameter`.[4] Each **EAnnotation**'s **source** is initialized using the value of the `annotation` element's `source` attribute, and one entry is added to the **details** map for each nested `detail` element as follows:

- **key** = the value of the `key` attribute, **value** = the content of the `detail` element

`<operation name="foo" ... >` `  <annotation` `  source="http://www.example.com/A1"/>` `  <parameter name="x" ... >` `  <annotation` `  source="http://www.example.com/A1">` `  <detail key="key0">someValue</detail>` `  <detail key="key1">otherValue</detail>` `  </annotation>` `  </parameter>` `  </annotation>` `  ...` `</operation>`	**EOperation name**="foo" … **EAnnotation** **source=**"http://www.example.com/A1" **EParameter name**="x" … **EAnnotation** **source=**"http://www.example.com/A1" **details**="key0"→"someValue", "key1"→"otherValue"

4. Elsewhere, schema `xsd:annotations` map directly to **EAnnotations**. See Section 9.8 for details.

9.4 Attribute Declarations

Each schema attribute declaration maps to an **EAttribute** or **EReference** in the **EClass** corresponding to the complex type definition containing the attribute if locally defined, or in the "DocumentRoot" **EClass** if the attribute is global.

An attribute declaration maps to an **EReference** in only a few special cases, which are described in Section 9.4.3. Otherwise, it maps to an **EAttribute** that is initialized as follows:

- **name** = the `name` of the attribute converted, if necessary, to a proper Java field name
- **eType** = an **EDataType** corresponding to the attribute's simple type
- **lowerBound** = 0 if `use="optional"` (the default), or 1 if `use="required"` (see Section 9.4.3)
- **upperBound** = 1
- **eAnnotations** = an extended metadata **EAnnotation**

If the type of the attribute is one of the predefined schema types, then the **eType** of the **EAttribute** is set to the corresponding **EDataType** from the XMLType model (see Section 9.9). Otherwise, the **eType** is set to a user-defined **EDataType** created from the simple type as described in Section 9.2.

The **details** map of the extended metadata **EAnnotation** contains the following entries:

- **key** = "name", **value** = the unaltered `name` of the attribute
- **key** = "kind", **value** = "attribute"

`<xsd:attribute name="productName"` `  type="xsd:string"/>`	**EAttribute** **name**="productName" **eType**="…/XMLType#//String" **lowerBound**=0 **upperBound**=1 **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"productName", "kind"→"attribute"

9.4.1 ID Attribute

An attribute of type `xsd:ID`, or of any type derived from it, maps to an **EAttribute** whose type is the "ID" **EDataType** from the XMLType model (see Section 9.9). In addition, the **iD** attribute of the **EAttribute** is set to true.

`<xsd:attribute name="id" type="xsd:ID"/>`	**EAttribute** **name**="id" **eType**="…/XMLType#//ID" **iD**=true …

9.4.2 ID Reference or URI Attribute

Attributes of types `xsd:IDREF`, `xsd:IDREFS`, and `xsd:anyURI`, and of types derived from them, usually are intended to represent references to objects defined elsewhere in a document. However, by default, they are handled no differently from attributes of other predefined schema simple types. They simply map to **EAttribute**s with **eType** set to the corresponding **EDataType** from the XMLType model (see Section 9.9). Such an **EAttribute** can only record the value of the object identifier appearing in a document, not refer to the object it actually represents.

`<xsd:attribute name="customer"` `type="xsd:IDREF"/>`	**EAttribute** **name**="customer" **eType**="…/XMLType#//IDREF" …

If, however, an attribute of one of these three types also includes an `ecore:reference` attribute, it maps to an **EReference** instead, capturing the semantic intent of the model. The reference is non-containment (**containment** is false) and its **eType** is set to the **EClass** corresponding to the complex type specified by the `ecore:reference` attribute. The **upperBound** is set to 1 if the attribute's type is `xsd:IDREF` or `xsd:anyURI`, or -1 (unbounded) for `xsd:IDREFS`. For `xsd:IDREF` and `xsd:IDREFS`, which cannot span documents, **resolveProxies** is set to false. For `xsd:anyURI`, which can span documents, it is set to true.

`<xsd:attribute name="customer"` `  type="xsd:IDREF"` `  ecore:reference="Customer"/>`	**EReference** **name**="customer" **eType**="//Customer" **upperBound**=1 **containment**=false **resolveProxies**=false …
`<xsd:attribute name="customers"` `  type="xsd:IDREFS"` `  ecore:reference="Customer"/>`	**EReference** **name**="customers" **eType**="//Customer" **upperBound**=-1 *(unbounded)* **containment**=false **resolveProxies**=false …
`<xsd:attribute name="customer"` `  type="xsd:anyURI"` `  ecore:reference="Customer"/>`	**EReference** **name**="customer" **eType**="//Customer" **upperBound**=1 **containment**=false **resolveProxies**=true …

If the relationship is bidirectional, an `ecore:opposite` attribute can be used to specify the attribute or element of the target complex type that corresponds to the reverse (**eOpposite**) **EReference**.

`<xsd:attribute name="customer"` `  type="xsd:anyURI"` `  ecore:reference="Customer"` `  ecore:opposite="orders"/>`	**EReference** **name**="customer" **eType**="//Customer" **upperBound**=1 **containment**=false **resolveProxies**=true **eOpposite**="//Customer/orders" …

The `ecore:opposite` attribute can be specified on either (or both) sides of the relationship.

9.4.3 Required Attribute

The **lowerBound** of an **EAttribute** or **EReference** corresponding to a required schema attribute is set to 1, instead of the default value of 0.

`<xsd:attribute use="required" ... />`	**EAttribute** **lowerBound**=1 …

9.4.4 Default Value

Specifying a `default` value on an attribute sets the **defaultValueLiteral** attribute of the corresponding **EAttribute**. The **EAttribute** is also unsettable in this case.

`<xsd:attribute name="message"` `  type="xsd:string"` `  default="hello world" ... />`	**EAttribute** **name**="message" **eType**="…/XMLType#//String" **defaultValueLiteral**="hello world" **unsettable**=true …

An attribute declaration without an explicit `default` value also maps to an unsettable **EAttribute** if the type has an intrinsic default value that is non-null (i.e., if the corresponding **eType** is an **EEnum** or an **EDataType** representing a primitive Java type).

`<xsd:attribute name="quantity"` `  type="xsd:int"/>`	**EAttribute** **name**="quantity" **eType**="…/XMLType#//Int" **unsettable**=true …

Section 9.2.2 described how a simple type restriction with enumeration facets can map to an ordinary **EDataType**, instead of an **EEnum**, when `ecore:enum="false"` is specified. If such a simple type maps to a primitive **EDataType** and is used as the type of an attribute, then the resulting **EAttribute** has its default value set, even if no explicit `default` is specified in the attribute declaration. In this case, the **defaultValueLiteral** of the **EAttribute** is set to the first enumeration value of the simple type.

