EMF Users’ Guide
Draft 1.0

Last updated on 9/17/02

1.0 Overview

The Eclipse Modeling Framework, EMF, is a Java framework and code generation facility for building tools and other applications based on a structured model. EMF provides a mechanism to easily create, save, and restore instances of the classes in your model. This makes it very easy to share data across different applications.

This document describes the basic steps for using EMF, including how to define your model, how to generate and customize your application source code, and how to manipulate and save instance data.

- Section 2.0, “Concepts,” on page 1 outlines some of the key concepts essential to the understanding of EMF.
- Section 3.0, “Tasks,” on page 6 gives examples for how to use EMF to accomplish some commonly encountered tasks.
- Section 4.0, “Quick Reference,” on page 56 is a short reference guide to the APIs used by an EMF application.

This document is a work in progress. Some sections have not yet been completed. Areas where changes and additions are anticipated are identified using a <tbd> tag. For example, *<tbd> This information will be provided in a future draft of the EMF User’s Guide.</tbd>*

2.0 Concepts

We first present some of the key concepts upon which EMF is based. This section is intended to introduce these fundamental concepts at a high level and to give an indication of how they relate to EMF. The concepts that are covered here are:

- Section 2.1, “Modeling,” on page 2
- Section 2.2, “Code Generation,” on page 2
- Section 2.3, “Serialization and Loading,” on page 3
- Section 2.4, “XMI,” on page 5
- Section 2.5, “Observers and Notifiers,” on page 6

2.1 Modeling

A model is an abstract representation of the data used by an application. A model may be expressed simply as an informal description of the data or it may be described precisely using any of a number of formal mechanisms, for example:

- UML created using a visual modeling tool
- XML that conforms to a schema that expresses all the necessary elements of the model
- Java interfaces that may have additional annotations to capture special information about the model that is not expressible directly in Java

One of the advantages of using a formal definition of a model is that EMF can then be used to assist in the automatic generation and maintenance of application code based on the model. This in turn facilitates the creation of multiple applications that can communicate easily because they share a common view of the underlying data.

A model is constructed from classes that describe objects in an application's domain. Each class may have attributes, associations, and operations associated with it. Usually, the classes in a model are grouped into one or more packages.

2.1.1 The Ecore Model

EMF is itself based on a model called Ecore. The Ecore model is the meta-model for all application models that are handled by EMF. (A "meta-model" is a model that represents other models.)

The root of the Ecore Model is *`EObject`*. All EMF objects implement the *`EObject`* interface.

The classes of the Ecore model include *`EPackage`*, *`EFactory`*, *`EClass`*, and *`EStructuralFeature`*. ("Appendix A - The Ecore Model" on page 76 gives a more complete description of the Ecore Model.)

If your model consists of a package that contains two classes, the EMF representation of that model will consist of a single instance each of *`EPackage`* and *`EFactory`* plus two instances of *`EClass`*, along with however many instances of *`EStructuralFeature`* are needed to represent the attributes and associations that belong to your classes. When your application creates instances of your classes, each of those instances will implement the *`EObject`* interface.

2.2 Code Generation

Code generation is the process of converting your model into the Java source code for your application. In many cases, after you generate your Java classes you will want to add methods or modify the generated methods. EMF enables you to do this in a such a way that if you later make a change to your model and regenerate, the code generation process will preserve your changes.

EMF provides two separate code generation facilities:

- Basic Code Generation creates Java interfaces and classes that represent the elements of your model. The generated APIs enable you to create instances of your classes and access the structural features of each class.
- EMF.Edit Code Generation creates a simple graphical editor that can be used to create, update, view, load, and store the Java classes that are generated by basic code generation.

2.2.1 Basic Code Generation

The result of EMF basic code generation is two or three Java packages for each package in your model. One of these Java packages consists entirely of Java interfaces that define the APIs to access instances of the classes in your model. The other Java package consists of Java classes that implement those interfaces. The third package, which is optional, contains a generated adapter factory and a generated switch class. These classes are useful when implementing adapters.

The interface package includes one interface for each class in your model. These interfaces extend the `EObject` interface from the Ecore model. Each interface provides methods that enable you to access an instance of a class and to get and set the values for each of the features of that class.

The interface package also contains two additional interfaces: one that extends the `EPackage` interface and another that extends the `EFactory` interface. The `EPackage` interface provides methods for accessing the meta-data from your model (i.e., the `EClass` and `EStructuralFeature` objects that describe your classes) and the factory interface provides methods for creating instances of your data (i.e., the `EObject` objects that implement your classes.)

The implementation package has Java classes that implement all the interfaces described above.

2.2.2 EMF.Edit Code Generation

<tbd>To be done</tbd>

2.3 Serialization and Loading

Serialization is the process of writing your instance data into a standardized, persistent form, e.g., a file on your file system or a Web resource.

Loading (sometimes referred to as “deserialization”) is the process of reading the persistent form of the data to recreate instances of `EObject` in memory. In EMF, loading can be accomplished either through an explicit call to a load API or it can happen automatically whenever a reference to an `EObject` that has not yet been loaded is encountered.

The APIs that are used in EMF to control the loading and saving of objects are defined on the Resource and ResourceSet interfaces.

The default implementation of Resource is `XMIResourceImpl`, which results in serialization as XMI documents. The XMIResource interface provides additional APIs that enable you to control some of the behavior of the default serializer and loader. If you wish to serialize in some format other than the default XMI format, you can provide your own implementation of Resource.

2.3.1 Resource

A `Resource` is a collection of `EObject` objects that are serialized into a single `Stream`.

The `Resource` interface is defined in the package `org.eclipse.emf.ecore.resource`. It provides APIs that enable you to:

- Load a `Resource` from an `InputStream`.
- Save a `Resource` to an `OutputStream`.
- Access any messages that were generated during the load or save operation.
- Optionally keep track of whether any objects in a `Resource` have been modified. The default is not to keep track of modifications. Modification tracking can add significant runtime overhead.
- Register the default factory for creating `Resource` objects. If you choose to provide your own implementation of the `Resource` interface, for example, so you can use a serialization format other than XMI, you will need to define and register a factory for creating your implementation of `Resource`.

2.3.2 ResourceSet

A `ResourceSet` is a collection of `Resource` objects that may have cross-references among them.

The ResourceSet interface defines APIs that enable you to:

- Create a new Resource
- Look up an individual object and, if necessary, load the `Resource` in which it is contained.
- Set and get the URIConverter used to normalize URIs and resolve relative URIs.
- Set and get a `Resource.Factory.Registry`. (This registry enables you to provide alternative implementations of Resource and to have the appropriate implementation selected based on either the extension or the protocol of a given URI.)
- Get the list of registered `AdapterFactory` instances.

2.3.3 URIConverter

Uniform Resource Identifiers or URIs, as specified in http://www.ietf.org/rfc/rfc2396.txt, are used to uniquely identify resources and objects within resources. For example, when one object references another object that is located in a different resource, a URI is used to identify the referenced object.

Often it is convenient to have a URI that is expressed as a relative location. For example, if the URI refers to a file on a file system, it may be convenient to describe that file relative to some known location rather than as an absolute path. This would enable your application to run in different environments. Similarly, a URI may refer to a resource within your Eclipse workbench, or to an object on the Web. Expressing the URI as a relative value rather than an absolute location provides the flexibility that is needed to be able to share resources.

