CHAPTER 2

Introducing EMF

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

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

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

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 value-add," you say.

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

Maybe a class diagram or two to fill out the documentation a bit, but other than that it simply doesn't help. So, to appease the boss, you produce this UML diagram (see Figure 2.1)[2]:

Figure 2.1 UML diagram of interfaces.

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

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

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

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

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 (that is, Java interfaces, UML diagram, or XML Schema), and generated the others from it. Even better, you start to wonder if maybe there's even enough information in this "model" to generate the Java implementation of the interfaces.

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

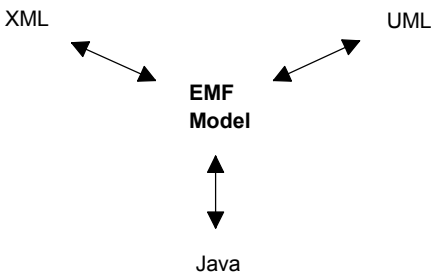

Figure 2.2 EMF unifies Java, XML, and UML.

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

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

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/modeling work from the low-level implementation programming, it brings them together as two well-integrated parts of the same job. Often, especially with large applications, this kind of separation is still desirable, but with EMF the degree to which it is done is entirely up to you.

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

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

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, the designers of EMF truly feel that its exact position in the middle represents the right level of modeling, at this point in the evolution of software development technology. We believe that it mixes just the right amount of modeling with programming to maximize the effectiveness of both. We must admit, though, that standing in the middle and arguing out of both sides of our mouths can get tiring!

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

2.3 Defining the Model

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

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

3. **items**, which is a UML association end or reference, a `get()` method in Java, and in XML Schema, a nested element declaration of another complex type.

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

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

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 also a meta-metamodel. People often get confused when talking about meta-metamodels (metamodels in general, for that matter), but the concept is actually quite simple. A metamodel is simply the model of a model, and if that model is itself a metamodel, then the metamodel is in fact a meta-metamodel.[4] Got it? If not, I wouldn't worry about it, since it's really just an academic issue anyway.

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

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

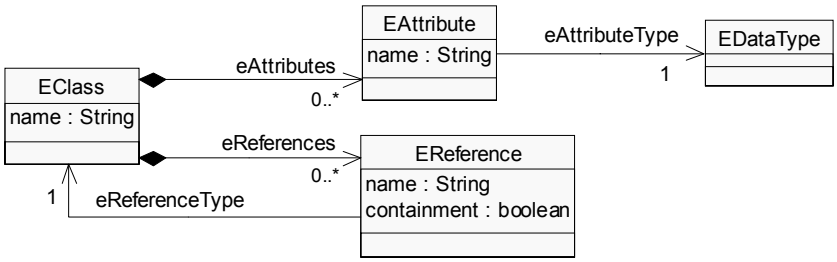

Figure 2.3 A simplified subset of the Ecore model.

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

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

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

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

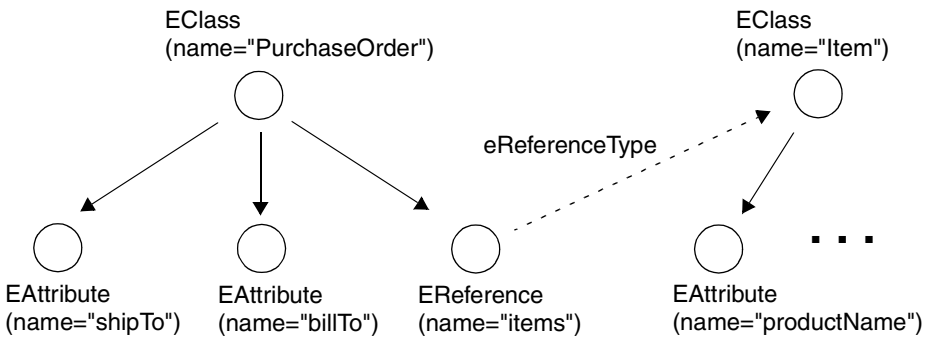

Figure 2.4 The purchase order Ecore instances.

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

2.3.2 Creating and Editing the Model

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

1. **Direct Ecore Editing**—EMF's simple tree-based sample Ecore editor and Omondo's (free) EclipseUML graphical editor[5] are examples.
2. **Import from UML**—The EMF Project Wizard provides this option for Rational Rose (*.mdl* files) only. The reason Rose has this special status is because it's the tool that we used to "bootstrap" the implementation of EMF itself.
3. **Export from UML**—This is essentially the same as option 2, except the conversion is invoked from the UML tool, instead of from the EMF Project Wizard.