`<xsd:attribute name="oneThreeFive">` `  <xsd:simpleType ecore:enum="false">` `  <xsd:restriction base="xsd:int">` `  <xsd:enumeration value="1"/>` `  <xsd:enumeration value="3"/>` `  <xsd:enumeration value="5"/>` `  </xsd:restriction>` `  </xsd:simpleType>` `</xsd:attribute>`	**EDataType** **name**="OneThreeFiveType" **instanceClassName**="int" … **EAttribute** **name**="oneThreeFive" **eType**="//OneThreeFiveType" **defaultValueLiteral**="1" **unsettable**=true …

9.4.5 Qualified Attribute

If a local attribute declaration has qualified form, either explicitly declared with a `form="qualified"` attribute or inherited from an `xsd:schema` with `attributeFormDefault="qualified"` (see Section 9.1.7), the **details** map of the extended metadata **EAnnotation** for the corresponding feature contains an additional entry:

- **key** = "namespace", **value** = "##targetNamespace"

`<xsd:attribute form="qualified" ... />`	**EAttribute** … **EAnnotation** **source=**"…/ExtendedMetaData" **details=** "namespace"→"##targetNamespace", …

9.4.6 Global Attribute

The **EAttribute** or **EReference** corresponding to a global attribute declaration is added to the package's "DocumentRoot" **EClass** as described in Section 9.1.5, unless it has an `ecore:ignore="true"` attribute, in which case it is ignored. The extended metadata **EAnnotation** on the feature also includes exactly the same "namespace" **details** entry (with value "##targetNamespace") as in the case of a qualified attribute, which was described in Section 9.4.5.

<xsd:schema ... > <xsd:attribute name="globalAttribute" type="xsd:string"/> ... *</xsd:schema>*	***EClass name**="DocumentRoot" …* **EAttribute** **name=**"globalAttribute" **eType=**"…/XMLType#//String" … **EAnnotation** **source=**"…/ExtendedMetaData" **details=** "namespace"→"##targetNamespace", …

9.4.7 Attribute Reference

An attribute reference (i.e., one with a `ref` attribute) maps to an **EAttribute** or **EReference** with a "namespace" entry in the **details** map of its extended metadata **EAnnotation**. If the reference is to a global attribute defined (or included) in the same schema, the value of this entry is "##targetNamespace".

<xsd:complexType ... > ... <xsd:attribute ref="globalAttribute"/> </xsd:complexType>	*EClass* ... **EAttribute** ... **EAnnotation** **source=**"…/ExtendedMetaData" **details=** "namespace"→"##targetNamespace", ...

However, if the reference is to a global attribute from a different schema, then the value of the "namespace" entry is set instead to the `targetNamespace` of that schema.

9.4.8 EMF Extensions

The initialization of an **EAttribute** or **EReference** corresponding to a schema attribute can be further customized through the use of several additional attributes from the Ecore namespace. Except as otherwise noted, these extensions are generally applicable to all local and global attribute declarations and attribute references. In cases where one of these attributes is used on both a global attribute declaration and a local reference to it, the value specified on the local attribute reference takes precedence.

The `ecore:reference`, `ecore:opposite`, and `ecore:ignore` attributes, which are not discussed here, have the specific uses outlined in Sections 9.4.2 and 9.4.6.

Name

An `ecore:name` attribute can be used to explicitly set the **name** of the **EAttribute** or **EReference**, if the default name conversion is unacceptable.

<xsd:attribute name="..." ecore:name="MyName" ... />	**EAttribute** **name=**"MyName" ...

Default Value

An `ecore:default` attribute can be added to a local attribute declaration to specify the default value of the corresponding **EAttribute**. This would typically be used only if the attribute is required and hence is not permitted to have a schema-specified default.

`<xsd:attribute ecore:default="value" ... />`	**EAttribute** **defaultValueLiteral**="value" ...

Multiplicity

An `ecore:many` attribute can be used on an attribute of a list simple type to indicate that it should map to a multiplicity-many **EAttribute** with an **eType** corresponding to the list's item type.

`<xsd:attribute type="xsd:IDREFS"` `ecore:many="true" ... />`	**EAttribute** **upperBound**="-1" **eType**=".../XMLType#//IDREF" ...

If the list type has an `xsd:maxLength` or `xsd:length` facet, that value is used as the upper bound. If the list has an `xsd:minLength` or `xsd:length` facet, and the attribute is not optional, that value is used as the lower bound. Otherwise, default lower and upper bounds of 0 and -1 apply.

The `ecore:lowerBound` and `ecore:upperBound` attributes can be used to explicitly override the **lowerBound** and **upperBound** of the structural feature corresponding to any attribute declaration or reference, if the default mapping rules don't produce the desired result.

`<xsd:attribute ecore:lowerBound="1" ... />`	**EAttribute** **lowerBound**="1" ...
`<xsd:attribute ecore:upperBound="10" ... />`	**EAttribute** **upperBound**="10" ...

When an attribute maps to a multiplicity-many structural feature, `ecore:ordered` and `ecore:unique` attributes can be specified on the declaration or reference to set the feature's **ordered** and **unique** attributes.

`<xsd:attribute ecore:ordered="false" ... />`	**EAttribute** **ordered**="false" ...
`<xsd:attribute ecore:unique="true" ... />`	**EAttribute** **unique**="true" ...

Although `ecore:ordered` and `ecore:unique` are allowed on attributes that map to both **EAttribute**s and **EReference**s, their usefulness is currently limited, as was described in Section 5.3: basically, only setting **unique** on an **EAttribute** is meaningful.

Behavior

Several boolean attributes of an **EAttribute** or **EReference**, which specify how a feature stores and accesses its values, can be set directly, using Ecore-namespace attributes of the same name. These attributes, which include `ecore:unsettable`, `ecore:changeable`, `ecore:derived`, `ecore:transient`, `ecore:volatile`, and `ecore:resolveProxies`, can be used when the default mapping rules don't produce the desired result.

`<xsd:attribute ecore:unsettable="true" ... />`	**EAttribute** **unsettable**="true" …
`<xsd:attribute ecore:changeable="true" ... />`	**EAttribute** **changeable**="true" …
`<xsd:attribute ecore:derived="true" ... />`	∫**EAttribute** **derived**="true" …
`<xsd:attribute ecore:transient="true" ... />`	**EAttribute** **transient**="true" …
`<xsd:attribute ecore:volatile="true" ... />`	**EAttribute** **volatile**="true" …
`<xsd:attribute ecore:reference="..." ecore:resolveProxies="true" ... />`	**EAttribute** **resolveProxies**="true" …

Note that the last of these, `ecore:resolveProxies`, is valid only on an attribute that maps to an **EReference** (see Section 9.4.2).

Accessor Visibility

Four Ecore-namespace attributes, `ecore:suppressedGetVisibility`, `ecore:suppressedSetVisibility`, `ecore:suppressedIsSetVisibility`, and `ecore:suppressedUnsetVisibility`, can be used to add an accessor

suppressing GenModel-namespace **EAnnotation** to the structural feature. As described in Section 5.7, such an **EAnnotation** instructs the code generator to suppress one or more of the accessors that would normally appear in the interface generated for the feature's containing class.

`<xsd:attribute` `ecore:suppressedGetVisibility="true"` `ecore:suppressedSetVisibility="true"` `ecore:suppressedIsSetVisibility="true"` `ecore:suppressedUnsetVisibility="true"` `... />`	**EAttribute** … **EAnnotation** **source**="…/emf/2002/GenModel" **details**= "suppressedSetVisibility"→"true", "suppressedGetVisibility"→"true", "suppressedIsSetVisibility"→"true", "suppressedUnsetVisibility"→"true"

9.5 Element Declarations

Each schema element declaration maps to an **EAttribute** or **EReference** in the **EClass** corresponding to the complex type definition containing the element, or in the "DocumentRoot" **EClass** if the element is global.