A URIConverter is used to resolve a relative URI into an absolute InputStream or OutputStream. The URIConverter also provides an API to normalize relative URIs. Normalization is used to determine if two different URIs in fact refer to the same underlying object.

EMF provides a default implementation of a URIConverter and also enables you to create and register your own implementation.

2.4 XMI

The XML Metadata Interchange (XMI) is the default serialization format used by EMF. This format is based on the XMI 2.0 specification from the OMG. This specification may be found at http://cgi.omg.org/cgi-bin/doc?ad/01-06-12.
The XML specification may be found at http://www.w3.org/TR/REC-xml.

2.4.1 XMIResource

An XMIResource is an extension to the Resource interface (See “Resource” on page 4.) that handles a resource whose contents are serialized as an XMI document. This is the default type of resource used by EMF.

The XMIResource interface provides APIs that enable you to:

- Access and modify the XMI IDs will be used when objects are serialized. (Note that IDs are optional. If an object does not have an ID, references to that object within a document are based on the relative position of the object. Using IDs can increase the size of your documents, so their use is not recommended.)
- Control whether the XMI documents are stored in zipped form. The default setting is to use unzipped files.
- Specify the XML encoding to be used when saving the resource

- Specify various save options:
    - Control whether the type of an element is written using “xmi:type” or “xsi:type”. The default is to use “xsi:type”
    - Control whether the encoded attribute style is used to serialize an attribute whose value is an EObject. When an attribute is serialized under this option, the value of an attribute is a QName URI pair, where the QName is optional, depending on whether the referenced object’s type is identical to the feature. When the option is not specified, an attribute whose value is an EObject is serialized as an element.
    - Determine the line width at which line breaks will be automatically added.
    - Determine whether the serialized document will begin with:

      ```
      <?xml version="1.0" encoding="encoding"?>
      ```
    - Control whether to skip processing for escape characters. This processing adds overhead that can be skipped if you know for sure that none of the values of your attributes contain a character that needs to be escaped. These characters are ampersand (‘&’), double-quote (‘”’), less-than(‘<‘), LF (‘\n’), CR (‘\r’), and tab (‘\t’)
    - Determine how dangling hrefs will be handled during save. A dangling href is a cross file reference where the target is not in a valid resource, which means that the URI for the target cannot be computed. The possible actions are to either throw an exception, discard them silently, or record them and continue.
- Specify various load options:
    - Control whether notifications are to be disabled during loading.

2.5 Observers and Notifiers

EMF provides a mechanism for attaching observers (also known as adaptors) to objects (sometimes referred to as notifiers.) The observers are informed of any changes to the notifiers to which they are attached. This allows you to extend the behavior of your EMF objects by implementing observers that provide the extended behavior and attaching those observers to your EMF objects.

3.0 Tasks

3.1 Defining Your Model

The first step in creating an EMF-based application is to define your model. EMF allows you to express your model in a variety of ways.

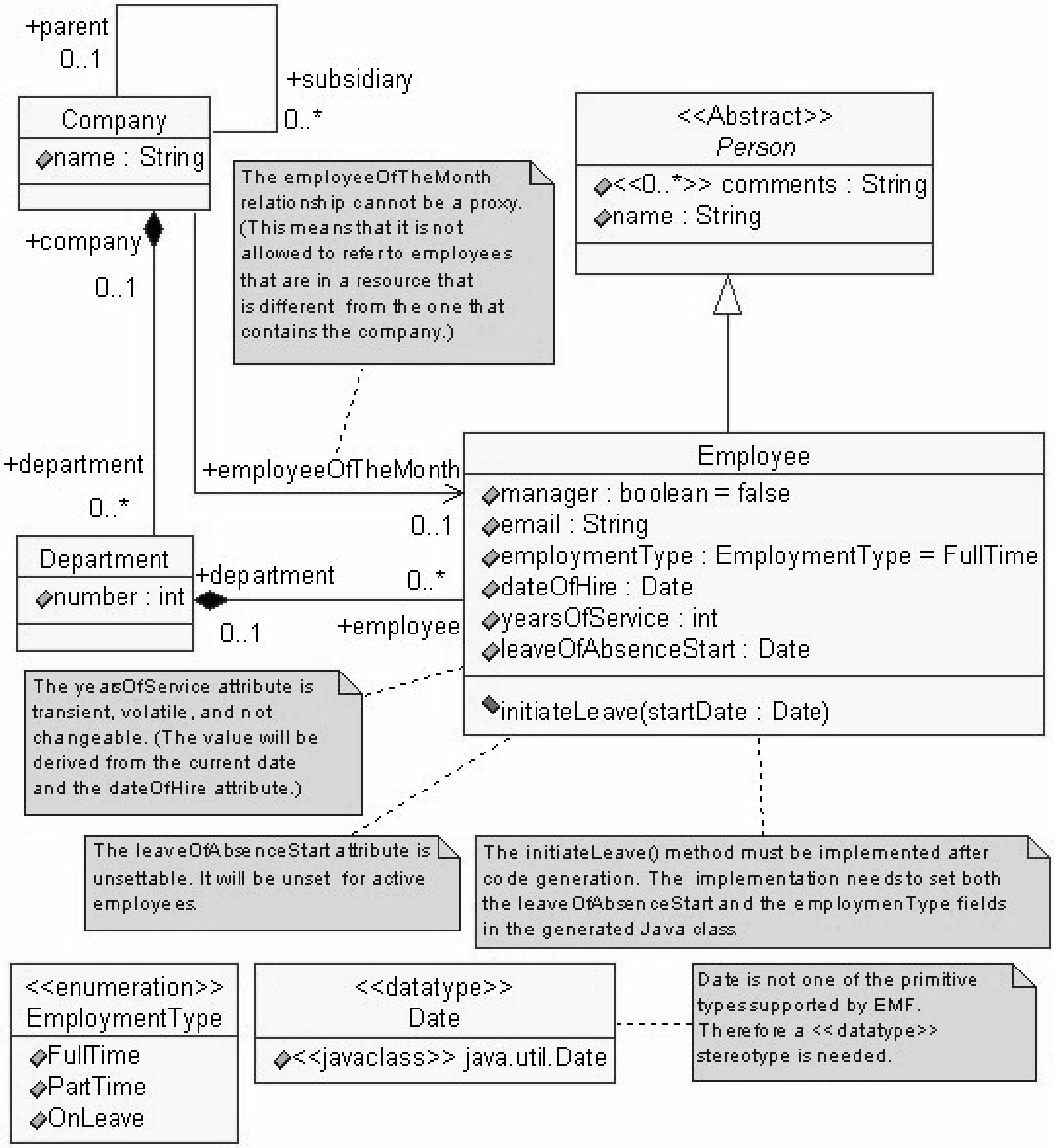
Whichever form you choose, your model specification will consist of some number of packages, classes, attributes, and associations. Each of these has various properties that you can specify.

Some properties are mandatory, but most have some default settings, so you only need to specify them explicitly if you wish to override the defaults. For example, when you specify an attribute, the attribute name is required but the multiplicity is not. (If you do not specify a multiplicity, single-valued is assumed.)

A complete list of the properties that are applicable to each model element is provided in the section "Ecore Properties and Codegen Specifications" on page 56.

For the examples that appear in the following sections, assume that we wish to create a model that consists of a single package called "enterprise" with classes that represent companies, departments, and employees. This details of this model are illustrate below using UML notation.