As you might imagine, option 1 is the most desirable. With it, there is no import or export step in the development process. You simply edit the model and then generate. Also, unlike the other options, you don't need to worry

5. You can download EclipseUML from the Omondo Web site at *www.omondo.com*.

about the core model being out of sync with the tool's own native model. The other two approaches require an explicit reimport or reexport step whenever the UML model changes.

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

2.3.3 XMI Serialization

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

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

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

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

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

```
<?xml version="1.0" encoding="ASCII"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
                name="po" nsURI="http:///com/example/po.ecore"
                nsPrefix="com.example.po">
```

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

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

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

2.3.4 Java Annotations

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

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

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

```
/**
 * @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 it returns a `List`, we need to specify the target type of the reference. We also need to specify `containment="true"` to indicate that we want purchase orders to be a container for their items and serialize them as children.

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

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. A core model can be created from any of at least three sources: a UML model, an XML Schema, or annotated Java interfaces.

3. Java implementation code and, optionally, other forms of the model can be generated from a core model.

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

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

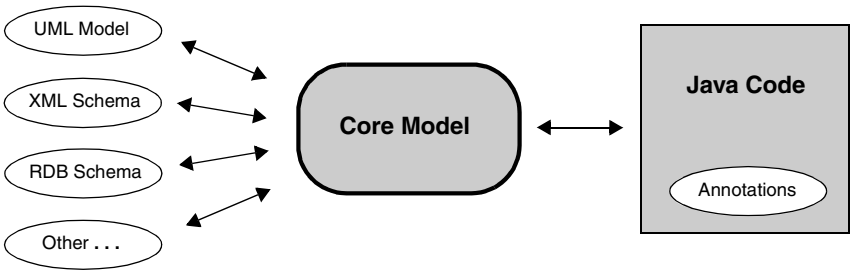

Figure 2.5 The core model and its sources.

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

2.4.1 Generated Model Classes

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

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

and a corresponding implementation class:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The generated package (for example, `POPackage`) provides convenient accessors for all the Ecore metadata for the model. You may already have noticed, in the code fragment on page 23, the use of `POPackage.PURCHASE_ORDER__SHIP_TO`, a static `int` constant representing the **shipTo** attribute. The generated package also includes convenient accessors for the **EClass**es, **EAttribute**s, and **EReference**s. 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[8] class (for example, `POAdapterFactory`) for the model. This convenient base class can be used to implement adapter factories that need to create type-specific adapters. For example, a `PurchaseOrderAdapter` for **PurchaseOrder**s, an `ItemAdapter` for **Item**s, and so on.
2. A convenience switch class (for example, `POSwitch`) that implements a "switch statement"-like callback mechanism for dispatching based on an object's type (that is, its **EClass**). The adapter factory class, as just described, uses this switch class in its implementation.
3. A plug-in manifest file, so that the model can be used as an Eclipse plug-in.
4. An XML Schema for the model.

If all you're interested in is generating a model, then this is the end of the story. However, as we'll see in Chapters 3 and 4, the EMF generator can, using the EMF.Edit extensions to the base EMF framework, generate adapter classes

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

that enable viewing and command-based, undoable editing of a model. It can even generate a working editor for your model. We will talk more about EMF.Edit and its capabilities in the following chapter. For now, we just stick to the basic modeling framework itself.

2.4.3 Regeneration and Merge

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

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

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

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

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

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

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

2.4.4 The Generator Model

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

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

The significance of all this is that the EMF generator runs off of a generator model instead of a core model; it's actually a generator model editor.[9] When you use the generator, you'll be editing a generator model, which in turn indirectly accesses the core model from which you're generating. As you'll see in Chapter 4 when we walk through an example of using the generator, there are two model resources (files) in the project: a *.ecore* file and a *.genmodel* file. The *.ecore* file is an XMI serialization of the core model, as we saw in Section 2.3.3. The *.genmodel* file is a serialized generator model with cross-document references to the *.ecore* file. Figure 2.6 shows the conceptual picture.

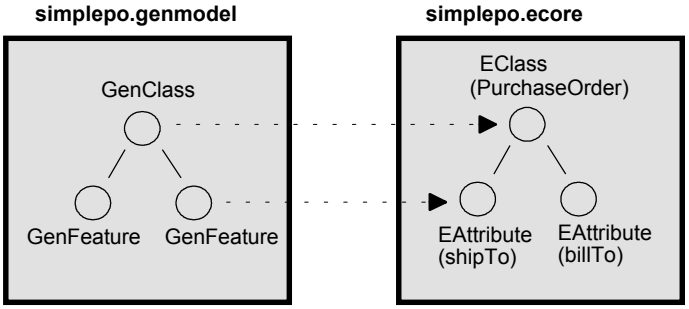

Figure 2.6 The *.genmodel* and *.ecore* files.

9. 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.

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

2.5 The EMF Framework

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

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 (that is, support additional interfaces without subclassing) of the object they're attached to. An adapter, as a simple observer, can be attached to any `EObject` (for example, `PurchaseOrder`) by adding to its adapter list like this:

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

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

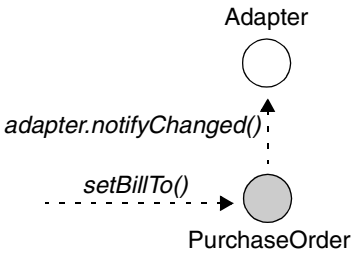

Figure 2.7 Calling the `notifyChanged()` method.

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

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

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

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

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

As you can see, an adapter must be attached to each individual `EObject` that it wants to observe. Sometimes, you may be interested in being informed of state changes to any object in a containment hierarchy, a resource, or even any of a set of related resources. Rather than requiring you to walk through the hierarchy and attach your observer to each object, the EMF framework

provides a very convenient adapter class, `EContentAdapter`, which can be used for this purpose. It can be attached to a root object, a resource, or even a resource set, and it will automatically attach itself to all the contents. It will then receive notification of state changes to any of the objects and it will even respond to content change notifications itself, by attaching or detaching itself as appropriate.

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

2.5.2 Object Persistence

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

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

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

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

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

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

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

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

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

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

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

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

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

A `ResourceSet`, as its name implies, is a set of resources that are accessed together, in order to allow for potential cross-document references among them. It's also the factory for its resources. So, to complete our example, we would create the resource, add the purchase order to it, and then save it something like this[11]:

```
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);
```

10. 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'll discuss this issue in more detail in Chapter 9.

11. 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 may 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 13.

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

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

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

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

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

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

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

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

Figure 2.8 **items** as a simple reference.

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

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

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

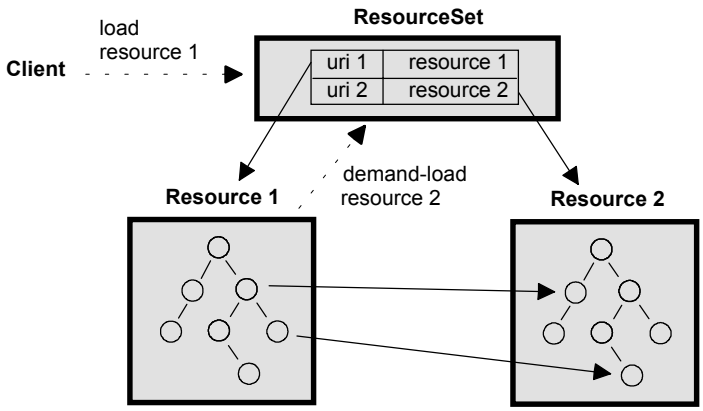

Figure 2.9 Resource set demand-loading of resources.

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

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

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 accessors like this:

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

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

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

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 simply create the core model at runtime, something like this:

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

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

EClass itemClass = EcoreFactory.eINSTANCE.createEClass();
itemClass.setName("Item");
poPackage.getEClassifiers().add(itemClass);

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

// and so on ...
```

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

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

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

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

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

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

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

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

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

2.5.5 Foundation for Data Integration

The last few sections have shown various features of the EMF framework that support sharing of data. Section 2.5.1 described how change notification is an intrinsic property of every EMF object, and how adapters can be used

to support open-ended extension. In Section 2.5.2, we showed how the EMF persistence framework uses `Resources` and `ResourceSets` to support cross-document referencing, demand-loading of documents, and arbitrary persistent forms. Finally, in Sections 2.5.3 and 2.5.4 we saw how EMF supports generic access to EMF models, including ones that may be partially or completely dynamic (that is, without generated implementation classes).

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

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

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 UML

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

EMF is concerned with only one aspect of UML, class modeling. This focus is in no way a rejection of UML's holistic approach. Rather, it is a starting

point, based on the pragmatic realization that the task of translating the ideas that can be expressed in various UML diagrams into concrete implementations is very large and very complex.

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

2.6.2 MOF

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

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

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

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

2.6.3 XMI

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

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

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

2.6.4 MDA

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

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

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