An element declaration maps to an **EAttribute** if its type is simple (with the exception of the special cases described in Section 9.5.3). Otherwise, if the type is complex, it maps to an **EReference**. In either case, the attributes of the feature are initialized as follows:

- **name** = the `name` of the element converted, if necessary, to a proper Java field name
- **eType** = an **EDataType** or **EClass** corresponding to the element's type
- **lowerBound** = the `minOccurs` value of the element declaration multiplied by the `minOccurs` of any containing model groups, or 0 for a global element or an element nested in an `xsd:choice`
- **upperBound** = the `maxOccurs` value of the element declaration multiplied by the `maxOccurs` of any containing model groups, or -2 (unspecified) for a global element (see Section 9.5.7)
- **eAnnotations** = an extended metadata **EAnnotation**

If the type of the element is one of the predefined schema types, then the **eType** of the corresponding **EAttribute** is set to the corresponding **EDataType** from the XMLType model (see Section 9.9). If the element has a user-defined simple type, the **eType** is set to an **EDataType** created from the simple type as described in Section 9.2.

Otherwise, if the element declaration maps to an **EReference**, the **eType** is set to the **EClass** corresponding to the element's type. The **EReference**'s **containment** attribute is true, except in the cases described in Section 9.5.3.

The **details** map of the extended metadata **EAnnotation** contains the following entries:

- **key** = "name", **value** = the unaltered `name` of the element
- **key** = "kind", **value** = "element"

`<xsd:element name="mySimple"` `  type="xsd:string"` `  maxOccurs="unbounded" />`	**EAttribute** **name**="mySimple" **eType**="…/XMLType#//String" **lowerBound**=1 **upperBound**=-1 *(unbounded)* **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"mySimple", "kind"→"element"
`<xsd:element name="myComplex">` `  <xsd:complexType ... >` `  ...` `  </xsd:complexType>` `</xsd:element>`	**EReference** **name**="myComplex" **eType**="//MyComplexType" **lowerBound**=1 **upperBound**=1 **containment**=true **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"myComplex", "kind"→"element"

9.5.1 AnyType Element

In addition to the **EDataTypes** for all the XML Schema predefined simple types (see Section 9.9), the XMLType model also includes an **EClass**, named "AnyType", that corresponds to the `xsd:anyType` complex type. An element of type `xsd:anyType` maps to an **EReference**, but not of this type, as you might expect. Instead, the **eType** of the **EReference** is **EObject**, the base class for all EMF objects.

`<xsd:element name="..."` `  type="xsd:anyType"/>`	**EReference** **eType**="…/Ecore#//EObject" …

Using **EObject** as the type of the reference allows it to take an instance of any EMF object as its value, which is the intended behavior. The purpose of the "AnyType" **EClass** is to handle situations where an instance contains arbitrary XML content. For example, when processing wildcard content in "lax mode" with no metadata available, an instance of the "AnyType" **EClass**, which like every other **EClass** implicitly extends **EObject**, will be used as the value of the feature. Such an instance can represent any arbitrary XML element content including any attributes and mixed text that it might have.

9.5.2 ID Element

Note: The XML Schema specification recommends avoiding the use of xsd:ID as the type of an element declaration.

An element of type `xsd:ID`, or of any type derived from it, maps to an **EAttribute** whose type is the "ID" **EDataType** from the XMLType model (see Section 9.9). In addition, the **iD** attribute of the **EAttribute** is set to true.

`<xsd:element name="..." type="xsd:ID"/>`	**EAttribute** **eType**="…/XMLType#//ID" **iD**=true …

9.5.3 ID Reference or URI Element

Note: The XML Schema specification recommends avoiding the use of xsd:IDREF or xsd:IDREFS as the type of an element declaration.

Elements of type `xsd:IDREF` or `xsd:anyURI`, or of any derived simple type, are given the same special treatment as was described for attributes of these types in Section 9.4.2. By default, they are treated as ordinary elements of simple type and mapped to **EAttribute**s. When an `ecore:reference` is specified, they map to **EReference**s, instead.. Unlike attributes, however, elements can be repeated, so the **upperBound** of the **EReference** is not always 1, but is instead set according to the `maxOccurs` attribute of the element declaration.

`<xsd:element name="customer"` `type="xsd:anyURI"` `maxOccurs="10"` `ecore:reference="Customer"/>`	**EReference** **name**="customer" **eType**="//Customer" **upperBound**=10 **containment**=false **resolveProxies**=true …

An `ecore:opposite` can also be specified to indicate that the relationship described by the element is bidirectional, just like for an attribute.

The `xsd:IDREFS` case is a little more complicated. This is because the `xsd:IDREFS` type represents multiple references and can, itself, be repeated (i.e., `maxOccurs` might be greater than 1). So, in this case the **EReference**'s **containment** attribute is set to true and its **eType** is set to an additional holder **EClass**, instead of to the type specified by the `ecore:reference` attribute.

`<xsd:element name="customers"` `  type="xsd:IDREFS"` `  ecore:reference="Customer"/>`	**EReference** **name**="customers" **eType**="//CustomersHolder" **containment**=true …

A holder **EClass**, "CustomersHolder" in this example, is automatically created for every element declaration of type `xsd:IDREFS` with an `ecore:reference` attribute specified. This **EClass** is initialized as follows:

- **name** = the `name` of the element converted, if necessary, to a proper Java class name, and with the string "Holder" appended
- **eReferences** = a single, multiplicity-many **EReference**
- **eAnnotations** = an extended metadata **EAnnotation**

The **details** map of the extended metadata **EAnnotation** for the **EClass** has the following entries:

- **key** = "name", **value** = the `name` of the element, with the string ":holder" appended
- **key** = "kind", **value** = "simple"

The **EReference** in the holder **EClass** is initialized as follows:

- **name** = "value"
- **eType** = the **EClass** corresponding to the type specified by the `ecore:reference` attribute
- **upperBound** = -1 (unbounded)
- **containment** = false
- **resolveProxies** = false

The details map of the extended metadata **EAnnotation** for the "value" **EReference** contains the following:

- **key** = "name", **value** = ":0"
- **key** = "kind", **value** = "simple"

`<xsd:element name="customers"` `  type="xsd:IDREFS"` `  ecore:reference="Customer"/>`	**EClass** **name**="CustomersHolder" **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"customers:holder", "kind"→"simple" … **EReference** **name**="value" **eType**="//Customer" **upperBound**=-1 *(unbounded)* **containment**=false **resolveProxies**=false **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→":0", "kind"→"simple" …

9.5.4 Nillable Element

A `nillable` element with `maxOccurs` equal to 1 maps to an **EAttribute** or **EReference** with **unsettable** set to true.

In the case of an element with simple type, if the type would ordinarily map to a primitive or enumerated type, then a wrapper **EDataType** is used as the **EAttribute**'s **eType**, instead. Recall from Sections 9.2.1 and 9.2.2 that an additional wrapper **EDataType** is produced for each primitive and enumerated type in a schema. In addition, the XMLType model defines wrappers for all of the built-in primitive schema types, as we will see in Section 9.9.