FIGURE 1. UML for enterprise model



The employeeOfTheMonth relationship cannot be a proxy. (This means that it is not allowed to refer to employees that are in a resource that is different from the one that contains the company.)

The yearsOfService attribute is transient, volatile, and not changeable. (The value will be derived from the current date and the dateOfHire attribute.)

The leaveOfAbsenceStart attribute is unsettable. It will be unset for active employees.

The initiateLeave() method must be implemented after code generation. The implementation needs to set both the leaveOfAbsenceStart and the employmentType fields in the generated Java class.

Date is not one of the primitive types supported by EMF. Therefore a <<datatype>> stereotype is needed.

You have three choices for how you could specify this model to EMF code generation:

- You can use the UML notation directly. (You will need to use some special annotations in the form of Rose properties). See “Code Generation Using Rational Rose” on page 9.
- You can write a file that expresses the classes of the model using XMI elements. See “Code Generation Using XMI documents” on page 21.
- You can write Java source files that define a Java interface for each of the classes in the model. (You will need to use some special annotations in the form of Java comments.) See “Code Generation Using Annotated Java Interfaces” on page 27.

Note that the comment boxes in Figure 1, “UML for enterprise model,” on page 8 indicate that there are implementation details for this model that cannot be expressed directly in UML. These comments have no impact on code generation. The actual mechanism that is used to specify this information to code generation will depend on which code generation technique you use. These mechanisms are discussed in the following sections.

3.1.1 Code Generation Using Rational Rose

If you use Rational Rose to define your model, you simply draw a Class Diagram containing Packages, Classes, Attributes, and Associations. An example of a UML diagram depicting a package is shown in Figure 1, “UML for enterprise model,” on page 8. In general, the UML elements in your diagram map directly to Ecore elements which determine the precise code generation patterns to be used. Additionally, there are a few special annotations that are used by the EMF basic code generation tool of which you may need to be aware.

These will be discussed in the following sections:

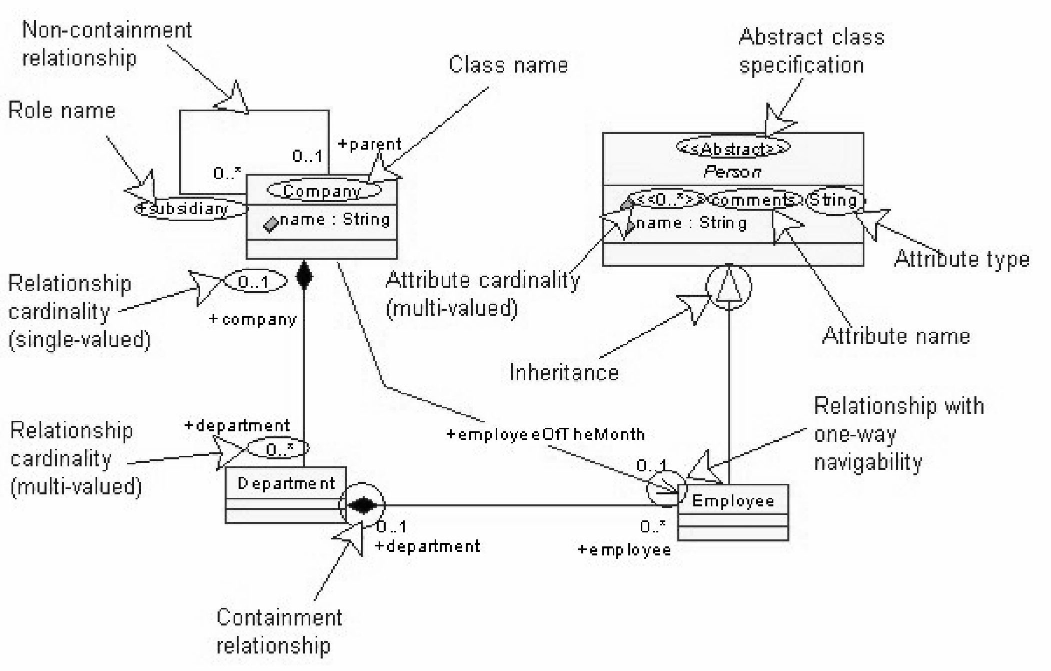
- Section 3.1.1.1, “Basic UML Model Elements,” on page 10 shows how you specify the most common properties of classes, attributes, and relationships.
- Section 3.1.1.2, “Specification of Abstract Classes,” on page 12 shows how an abstract class is specified.
- Section 3.1.1.3, “Attribute Specifications in UML,” on page 13 shows how you specify operations, datatypes and enumerations.
- Section 3.1.1.4, “The eCore Properties Page,” on page 15 shows how you set up your Rose model to include special ecore properties that are not part of standard UML
- Section 3.1.1.5, “Ecore Properties for Attributes,” on page 16 shows how you specify ecore properties that apply to attributes (i.e. transience, volatility, changeability, settability, and uniqueness.)
- Section 3.1.1.6, “Ecore Properties for Relationships,” on page 18 shows how you specify ecore properties that apply to relationships (i.e. transience, volatility, changeability, settability, and resolveability.)

- Section 3.1.1.7, “Ecore Properties for Packages,” on page 19 shows how you specify ecore properties that apply to attributes (i.e. prefix, package name, base package, namespace prefix and namespace URI.)
- Section 3.1.1.8, “Specifying Multiple Inheritance in UML,” on page 20 shows how you can specify that a class has multiple superclasses.

3.1.1.1 Basic UML Model Elements

The basic elements in your UML model are Classes, Attributes and Relationships. For example:

FIGURE 2. Basic Ecore elements in UML diagram



In most cases, the code generation utility will create a Java interface and a Java implementation class for each UML class. Each interface will have accessor methods to get and set each of the attributes and relationships specified in the model. For example, the *`Company`* class shown in Figure 2, “Basic Ecore elements in UML diagram,” on page 10 will generate a Java interface named `Company` and a Java class named `CompanyImpl`.

The accessor methods that are created for each attribute and association will vary depending on the properties of the corresponding UML elements.

- Single-valued attributes and navigable relationships will generate a `get()` method[1] that returns a value of the appropriate type and a `set()` method that accepts a parameter of that type.
- Multi-valued attributes and navigable relationships will generate only a `get()` method that returns an `EList`. The actual implementation of the `EList` that is returned is constrained to only accept values of the appropriate type.
- For relationships, the implementation of the `set()` method (in the case of single-valued relationships) or the `EList` (in the case of multi-valued relationships) will be different depending on whether the relationship uses containment. In particular, the implementation of a containment relationship will enforce the semantics that an instance of an object can only have a single container.
- Relationships that are not navigable will not result in the generation of any accessor methods.
- If you specify an inheritance relationship in your model, the resulting generated interface and implementation class will have the same inheritance structure.

For example, the *`Company`* class has a single-valued attribute called "*`name`*", two single-valued relationships called "*`employeeOfTheMonth`*" and "*`parent`*", and two multi-valued relationships called "*`department`*" and "*`subsidiary`*. Therefore the generated Company interface will include the following:

```
public interface Company extends EObject{
  String getName();
  void setName(String value);
  EList getDepartment();
  Employee getEmployeeOfTheMonth();
  void setEmployeeOfTheMonth(Employee value);
  Company getParent();
  void setParent(Company);
  EList getSubsidiary();
} // Company
```

Note that the "*`department`*" relationship on the *`Company`* class is a containment relationship while the "*`subsidiary`*" relationship is non-containment. Both of these relationships are multi-valued, which means they generated interface has a get method but no set method. The difference in the containment property leads to different implementations for the generated `get()` methods. For example, the generated `CompanyImpl` class will include the following methods:

---

[1] For a single-valued attribute of type boolean, an `is()` method is generated instead of a `get()` method.

```
public EList getDepartment() {
   if (department == null) {
      department =
         new EObjectContainmentWithInverseEList(
            Department.class,
            this,
            EnterprisePackage.COMPANY__DEPARTMENT,
            EnterprisePackage.DEPARTMENT__COMPANY);
   }
   return department;
}
public EList getSubsidiary() {
   if (subsidiary == null) {
      subsidiary =
         new EObjectWithInverseResolvingEList(
            Company.class,
            this,
            EnterprisePackage.COMPANY__SUBSIDIARY,
            EnterprisePackage.COMPANY__PARENT);
   }
   return subsidiary;
}
```

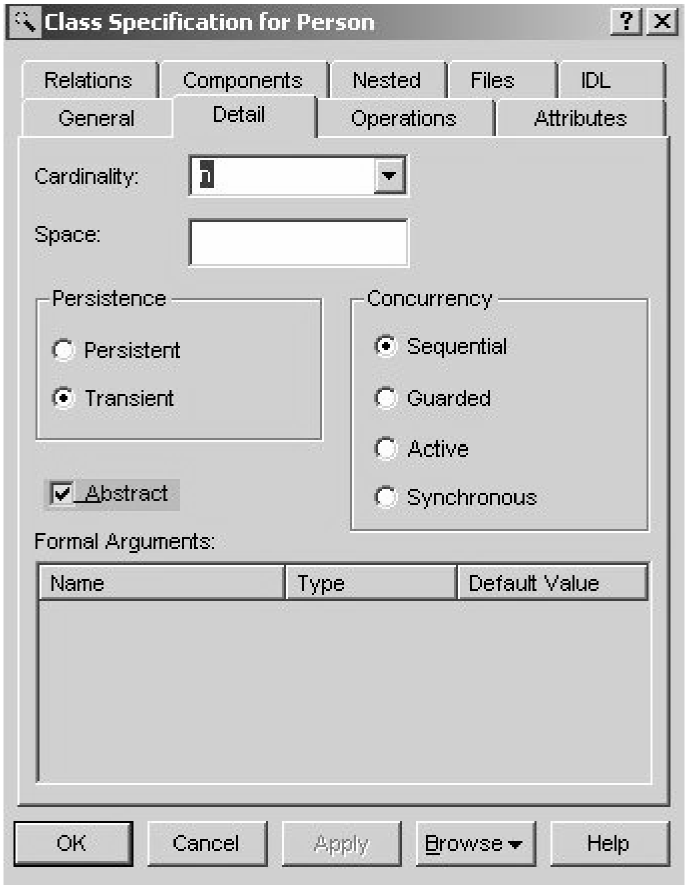
The *`Employee`* class inherits from the *`Person`* class. Therefore, the first line of the interface for `Employee` will start with:

```
public interface Employee extends Person
```

3.1.1.2 Specification of Abstract Classes

Note that the *`Person`* class is marked with the *`<<Abstract>>`* stereotype. This stereotype is only used in the UML for informational purposes to indicate that the Abstract property is set for this class. This property is set via the *`Detail`* page of the *`Specification`* dialog for the cass. For example:

FIGURE 3. Specification Dialog for the Person Class



Cardinality: n

Space:

Persistence

○ Persistent

● Transient

☑ Abstract

Concurrency

● Sequential

○ Guarded

○ Active

○ Synchronous

Formal Arguments:

- If you specify the *`Abstract`* property in your UML, the resulting generated implementation class will be abstract.

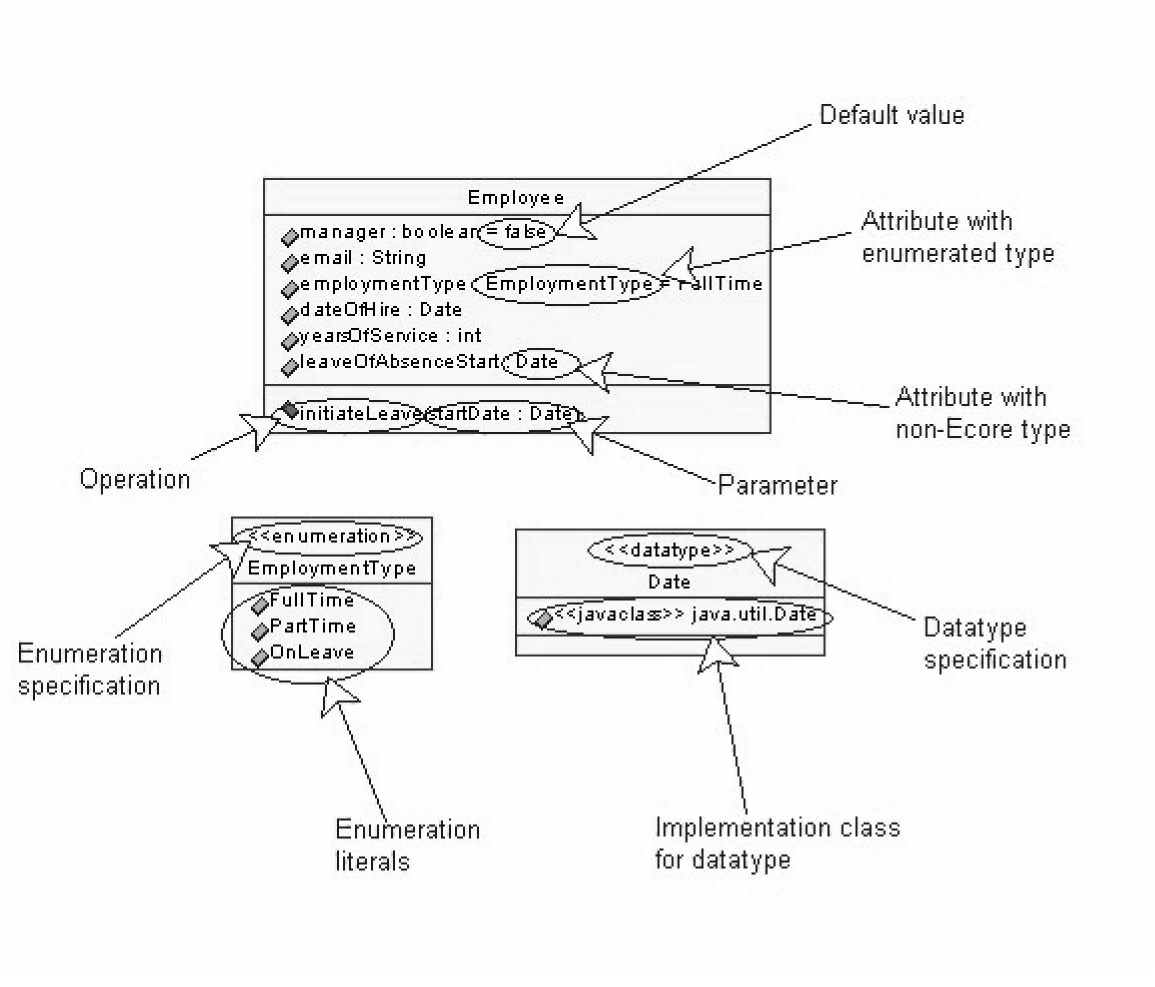
For example, the declaration for `PersonImpl` will be:

```
public abstract class PersonImpl extends EObjectImpl implements Person
```