`<xsd:element type="xsd:int"` `  nillable="true" ... />`	**EAttribute** **eType**="…/XMLType#//IntObject" **unsettable**=true …

9.5.5 Default Value

When an element maps to an **EAttribute**, specifing a `default` value on the element sets the **EAttribute**'s **defaultValueLiteral**. The **EAttribute** is also unsettable in this case.

`<xsd:element name="message"` `  type="xsd:string"` `  default="hello world" ... />`	**EAttribute** **name**="message" **eType**="…/XMLType#//String" **defaultValueLiteral**="hello world" **unsettable**=true …

An element declaration without an explicit `default` value also maps to an unsettable **EAttribute** if the type has an intrinsic default value that is non-null (i.e., if the corresponding **eType** is an **EEnum** or an **EDataType** representing a primitive Java type).

`<xsd:element name="quantity"` `  type="xsd:int"/>`	**EAttribute** **name**="quantity" **eType**="…/XMLType#//Int" **unsettable**=true …

Section 9.2.2 described how a simple type restriction with enumeration facets can map to an ordinary **EDataType**, instead of an **EEnum**, when `ecore:enum="false"` is specified. If such a simple type maps to a primitive **EDataType** and is used as the type of an element, then the resulting **EAttribute** has its default value set, even if no explicit `default` is specified for the element. In this case, the **defaultValueLiteral** of the corresponding **EAttribute** is set to the first enumeration value of the simple type.

`<xsd:element name="oneThreeFive">` `  <xsd:simpleType ecore:enum="false">` `  <xsd:restriction base="xsd:int">` `  <xsd:enumeration value="1"/>` `  <xsd:enumeration value="3"/>` `  <xsd:enumeration value="5"/>` `  </xsd:restriction>` `  </xsd:simpleType>` `</xsd:element>`	**EDataType** **name**="OneThreeFiveType" **instanceClassName**="int" … **EAttribute** **name**="oneThreeFive" **eType**="//OneThreeFiveType" **defaultValueLiteral**="1" **unsettable**=true …

9.5.6 Qualified Element

If a local element declaration has qualified form, either explicitly declared with a `form="qualified"` attribute or inherited from an `xsd:schema` with `elementFormDefault="qualified"` (see Section 9.1.7), then the **details** map of the extended metadata **EAnnotation** for the corresponding feature contains an additional entry:

- **key** = "namespace", **value** = "##targetNamespace"

`<xsd:element form="qualified" ... />`	**EReference** ... **EAnnotation** **source=**"…/ExtendedMetaData" **details=** "namespace"→"##targetNamespace", …

9.5.7 Global Element

The **EAttribute** or **EReference** corresponding to a global element declaration is added to the package's "DocumentRoot" **EClass** as described in Section 9.1.5, unless it has an `ecore:ignore="true"` attribute, in which case it is ignored. The **upperBound** of the feature is set to -2 (unspecified). The extended metadata **EAnnotation** on the feature also includes exactly the same "namespace" **details** entry (with value "##targetNamespace") as in the case of a qualified element, which was described in Section 9.5.6.

`<xsd:schema ... >` `<xsd:element name="address"` `type="USAddress"/>` `...` `</xsd:schema>`	***EClass name**="DocumentRoot" ...* **EReference** **name**="address" **eType**="//USAddress" **upperBound**=-2 *(unspecified)* … **EAnnotation** **source=**"…/ExtendedMetaData" **details=** "namespace"→"##targetNamespace", …

9.5.8 Element Reference

An element reference (i.e., one with a `ref` attribute) maps to an **EAttribute** or **EReference** with a "namespace" entry in the **details** map of its extended metadata **EAnnotation**. If the reference is to a global element defined (or included) in the same schema, the value of this entry is "##targetNamespace".

`<xsd:complexType ...>` `  <xsd:sequence>` `  <xsd:element ref="address"/>` `  </xsd:sequence>` `</xsd:complexType>`	*EClass* ... EReference ... EAnnotation source="…/ExtendedMetaData" details= "namespace"→"##targetNamespace", ...

However, if the reference is to a global element from a different schema, then the value of the "namespace" entry is set instead to the `targetNamespace` of that schema.

9.5.9 Substitution Group

A `substitutionGroup` attribute in a global element declaration produces an additional entry in the details map of the extended metadata **EAnnotation** of the corresponding **EReference** or **EAttribute**:

- **key** = "affiliation", **value** = the value of the `substitutionGroup` attribute

`<xsd:schema ... >` `  <xsd:element name="staffComment"` `  substitutionGroup="comment">` `  <xsd:complexType>` `  ...` `  </xsd:complexType>` `  </xsd:element>` `  ...` `  <xsd:element name="comment"` `  type="xsd:string"/>` `</xsd:schema>`	*EClass name="DocumentRoot" ...* EReference name="staffComment" eType="//StaffCommentType" EAnnotation source="…/ExtendedMetaData" details="affiliation"→"comment", ... EAttribute name="comment" eType="…/XMLType#//String" ...

To be in a substitution group, an element must have the same type as the head element (the element named by the `substitutionGroup` attribute) or a type derived from it. In this example, the anonymously defined complex type of the "staffComment" element would extend the "comment" head element's simple string type.

Any reference to a substitution group's head element in a complex type produces an additional feature map **EAttribute** in the corresponding **EClass**, from which the ordinary **EAttribute** or **EReference** derives. By default, the **name** of the feature map **EAttribute** is formed by appending "Group" to the **name** of the ordinary feature. In addition, the extended metadata **EAnnotation** on the feature map **EAttribute** contains the following **details** entries:

- **key** = "name", **value** = the `name` of the element, with the suffix ":group" appended
- **key** = "kind", **value** = "group"

`<xsd:complexType name="PurchaseOrder">` `  <xsd:sequence>` `  ...` `  <xsd:element ref="comment"/>` `  ...` `  </xsd:sequence>` `</xsd:complexType>`	***EClass name**="PurchaseOrder" ...* **EAttribute** **name**="commentGroup" **eType**="…/Ecore#//EFeatureMapEntry" **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"comment:group", "kind"→"group", … … **EAttribute** **name**="comment" **eType**="…/XMLType#//String" **volatile**=true **transient**=true **derived**=true *(from "commentGroup")* **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"comment", "kind"→"element", "group"→"comment:group", … …

This feature-map-based implementation is required to allow instances of the substitution elements to be serialized in an XML document without using an `xsi:type` attribute. The same pattern is also used for a reference to an `abstract` global element, as such an element is prohibited from being used

directly, requiring that substitution elements be serialized in its place. The derived feature is also non-changeable in the abstract case.

If the reference to a head or abstract element is nested within a schema component for which a feature map would already be produced (e.g., if the containing complex type is `mixed`), the resulting feature map **EAttribute** then derives from that other feature map.

It is possible to specify a different **name** for the feature map **EAttribute** as the value of an `ecore:featureMap` attribute on the head element declaration or reference. Or, if you don't really need the ability to serialize elements from the substitution group, you can disable the feature-map-based implementation by specifying `ecore:featureMap=""`.

`<xsd:element ref="comment"` `  ecore:featureMap=""/>`	**EAttribute** **name**="comment" **eType**="…/XMLType#//String" **volatile**=false …

In fact, even when an element is not the head of a substitution group, an `ecore:featureMap` attribute can be used to introduce the feature-map-based structure that we have seen in this section.