3.1.1.3 Attribute Specifications in UML

The are some special conventions used by EMF to define the types of certain attributes. Consider the following segment of the *`enterprise`* package:

FIGURE 4. Enumerations, DataTypes, and Default Values


Default value
Employee
manager : boolean = false
email : String
employmentType : EmploymentType = FullTime
dateOfHire : Date
yearsOfService : int
leaveOfAbsenceStart : Date
initiateLeave(startDate : Date)
Attribute with enumerated type
Attribute with non-Ecore type
Operation
Parameter
<<enumeration>>
EmploymentType
FullTime
PartTime
OnLeave
<<datatype>>
Date
<<javaclass>> java.util.Date
Datatype specification
Enumeration specification
Enumeration literals
Implementation class for datatype

- If you specify an initial value in your UML, the resulting attribute will be initialized with the specified value
- If you specify an operation, the resulting interface will include the signature for that operation and the implementation class will have a stub method. (The generated implementation of the stub method will throw a `UnsupportedOperationException`, so you will need to modify this method by hand after code generation.)
- If you need to refer to a type that is not an EMF class in your model, you can declare that type using a *`<<datatype>>`* stereotype. UML classes with this stereotype do not generate any code. Note that this class must include a single attribute that defines the underlying type. The attribute should be flagged with a stereotype of *`<<javaclass>>`*.
- If you wish to define an enumeration, you can do so using a UML class with the *`<<enumeration>>`* stereotype. This results in the generation of a **`final class`** that has **`static final`** fields that represent the enumeration values and enumeration literal instances. The accessor methods for attributes of this enumeration type pass the literal instances that are defined in this class.

The Employee class has a “`dateOfHire`” attribute whose type is the datatype “`Date`” and an “`employmentType`” attribute whose type is the enumeration “`EmploymentType`”. Therefore, the `Employee` interface will include the following methods:

```
  Date getDateOfHire();
  void setDateOfHire(Date value);
  EmploymentType getEmploymentType();
  void setEmploymentType(EmploymentType value);
  void initiateLeave(Date startDate);
```

The `EmploymentType` interface is generated as follows:

```
public final class EmploymentType extends AbstractEnumerator
{
  public static final int FULL_TIME = 0;
  public static final int PART_TIME = 1;
  public static final int ON_LEAVE = 2;
  public static final EmploymentType FULL_TIME_LITERAL =
                                   new EmploymentType(FULL_TIME, "FullTime");
  public static final EmploymentType PART_TIME_LITERAL =
                                   new EmploymentType(PART_TIME, "PartTime");
  public static final EmploymentType ON_LEAVE_LITERAL =
                                   new EmploymentType(ON_LEAVE, "OnLeave");
  public static final List VALUES =
                   Collections.unmodifiableList(Arrays.asList(VALUES_ARRAY));
  public static EmploymentType get(String name)
  {...}
  public static EmploymentType get(int value)
  {...}
  private EmploymentType(int value, String name)
  {
    super(value, name);
  }

} //EmploymentType
```

3.1.1.4 The eCore Properties Page

In some cases there are properties that are required for code generation that cannot be expressed in standard UML. For these cases, EMF provides a special Rose properties file called ecore.pty. These properties are not shown explicitly in the example in Figure 1 on page 8, although annotations are used in the diagram to indicate that the properties have been defined.

Before you can use these properties, you must first add the appropriate model properties file to your Rose model. This file is the ecore.pty file that is shipped in the org.eclipse.emf.ecore plugin in the src\models directory.

The mechanism for adding these properties to your model is shown in Figure 5, “Adding a Properties File to Your Model,” on page 16 and Figure 6, “Selecting the eCore.pty file,” on page 16.

FIGURE 5. Adding a Properties File to Your Model

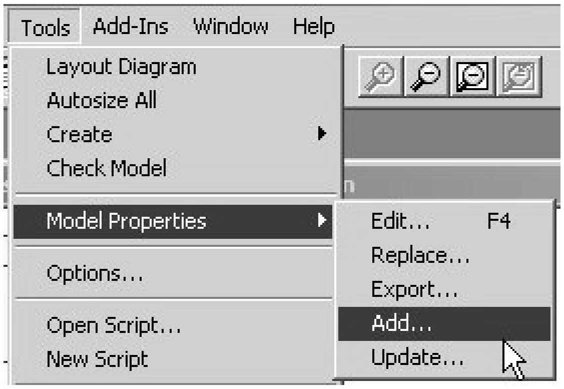
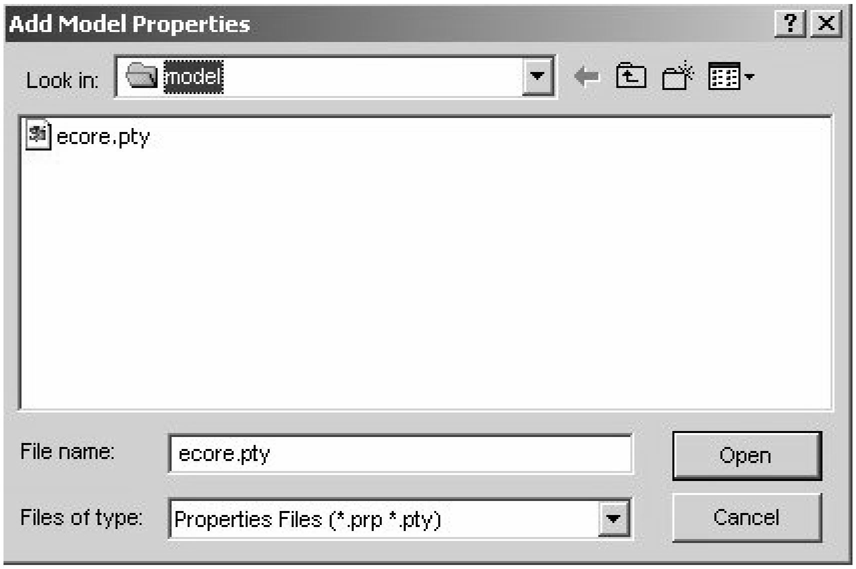


FIGURE 6. Selecting the eCore.pty file



Look in: model

ecore.pty

File name: ecore.pty

Files of type: Properties Files (*.prp *.pty)

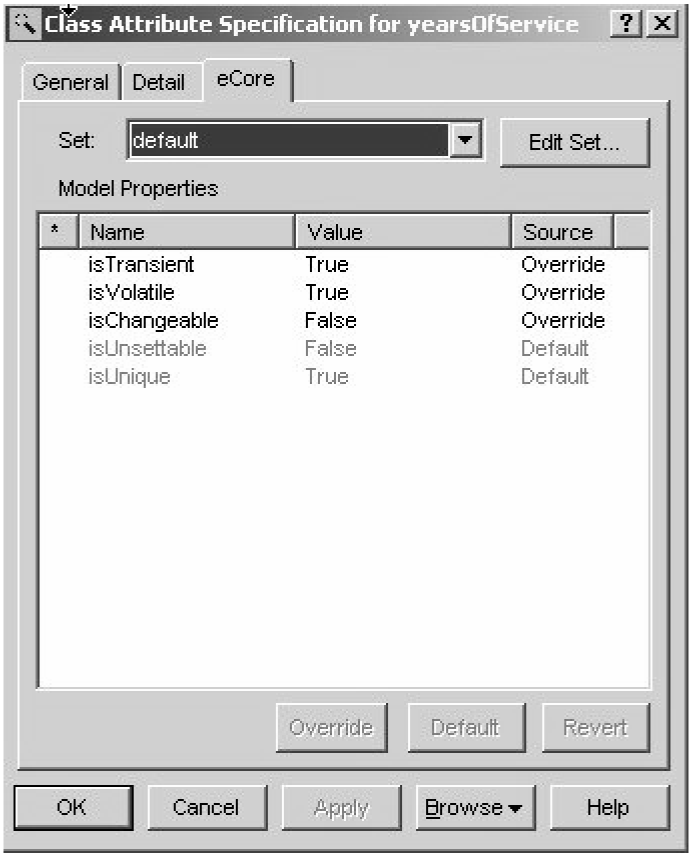
In order to see a particular property, you need to open up the specifications page for the object to which the property applies. The various properties are illustrated in

- “Ecore Properties for Attributes” on page 16
- “Ecore Properties for Relationships” on page 18
- “Ecore Properties for Packages” on page 19

3.1.1.5 Ecore Properties for Attributes

The following figure illustrates the eCore Properties page for Attributes.:

FIGURE 7. Properties page for the yearsOfService attribute



Set: default

*	Name	Value	Source
	isTransient	True	Override
	isVolatile	True	Override
	isChangeable	False	Override
	isUnsettable	False	Default
	isUnique	True	Default

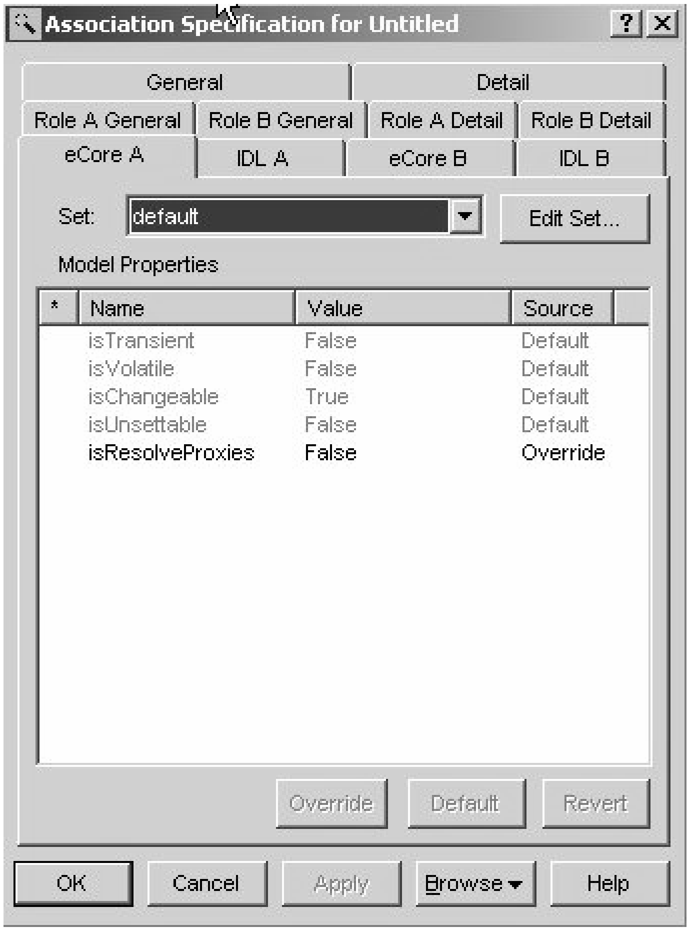
- If `isTransient` is `True`, the attribute or relationship will not be stored.
- If `isVolatile` is `True`, the attribute or relationship will not have any storage associated with it and the generated implementations of the `get()` and `set()` method for the attribute or relationship will throw an `UnsupportedOperationException`. (In other words, you will need to implement these methods explicitly.)
- If `isChangeable` is `True`, no `set()` method is generated for the attribute or association.
- If `isUnsettable` is `True`, `isSet()` and `unset()` methods are generated for the attribute or association. (Note that this requires additional runtime storage for each such attribute or association.)
- If `isUnique` is `True` for an attribute that has multi-valued multiplicity, uniqueness semantics are enforced for the list that holds the attribute values.

In the case of the `yearsOfService` attribute shown here, the intention is to implement the attribute to be derived from the `dateOfHire` attribute and from current date when the application is run. The settings on the ecore properties page instruct code generation to omit the default implementation of the get method from the implementation class (`isVolatile=True`), to omit the set method altogether from the interface (`isChangeable=False`), and to mark the attribute as transient so it will not be serialized (`isTransient=True`.)

3.1.1.6 Ecore Properties for Relationships

The following figure shows the specification dialog for the `employeeOfTheMonth` association. It is opened to the **`eCore A`** page, which displays that Ecore attributes that apply to the `employeeOfTheMonth` role of the association.

FIGURE 8. Properties page for the employeeOfTheMonth association



Set: default

*	Name	Value	Source
	isTransient	False	Default
	isVolatile	False	Default
	isChangeable	True	Default
	isUnsettable	False	Default
	isResolveProxies	False	Override

For the most part, the properties that apply to attributes also apply to relationships. (See Figure 7, “Properties page for the yearsOfService attribute,” on page 17 .) The exceptions are that the `isUnique` property only applies to attributes and that the `isResolveProxies` property only applies to relationships.

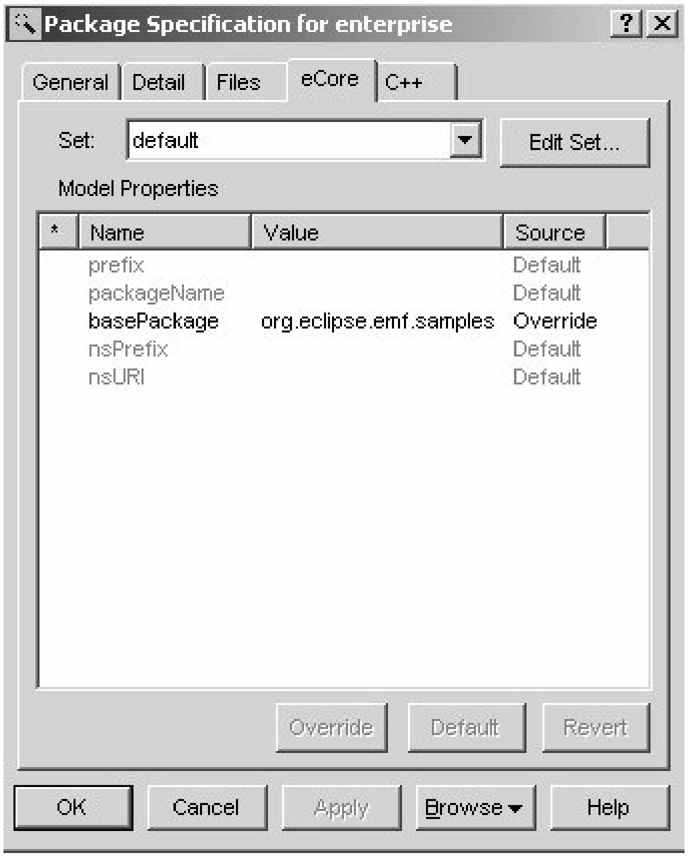
- If `isResolveProxies` is `False` for a relationship, the two ends of a relationship would typically not be stored in separate documents. If they are stored in separate documents, you will need to resolve the proxies manually.