<xsd:complexType name="PurchaseOrder"> *<xsd:sequence>* ... <xsd:element name="address" type="USAddress" ecore:featureMap="addressGroup"/> ... *</xsd:sequence>* *</xsd:complexType>*	***EClass name**="PurchaseOrder" …* **EAttribute** **name**="addressGroup" **eType**="…/Ecore#//EFeatureMapEntry" **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"address:group", "kind"→"group", … … **EReference** **name**="comment" **eType**="…//USAddress" **volatile**=true **transient**=true **derived**=true *(from "addressGroup")* **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"address", "kind"→"element", "group"→"#address:group", … …

9.5.10 EMF Extensions

The initialization of an **EAttribute** or **EReference** corresponding to an element can be further customized through the use of several additional attributes from the Ecore namespace. Except as otherwise noted, these extensions are generally applicable to all local and global element declarations and element references. In cases where one of these attributes is used on both a global element declaration and a local reference to it, the value specified on the local element reference takes precedence.

The `ecore:reference`, `ecore:ignore`, and `ecore:featureMap` attributes, which are not discussed here, have the specific uses outlined in Sections 9.5.3, 9.5.7, and 9.5.9.

Name

An `ecore:name` attribute can be used to explicitly set the **name** of the **EAttribute** or **EReference**, if the default name conversion is unacceptable.

`<xsd:element name="..."` `  ecore:name="MyName" ... />`	**EReference** **name**="MyName" ...

Multiplicity

The `ecore:lowerBound` and `ecore:upperBound` attributes can be used to explicitly override the **lowerBound** and **upperBound** of the structural feature corresponding to any element declaration or reference, if the default mapping rules don't produce the desired result.

`<xsd:element ecore:lowerBound="1" ... />`	**EReference** **lowerBound**="1" ...
`<xsd:element ecore:upperBound="10" ... />`	**EReference** **upperBound**="10" ...

When an element maps to a multiplicity-many structural feature, `ecore:ordered` and `ecore:unique` attributes can be specified on the declaration or reference to set the feature's **ordered** and **unique** attributes.

`<xsd:element ecore:ordered="false" ... />`	**EAttribute** **ordered**="false" …
`<xsd:element ecore:unique="true" ... />`	**EAttribute** **unique**="true" …

Although `ecore:ordered` and `ecore:unique` are allowed on elements that map to both **EAttribute**s and **EReference**s, their usefulness is currently limited, as was described in Section 5.3: basically, only setting **unique** on an **EAttribute** is meaningful.

Behavior

Several boolean attributes of an **EAttribute** or **EReference**, which specify how a feature stores and accesses its values, can be set directly, using Ecore-namespace attributes of the same name. These attributes, which include `ecore:unsettable`, `ecore:changeable`, `ecore:derived`, `ecore:transient`, `ecore:volatile`, and `ecore:resolveProxies`, can be used when the default mapping rules don't produce the desired result.

`<xsd:element` `ecore:unsettable="true" ... />`	**EReference** **unsettable**="true" …
`<xsd:element` `ecore:changeable="true" ... />`	**EReference** **changeable**="true" …
`<xsd:element` `ecore:derived="true" ... />`	**EReference** **derived**="true" …
`<xsd:element` `ecore:transient="true" ... />`	**EReference** **transient**="true" …
`<xsd:element` `ecore:volatile="true" ... />`	**EReference** **volatile**="true" …
`<xsd:element` `ecore:resolveProxies="true" ... />`	**EReference** **resolveProxies**="true" …

Note that the last of these, `ecore:resolveProxies`, is only valid on elements that map to **EReference**s, while the rest are valid on all elements.

Opposite

For any element that maps to an **EReference**, an `ecore:opposite` attribute can be used to make the relationship bidirectional and specify an **eOpposite EReference**. If the reference is non-containment (see Section 9.5.3), the `ecore:opposite` attribute identifies an attribute or element declaration in the target complex type, as described in Section 9.4.2. Otherwise, it simply specifies the name of a type-safe container reference to be added automatically to the target **EClass**.

`<xsd:element name="items"` `  type="Item"` `  maxOccurs="unbounded"` `  ecore:opposite="order"/>`	**EReference** **name**="items" **eType**="//Item" **upperBound**=-1 *(unbounded)* **containment**=true **eOpposite**="//Item/order" …

Accessor Visibility

Four Ecore-namespace attributes, `ecore:suppressedGetVisibility`, `ecore:suppressedSetVisibility`, `ecore:suppressedIsSetVisibility`, and `ecore:suppressedUnsetVisibility`, can be used to add an accessor suppressing GenModel-namespace **EAnnotation** to the structural feature. As described in Section 5.7, such an **EAnnotation** instructs the code generator to suppress one or more of the accessors that would normally appear in the interface generated for the feature's containing class.

`<xsd:element` `  ecore:suppressedGetVisibility="true"` `  ecore:suppressedSetVisibility="true"` `  ecore:suppressedIsSetVisibility="true"` `  ecore:suppressedUnsetVisibility="true"` `  … />`	**EReference** … **EAnnotation** **source**="…/emf/2002/GenModel" **details**= "suppressedSetVisibility"→"true", "suppressedGetVisibility"→"true", "suppressedIsSetVisibility"→"true", "suppressedUnsetVisibility"→"true"

9.6 Model Groups

XML Schema model groups (`xsd:sequence`, `xsd:choice`, and `xsd:all`) with `maxOccurs` equal to 1 (the default) produce no corresponding elements in the Ecore model. These constructs simply serve to aggregate the elements under them. In Ecore, the **EClass** corresponding to the containing complex type already provides this aggregation function for its features. The only case requiring special treatment is when a model group is allowed to repeat, as described in the following section.

9.6.1 Repeating Model Group

A repeating `xsd:sequence` or `xsd:choice` model group[5] (one with `maxOccurs` greater than 1) produces a feature map **EAttribute** in the **EClass** corresponding to the complex type definition containing the group. This is to represent the kind of cross-feature ordering described in Chapter 8. The **EAttribute** is initialized as follows:

- **name** = "group"
- **lowerBound** = the `minOccurs` value of the model group multiplied by the `minOccurs` of any containing model groups, or 0 if the group is nested in an `xsd:choice`
- **upperBound** = the `maxOccurs` value of the model group multiplied by the `maxOccurs` of any containing model groups
- **eAnnotations** = an extended metadata **EAnnotation**

The **details** map of the extended metadata **EAnnotation** contains the following entries:

- **key** = "name", **value** = the **name** of the **EAttribute**, followed by ":" and its feature ID
- **key** = "kind", **value** = "group"

`<xsd:choice maxOccurs="unbounded">` `  ...` `</xsd:choice>`	**EAttribute** **name**="group" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"group:0", "kind"→"group"

5. A repeating `xsd:all` model group is not a valid XML Schema construct.

All other **EReference**s and **EAttribute**s corresponding to element declarations in the model group have derived implementations that delegate to the feature map:

`<xsd:choice maxOccurs="unbounded">` `  <xsd:element name="priorityOrders"` `  type="PurchaseOrder"/>` `  ...` `</xsd:choice>`	**EAttribute** **name**="priorityOrders" **volatile**=true **transient**=true **derived**=true *(from "group")* ... **EAnnotation** **source**="…/ExtendedMetaData" **details**="group"→"#group:0", …

If the repeating model group is nested within a schema component for which a feature map would already be produced (e.g., if the containing complex type is `mixed`), the group's corresponding feature map **EAttribute** then derives from that other feature map.