In the case of the `employeeOfTheMonth` relationship shown here, the intention is that the indicated employee must be one of the employees that contained in the company. Therefore, it will not be necessary to allow for the possibility that the target of this relationship is in a different XMI document from the source. Setting the `isResolveProxies` property to `False` suppresses the generated code that would attempt to resolve proxies when accessing this relationship, which makes the accessor method more efficient than it would otherwise be.

3.1.1.7 Ecore Properties for Packages

The following figure illustrate the eCore page of the Specification dialog for Packages.

FIGURE 9. Ecore Properties Page for the Package



Set: default

Model Properties

*	Name	Value	Source
	prefix		Default
	packageName		Default
	basePackage	org.eclipse.emf.samples	Override
	nsPrefix		Default
	nsURI		Default

- If *`prefix`* is specified, the value is used as the prefix for constructing the name of the `Package` and `Factory` interfaces and classes. Otherwise, the capitalized package name is used to construct these names.
- If *`packageName`* is specified, the value is used for the rightmost part of the name of the generated Java package. Otherwise, the package name is used.
- If *`basePackage`* is specified, the value is used for the qualifier for the generated Java package.
- If *`nsPrefix`* is specified, the value is used as the namespace prefix for any XMI documents that contain instances of classes from this package. Otherwise, the package name is used.
- If *`nsURI`* is specified, the value is used as the namespace URI for any XMI documents that contain instances of or associations to classes from this package.

In this example, there is no *`prefix`*. There is also no *`packageName`*, so the default name will be the name of the package, which is "*`enterprise`*". The *`basePackage`* is "*`org.eclipse.emf.samples`*". Therefore the fully qualified name of the generated package interface will be

`org.eclipse.emf.samples.enterprise.EnterprisePackage.java`.

3.1.1.8 Specifying Multiple Inheritance in UML

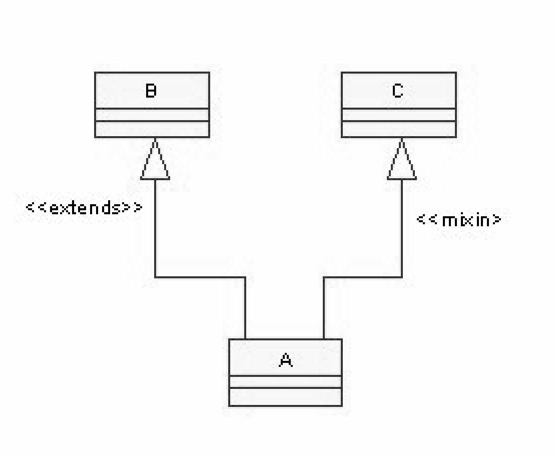
EMF supports multiple inheritance of interfaces in much the same way that multiple inheritance is supported in Java. You can define a class in UML that inherits from more than one superclass. The code that is generated for this class will include an interface that extends all the interfaces corresponding to the specified superclasses, but the implementation class will only extend one of the implementation classes.

This means that a generated class may not define all the methods that it is required to implement based on the interfaces that the class supports. You will need to make sure that an implementation is provided for any method that is missing.

You can specify which superclass is the primary superclass in your UML by attaching a stereotype to the generalization relationship that appears in your model. A stereotype of *`<<extends>>`* indicates the primary superclass and a stereotype of *`<<mixin>>`* indicates some other superclass. For a given subclass, there can only be one generalization that carries the *`<<extends>>`* stereotype.

For example:

FIGURE 10. UML for Multiple Inheritance



In this model, class *`A`* has two superclasses, *`B`* and *`C`*. The generated interface, `A,` will start with:

```
public interface A extends B, C
```

while the generated implementation class, `AImpl,` will begin with:

```
public class AImpl extends BImpl implements A
```

3.1.2 Code Generation Using XMI documents

You can also write XMI documents that can fully specify your model. There are two different types of XMI documents that are needed to specify a model:

- There will be one ecore document for each package in your model. This document contains the detailed definitions of all the packages and classes in your model. The elements of the ecore document are the classes and attributes of the Ecore model. The details for specifying the XMI document are discussed in “Ecore Properties and Codegen Specifications” on page 56. The extension for the ecore documents should be “.ecore”. \

  The ecore document contains classes that are defined in the Ecore model that is used by the org.eclipse.emf.ecore plugin.
- There will be one genmodel document for the entire model. This document is the input to the code generation utility. It has references to elements defined in the ecore documents and also includes some additional information needed for code generation that is not part of the model. (For example, information about how names should be constructed.) The extension for the genmodel documents is “.genmodel”

  The genmodel document contains classes that are defined in the GenModel model that is used by the org.eclipse.emf.codegen.ecore plugin.

The following sections illustrate the contents of the `genmodel` and `ecore` documents that are used to specify the *`enterprise`* model show in Figure 1, “UML for enterprise model,” on page 8. (Note that for this example, the model consists of only a single package, hence there is only one ecore document. If the model were to include multiple root packages, there would be one ecore document for each package.)

The enterprise.genmodel document is shown in Section 3.1.2.1, “Genmodel Document for the enterprise Model,” on page 22.

The main elements of the enterprise.ecore document are shown in the following sections. (Note that these elements are, in fact, all part of a single document. They are separated into the various sections below to help direct your attention to the salient features of each element.)

- Section 3.1.2.2, “The Enterprise Package Element in the Ecore Document,” on page 23
- Section 3.1.2.3, “The Company Class Element in the Ecore Document,” on page 24
- Section 3.1.2.4, “The Department Class Element in the Ecore Document,” on page 25
- Section 3.1.2.5, “The Person Class Element in the Ecore Document,” on page 25
- Section 3.1.2.6, “The Employee Class Element in the Ecore Document,” on page 25
- Section 3.1.2.7, “The EmploymentType Enumeration Element in the Ecore Document,” on page 26