An `ecore:featureMap` attribute can be added to the model group to override the default name of the corresponding feature map **EAttribute**.

`<xsd:choice maxOccurs="unbounded"` `  ecore:featureMap="choices">` `  ...` `</xsd:choice>`	**EAttribute** **name**="choices" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* **EAnnotation** **source**="…/ExtendedMetaData" **details**="name"→"choices:0", "kind"→"group"

Alternatively, if order preservation among the elements in the group is not desired, the feature map implementation can be suppressed by specifying `ecore:featureMap=""`. which produces the ordinary, non-derived implementation pattern for the elements in the group.

`<xsd:choice maxOccurs="unbounded"` `  ecore:featureMap="">` `  ...` `</xsd:choice>`	*No feature map attribute*

Finally, an `ecore:featureMap` attribute can also be used on a *non-repeating* model group to introduce a feature map implementation. The common use of this is to provide order preservation in an `xsd:all` group. By definition, an

`xsd:all` group is one that places no restriction on the order in which instances of its elements can appear. By default, EMF interprets this as meaning serialization order is irrelevant, so it does not use a feature map to implement the `xsd:all` group. The other possible interpretation is that the elements can appear in any order, but the order they're in is important and must be maintained. If this is the desired behavior, an `ecore:featureMap` attribute can be added to override the simpler default mapping and produce a feature map **EAttribute** for the group.

`<xsd:all ecore:featureMap="allGroup">` `  ...` `</xsd:all>`	**EAttribute** **name**="allGroup" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"allGroup:0", "kind"→"group"

9.6.2 Repeating Model Group Reference

The feature map **EAttribute** corresponding to a repeating reference to a model group definition (`xsd:group`) has its **name** set to that of the model group definition, instead of "group" as described in Section 9.6.1. The name is converted to a proper Java field name if necessary.

`<xsd:group name="content">` `  <xsd:choice>` `  ...` `  </xsd:choice>` `</xsd:group>` `...` `<xsd:complexType name="...">` `  <xsd:group ref="content"` `  maxOccurs="unbounded"/>` `</xsd:complexType>`	***EClass name***="…" **EAttribute** **name**="content" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"content:0", "kind"→"group"

An `ecore:name` attribute can also be added to the group definition to override the **name** of the feature map **EAttribute**.

<xsd:group name="content" ecore:name="orders"> <xsd:choice> ... </xsd:choice> </xsd:group> ... <*xsd:complexType* name="..."> <xsd:group ref="content" maxOccurs="unbounded"/> </*xsd:complexType*>	***EClass name***="…" **EAttribute** **name**="orders" **eType**="…/Ecore#//EFeatureMapEntry" **upperBound**=-1 *(unbounded)* **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→"orders:0", "kind"→"group"

Note that the same effect could be achieved with an `ecore:featureMap` attribute on the `xsd:choice` itself. If both are specified, the `ecore:featureMap` attribute takes precedence.

9.7 Wildcards

Element and attribute wildcards are represented in Ecore by feature map **EAttribute**s, so as to be able to accommodate values of structural features corresponding to any elements and attributes.

9.7.1 Element Wildcard

An element wildcard (`xsd:any`) maps to a feature map **EAttribute** in the **EClass** corresponding to the complex type definition containing the wildcard. The **EAttribute** is initialized as follows:

- **name** = "any"
- **lowerBound** = 0 if the wildcard is nested in an `xsd:choice` model group; otherwise, the value of the `minOccurs` attribute multiplied by the `minOccurs` of any containing model groups
- **upperBound** = the value of the `maxOccurs` attribute multiplied by the `maxOccurs` of any containing model groups
- **eAnnotations** = an extended metadata **EAnnotation**

The case where **upperBound** is 1 is somewhat special. It is still implemented using a feature map, rather than a feature map entry; however, the feature map is permitted to contain just a single entry.

The **details** map of the **EAttribute**'s extended metadata **EAnnotation** contains the following entries:

- **key** = "name", **value** = ":" followed by the feature ID of the **EAttribute**
- **key** = "kind", **value** = "elementWildcard"
- **key** = "wildcards", **value** = the value of the `namespace` attribute ("##any" by default)
- **key** = "processing", **value** = the value of the `processContents` attribute ("strict" by default)

The value of the "processing" entry determines how to handle unrecognized elements when loading an instance document. If it is "strict", then they are not allowed (metadata must be available for all elements). If it is "lax" or "skip", then metadata will be demand created for such elements, and instances of the "AnyType" **EClass** (described in Section 9.5.1) will be used to represent their contents.

`<xsd:any namespace="##other"` `  maxOccurs="unbounded"/>`	**EAttribute** **name**="any" **eType**="…/Ecore#//EFeatureMapEntry" **lowerBound**=1 **upperBound**=-1 *(unbounded)* **EAnnotation** **source=**"…/ExtendedMetaData" **details**="name"→":0", "kind"→"elementWildcard", "wildcards"→"##other", "processing"→"strict"

If a wildcard is nested within a schema component for which a feature map would already be produced (e.g., if the containing complex type is `mixed`), the wildcard's corresponding feature map **EAttribute** then derives from the containing feature map.

9.7.2 Attribute Wildcard

An attribute wildcard (`xsd:anyAttribute`) also maps to a feature map **EAttribute** in the **EClass** corresponding to the complex type definition containing the wildcard. The **EAttribute** is initialized as follows:

- **name** = "anyAttribute"
- **lowerBound** = 0
- **upperBound** = -1 (unbounded)
- **eAnnotations** = an extended metadata **EAnnotation**

The **details** map of the extended metadata **EAnnotation** contains the following entries:

- **key** = "name", **value** = ":" followed by the feature ID of the **EAttribute**
- **key** = "kind", **value** = "attributeWildcard"
- **key** = "wildcards", **value** = the value of the `namespace` attribute ("##any" by default)
- **key** = "processing", **value** = the value of the `processContents` attribute ("strict" by default)

The value of the "processing" entry determines how to handle unrecognized attributes when loading an instance document. If it is "strict", then they are not allowed (metadata must be available for all attributes). If it is "lax" or "skip", then metadata will be demand created for such attributes.

`<xsd:anyAttribute processContents="lax"/>`	**EAttribute** **name**="anyAttribute" **eType**="…/Ecore#//EFeatureMapEntry" **lowerBound**=0 **upperBound**=-1 *(unbounded)* **EAnnotation** **source**="…/ExtendedMetaData" **details**="name"→":1", "kind"→"attributeWildcard", "wildcards"→"##any", "processing"→"lax"

If a wildcard is nested within a schema component for which a feature map would already be produced (e.g., if the containing complex type is `mixed`), the wildcard's corresponding feature map **EAttribute** then derives from the containing feature map.

9.7.3 EMF Extensions

There is only one Ecore-namespace attribute applicable to wildcards: `ecore:name`, which can be used to set the **name** of a wildcard **EAttribute** to something other than the default value of "any" or "anyAttribute".

`<xsd:any ecore:name="extension"/>`	**EAttribute** **name**="extension" **eType**="…/Ecore#//EFeatureMapEntry" …

9.8 Annotations

XML Schema annotations map to **EAnnotation**s in Ecore. More specifically, each `xsd:documentation` and `xsd:appinfo` in a schema component's `xsd:annotation` maps to an **EAnnotation** in the **eAnnotations** list of the corresponding Ecore element. Non-schema attributes on schema components are similarly represented as **EAnnotation**s in Ecore.

9.8.1 Documentation

An `xsd:documentation` element in a schema component's `xsd:annotation` maps to a particular GenModel-sourced **EAnnotation**, allowing its contents to be generated into the Javadoc comments of the corresponding Java code. As described in Section 5.7.1, such an **EAnnotation** has as its **source** "http://www.eclipse.org/emf/2002/GenModel". Its **details** map contains a single entry:

- **key** = "documentation", **value** = the contents of the `xsd:documentation` element

`<xsd:annotation>` `  <xsd:documentation xml:lang="en">` `  some information` `  </xsd:documentation>` `</xsd:annotation>`	**EAnnotation** **source**="…/emf/2002/GenModel" **details**= "documentation"→" some information "

A single **EAnnotation** is used to represent all the `xsd:documentation` elements in the `xsd:annotation` on a given schema component, should there be more than one. In this case, the value of the "documentation" entry is simply the concatenation of the individual documentation elements.

`<xsd:annotation>` `  <xsd:documentation xml:lang="en">` `  some information` `  </xsd:documentation>` `  <xsd:documentation xml:lang="en">` `  more information` `  </xsd:documentation>` `</xsd:annotation>`	**EAnnotation** **source**="…/emf/2002/GenModel" **details**= "documentation"→" some information more information "

Note that the contents of each `xsd:documentation` element is stored exactly as is, without removing any line breaks or other whitespace.

9.8.2 Appinfo

An `xsd:appinfo` element in a schema component's `xsd:annotation` maps to an **EAnnotation** whose **source** is the same as the `source` attribute of the `xsd:appinfo`, or null if none is provided. The **EAnnotation**'s **details** map contains a single entry:

- **key** = "appinfo", **value** = the contents of the `xsd:appinfo` element

`<xsd:annotation>` `  <xsd:appinfo source="http://myURI">` `  <info>hello</info>` `  </xsd:appinfo>` `</xsd:annotation>`	**EAnnotation** **source**="http://myURI" **details**="appinfo"→"<info>hello</info>"

Alternatively, an `ecore:key` attribute can be used to specify an arbitrary key for the **details** entry:

`<xsd:annotation>` `  <xsd:appinfo source="http://myURI"` `  ecore:key="info">hello</xsd:appinfo>` `</xsd:annotation>`	**EAnnotation** **source**="http://myURI" **details**="info"→"hello"

A single **EAnnotation** is used to represent all the `xsd:appinfo` elements with a given `source` on a schema component, should there be more than one. If each such element uses `ecore:key` to specify a different key, then each value appears in a separate **details** entry. The values of `xsd:appinfo` elements with the same source and key are concatenated into a single value.

9.8.3 Ignored Annotation

An `ecore:ignore` attribute can be added to an `xsd:annotation` to suppress the **EAnnotations** corresponding to all its `xsd:documentation` and `xsd:appinfo` children.

`<xsd:annotation ecore:ignore="true">` `  ...` `</xsd:annotation>`	*No **EAnnotation***

Alternatively, `ecore:ignore` can be specified on individual `xsd:documentation` or `xsd:appinfo` elements to suppress only their corresponding **EAnnotations**.

9.8.4 Non-schema Attribute

An attribute from a namespace other than the XML Schema namespace maps to an **EAnnotation** with **source** set to the the attribute's namespace.[6] The **details** map of the **EAnnotation** contains a single entry:

- **key** = the local name of the attribute, **value** = the value of the attribute

`<xsd:element xmlns:x="http://x"` `  x:a="b" .../>`	*EAttribute* ... **EAnnotation** **source**="http:///x" **details**="a"→"b"

9.9 Predefined Schema Simple Types

Each predefined XML Schema simple type maps to a corresponding built-in **EDataType** from the special XMLType model. This model defines a single package named "type" with namespace URI "http://www.eclipse.org/emf/2003/XMLType", which contains these simple type counterparts. Table 9.1 details the mapping between XML Schema simple types and XMLType **EDataType**s, including the instance class for each **EDataType**. As pointed out in Section 9.5.4, the model includes wrapper **EDataType**s for all the schema primitive types. These are provided for use in **EAttribute**s corresponding to nillable elements.

Table 9.1 Schema Simple Types

Schema Type	*EDataType*	*Instance Class*
anySimpleType	AnySimpleType	java.lang.Object
anyURI	AnyURI	java.lang.String
base64Binary	Base64Binary	byte[]
boolean	Boolean	java.lang.boolean
boolean (nillable="true")	BooleanObject	java.lang.Boolean
byte	Byte	byte
byte (nillable="true")	ByteObject	java.lang.Byte

6. Since attributes from the Ecore namespace are used specifically to customize the mapping from XML Schema to Ecore, they are not represented as an **EAnnotation** either.

Schema Type	EDataType	Instance Class
date	Date	java.lang.Object[7]
dateTime	DateTime	java.lang.Object
decimal	Decimal	java.math.BigDecimal
double	Double	double
double (nillable="true")	DoubleObject	java.lang.Double
duration	Duration	java.lang.Object
ENTITIES	ENTITIES	java.util.List
ENTITY	ENTITY	java.lang.String
float	Float	float
float (nillable="true")	FloatObject	java.lang.Float
gDay	GDay	java.lang.Object
gMonth	GMonth	java.lang.Object
gMonthDay	GMonthDay	java.lang.Object
gYear	GYear	java.lang.Object
gYearMonth	GYearMonth	java.lang.Object
hexBinary	HexBinary	byte[]
ID	ID	java.lang.String
IDREF	IDREF	java.lang.String
IDREFS	IDREFS	java.util.List
int	Int	int
int (nillable="true")	IntObject	java.lang.Integer
integer	Integer	java.math.BigInteger
language	Language	java.lang.String
long	Long	long
long (nillable="true")	LongObject	java.lang.Long

(continues)

7. Beginning in EMF 2.3, standard Java XML types (from the package `java.xml.datatype`) are used as the instance class for several of these data types. In particular, for **Date**, **DateTime**, **GDate**, **GMonth**, **GMonthDay**, **GYear**, **GYearMonth**, and **Time**, the instance class is `XMLGregorianCalendar`. The instance class of **Duration** is `Duration`. For **NOTATION** and **QName**, the instance class is `QName`. These XML types were introduced in Java 5.0, so they couldn't be used by EMF 2.2, which can run on Java 1.4, or earlier.

Table 9.1 Schema Simple Types (continued)

Schema Type	*EDataType*	*Instance Class*
Name	Name	java.lang.String
NCName	NCName	java.lang.String
negativeInteger	NegativeInteger	java.math.BigInteger
NMTOKEN	NMToken	java.lang.String
NMTOKENS	NMTOKENS	java.util.List
nonNegativeInteger	NonNegativeInteger	java.math.BigInteger
nonPositiveInteger	NonPositiveInteger	java.math.BigInteger
normalizedString	NormalizedString	java.lang.String
NOTATION	NOTATION	java.lang.Object
positiveInteger	PositiveInteger	java.math.BigInteger
QName	QName	java.lang.Object
short	Short	short
short (nillable="true")	ShortObject	java.lang.Short
string	String	java.lang.String
time	Time	java.lang.Object
token	Token	java.lang.String
unsignedByte	UnsignedByte	short
unsignedByte (nillable="true")	UnsignedByteObject	java.lang.Short
unsignedInt	UnsignedInt	long
unsignedInt (nillable="true")	UnsignedIntObject	java.lang.Long
unsignedLong	UnsignedLong	java.math.BigInteger
unsignedShort	UnsignedShort	int
unsignedShort (nillable="true")	UnsignedShortObject	java.lang.Integer

9.10 EMF Extensions

As described throughout this chapter, attributes from the Ecore namespace ("http://www.eclipse.org/emf/2002/Ecore") can be added to schema components to customize the mapping to Ecore. The following attributes are recognized:

- `ecore:changeable` on an attribute or element declaration specifies the value of the **changeable** attribute of the corresponding **EStructuralFeature** (see Sections 9.4.8 and 9.5.10).
- `ecore:constraints` on a simple or complex type definition adds an **EAnnotation** declaring named constraints to the corresponding **EClassifier** (see Sections 9.2.6 and 9.3.6).
- `ecore:default` on an attribute declaration specifies the value of the **defaultValueLiteral** attribute of the corresponding **EAttribute** (see Section 9.4.8).
- `ecore:derived` on an attribute or element declaration specifies the value of the **derived** attribute of the corresponding **EStructuralFeature** (see Sections 9.4.8 and 9.5.10).
- `ecore:documentRoot` on a schema is used to change the name of the document root **EClass** from the default, "DocumentRoot" (see Section 9.1.3).
- `ecore:enum` on a simple type definition with enumeration facets determines whether the type maps to an **EEnum** or to an **EDataType** with the facets recorded in its extended metadata annotation (see Section 9.2.2).
- `ecore:featureMap` on a model group or reference, an element declaration or reference, or a complex type, can be used to force or block the use of a feature map in the corresponding Ecore representation. By default, feature maps are used to implement mixed complex types (see Section 9.3.5), substitution groups and abstract elements (see Section 9.5.9), repeating model groups (see Section 9.6.1), and wildcards (see Section 9.7).
- `ecore:ignore` on a global attribute or element, facet, annotation, documentation or appinfo determines whether the component is excluded from the corresponding Ecore representation (see Sections 9.4.6, 9.5.7, 9.2.1 and 9.8.3).
- `ecore:implements` on a complex type definition specifies additional **eSuperTypes** for the corresponding **EClass** (see Section 9.3.6).
- `ecore:instanceClass` on a simple or complex type definition specifies the **instanceClassName** (i.e., Java class) of the corresponding **EClassifier** (see Sections 9.2.6 and 9.3.6).
- `ecore:interface` on a complex type definition specifies the value of the **interface** attribute of the corresponding **EClass** (see Section 9.3.6).
- `ecore:key` on an `xsd:appinfo` annotation component specifies the key for the corresponding **EAnnotation details** entry (see Section 9.8.2).

- `ecore:lowerBound` on an attribute or element declaration specifies the value of the **lowerBound** attribute of the corresponding **EStructuralFeature** (see Sections 9.4.8 and 9.5.10).
- `ecore:many` on an attribute declaration with a list simple type determines whether it maps to a multi-valued **EAttribute** of a type corresponding to the list's item type (see Section 9.4.8).
- `ecore:mixed` on a complex type definition that is not actually mixed specifies whether to use the feature-map-based Ecore representation for a mixed type anyway (see Section 9.3.5).
- `ecore:name` on any named schema component or on a wildcard can be used to override the default **name** of the corresponding **ENamedElement** (see Sections 9.2.2, 9.2.6, 9.3.6, 9.4.8, 9.5.10, 9.6.2, and 9.7.3).
- `ecore:nsPrefix` on a schema specifies the value of the **nsPrefix** attribute of the corresponding **EPackage** (see Section 9.1.5).
- `ecore:opposite` on an element or attribute declaration that maps to an **EReference** specifies the element or attribute corresponding to the reference's **eOpposite** (see Sections 9.4.2 and 9.5.10).
- `ecore:ordered` on an attribute or element declaration specifies the value of the **ordered** attribute of the corresponding **EStructuralFeature** (see Sections 9.4.8 and 9.5.10).
- `ecore:package` on a schema specifies the fully qualified Java package name for the corresponding **EPackage** (see Section 9.1.5).
- `ecore:reference` on an attribute or element declaration of type `xsd:IDREF`, `xsd:IDREFS`, or `xsd:anyURI` specifies the target type of the corresponding **EReference** (see Sections 9.4.2 and 9.5.3).
- `ecore:resolveProxies` on an attribute or element declaration specifies the value of the **resolveProxies** attribute of the corresponding **EReference** (see Sections 9.4.8 and 9.5.10).
- `ecore:serializable` on a simple type definition specifies the **serializable** attribute of the corresponding **EDataType** (see Section 9.2.6).
- `ecore:suppressedGetVisibility` on an attribute or element declaration adds a GenModel-namespace **EAnnotation** to the corresponding **EStructuralFeature**, specifying that the feature's `get()` accessor should be suppressed (see Sections 9.4.8 and 9.5.10).
- `ecore:suppressedIsSetVisibility` on an attribute or element declaration adds a GenModel-namespace **EAnnotation** to the corresponding **EStructuralFeature**, specifying that the feature's `isSet()` accessor should be suppressed (see Sections 9.4.8 and 9.5.10).

- `ecore:suppressedSetVisibility` on an attribute or element declaration adds a GenModel-namespace **EAnnotation** to the corresponding **EStructuralFeature,** specifying that the feature's `set()` accessor should be suppressed (see Sections 9.4.8 and 9.5.10).
- `ecore:suppressedUnsetVisibility` on an attribute or element declaration adds a GenModel-namespace **EAnnotation** to the corresponding **EStructuralFeature,** specifying that the feature's `unset()` accessor should be suppressed (see Sections 9.4.8 and 9.5.10).
- `ecore:transient` on an attribute or element declaration specifies the value of the **transient** attribute of the corresponding **EStructuralFeature** (see Sections 9.4.8 and 9.5.10).
- `ecore:unique` on an attribute or element declaration specifies the value of the **unique** attribute of the corresponding **EStructuralFeature** (see Sections 9.4.8 and 9.5.10).
- `ecore:unsettable` on an attribute or element declaration specifies the value of the **unsettable** attribute of the corresponding **EStructuralFeature** (see Sections 9.4.8 and 9.5.10).
- `ecore:upperBound` on an attribute or element declaration specifies the value of the **upperBound** attribute of the corresponding **EStructuralFeature** (see Sections 9.4.8 and 9.5.10).
- `ecore:value` on an enumeration facet specifies the **value** attribute of the corresponding **EEnumLiteral** (see Section 9.2.2).
- `ecore:volatile` on an attribute or element declaration specifies the value of the **volatile** attribute of the corresponding **EStructuralFeature** (see Sections 9.4.8 and 9.5.10).

These attributes are described in more detail in the referenced sections.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X – Z