- Section 3.1.2.8, “The Date Datatype Element in the Ecore Document,” on page 27

3.1.2.1 Genmodel Document for the enterprise Model

The `genmodel` document is the document that ties together all the packages, classes, and features in a model and provides any additional information that is not in the model but that is needed by the code generation utility to produce the appropriate source code.

```
<?xml version="1.0" encoding="ASCII"?>
<genmodel:GenModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:ecore="http://
```

```
www.eclipse.org/emf/2002/Ecore"
    xmlns:genmodel="http://www.eclipse.org/emf/2002/GenModel"
    modelDirectory="/org.eclipse.emf.samples/src"
    editDirectory="/org.eclipse.emf.samples.edit/src" editorDirectory="/org.eclipse.emf.samples.editor/src"
    modelPluginID="org.eclipse.emf.samples.enterprise">
  <foreignModel>C:\emf\eclipse\plugins\org.eclipse.emf.samples\src\model\enterprise.mdl</foreignModel>
  <genPackages prefix="Enterprise" basePackage="org.eclipse.emf.samples"
                               ecorePackage="enterprise.ecore#/">
    <genEnums ecoreEnum="enterprise.ecore#//EmploymentType">
      <genEnumLiterals ecoreEnumLiteral="enterprise.ecore#//EmploymentType/FullTime"/>
      <genEnumLiterals ecoreEnumLiteral="enterprise.ecore#//EmploymentType/PartTime"/>
      <genEnumLiterals ecoreEnumLiteral="enterprise.ecore#//EmploymentType/OnLeave"/>
    </genEnums>
    <genDataTypes ecoreDataType="enterprise.ecore#//Date"/>
    <genClasses ecoreClass="enterprise.ecore#//Department">
      <genFeatures ecoreFeature="ecore:EAttribute enterprise.ecore#//Department/number"/>
      <genFeatures ecoreFeature="ecore:EReference enterprise.ecore#//Department/company"/>
      <genFeatures ecoreFeature="ecore:EReference enterprise.ecore#//Department/employee"/>
    </genClasses>
    <genClasses ecoreClass="enterprise.ecore#//Company">
      <genFeatures ecoreFeature="ecore:EAttribute enterprise.ecore#//Company/name"/>
      <genFeatures ecoreFeature="ecore:EReference enterprise.ecore#//Company/department"/>
      <genFeatures ecoreFeature="ecore:EReference enterprise.ecore#//Company/parent"/>
      <genFeatures ecoreFeature="ecore:EReference enterprise.ecore#//Company/subsidiary"/>
      <genFeatures ecoreFeature="ecore:EReference enterprise.ecore#//Company/employeeOfTheMonth"/>
    </genClasses>
    <genClasses ecoreClass="enterprise.ecore#//Person">
      <genFeatures ecoreFeature="ecore:EAttribute enterprise.ecore#//Person/comments"/>
      <genFeatures ecoreFeature="ecore:EAttribute enterprise.ecore#//Person/name"/>
    </genClasses>
    <genClasses ecoreClass="enterprise.ecore#//Employee">
      <genFeatures ecoreFeature="ecore:EAttribute enterprise.ecore#//Employee/manager"/>
      <genFeatures ecoreFeature="ecore:EAttribute enterprise.ecore#//Employee/email"/>
      <genFeatures ecoreFeature="ecore:EAttribute enterprise.ecore#//Employee/employmentType"/>
      <genFeatures ecoreFeature="ecore:EAttribute enterprise.ecore#//Employee/dateOfHire"/>
      <genFeatures ecoreFeature="ecore:EAttribute enterprise.ecore#//Employee/yearsOfService"/>
      <genFeatures ecoreFeature="ecore:EAttribute enterprise.ecore#//Employee/leaveOfAbsenceStart"/>
      <genFeatures ecoreFeature="ecore:EReference enterprise.ecore#//Employee/department"/>
      <genOperations ecoreOperation="enterprise.ecore#//Employee/initiateLeave">
        <genParameters ecoreParameter="enterprise.ecore#//Employee/initiateLeave/startDate"/>
      </genOperations>
    </genClasses>
  </genPackages>
</genmodel:GenModel>
```

Note that most of the elements of this document consist entirely of references into the `enterprise.ecore` document, which is described in the following sections.

The elements in this document that may contain additional information are the `genmodel:GenModel` element and the `genPackages` element.

3.1.2.2 The Enterprise Package Element in the Ecore Document

A package is specified as an `ecore:EPackage` element in an `ecore` document.

```
<?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="enterprise"
    nsURI="http:///enterprise.xmi" nsPrefix="enterprise">
   <eClassifiers ... >
      ...
   </eClassifiers>
</ecore:EPackage>
```

In this case we are defining a package whose name is “`enterprise`”. The details for the `eClassifiers` that comprise this package are illustrated in the following sections.

3.1.2.3 The Company Class Element in the Ecore Document

A class is specified as an `ecore:EClass` element in an XMI document. For example, the XMI that defines the *`Company`* class from the *`Enterprise`* model is:

```
  <eClassifiers xsi:type="ecore:EClass" name="Company">
    <eReferences name="department" eType="#//Department" upperBound="-1"
             containment="true" eOpposite="#//Department/company"/>
    <eReferences name="employeeOfTheMonth" eType="#//Employee"
             resolveProxies="false"/>
    <eReferences name="parent" eType="#//Company"
             eOpposite="#//Company/subsidiary"/>
    <eReferences name="subsidiary" eType="#//Company" upperBound="-1"
             eOpposite="#//Company/parent"/>
    <eAttributes name="name"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
```

Note the following:

- When no `lowerBound` is specified, `"0"` is assumed. When no `upperBound` is specified `“1”` is assumed. In the above example, the `employeeOfTheMonth` and `parent` references and the `name` attribute use both defaults, and therefore are single-valued. This means that code generation will generate `get()` and `set()` methods for these features.
- An `upperBound` that is set to “`-1`” (as in the `department` and `subsidiary` references) indicates that there is no upper bound. This implies that the cardinality is multi-valued. Therefore no `set()` method will be generated and that the `get()` method will return an `EList`.
- The `eOpposite` attribute identifies the opposite end of a relationship that is navigable in both directions. (For example, the `department` reference specifies an `eOpposite` attribute while the `employeeOfTheMonth` reference does not.) The `eOpposite` attribute is needed so that the generated code will ensure that when one end of a relationship is modified, the other end will be updated accordingly.
- When the `resolveProxies` attribute is set to `"false"`, (see the `employeeOfTheMonth` reference), the generated `get()` method will assume that the target object is never a proxy, and therefore will not attempt to resolve the target. This improves the performance of the

`get()` method, but it should only be used if you are sure that the target oar a reference will never be stored in a different document from the source.

3.1.2.4 The Department Class Element in the Ecore Document

The XMI that defines the *`Department`* class from the *`Enterprise`* model is:

```
<eClassifiers xsi:type="ecore:EClass" name="Department">
  <eReferences name="company" eType="#//Company" transient="true"
             eOpposite="#//Company/department"/>
  <eReferences name="employee" eType="#//Employee" upperBound="-1"
             containment="true" eOpposite="#//Employee/department"/>
  <eAttributes name="number"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>
```

3.1.2.5 The Person Class Element in the Ecore Document

An abstract class is specified as an `ecore:EClass` element in an XMI document where the `abstract` attribute is set to “`true`”. For example, the XMI that defines the *`Person`* class from the *`Enterprise`* model is

```
<eClassifiers xsi:type="ecore:EClass" name="Person" abstract="true">
  <eAttributes name="comments"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"
      upperBound="-1"/>
  <eAttributes name="name"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
</eClassifiers>
```

Note the following:

- The `abstract="true"` attribute will cause code generation to add the abstract keyword to the generated implementation class.

3.1.2.6 The Employee Class Element in the Ecore Document

The XMI that defines the *`Employee`* class from the *`Enterprise`* model is:

```
<eClassifiers xsi:type="ecore:EClass" name="Employee" eSuperTypes="#//Person">
  <eOperations name="initiateLeave">
    <eParameters name="startDate" eType="#//Date"/>
  </eOperations>
  <eReferences name="department" eType="#//Department" transient="true" eOpposite="#//Department/
employee"/>
  <eAttributes name="manager" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBool-
ean"
```

```
      defaultValueLiteral="false"/>
    <eAttributes name="email" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eAttributes name="employmentType" eType="#//EmploymentType" defaultValueLiteral="FullTime"/>
    <eAttributes name="dateOfHire" eType="#//Date"/>
    <eAttributes name="yearsOfService" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//
EInt"
      changeable="false" volatile="true" transient="true"/>
    <eAttributes name="leaveOfAbsenceStart" eType="#//Date" unsettable="true"/>
  </eClassifiers>
```

Note the following:

- The eSuperTypes="#//Person" attribute means that the generated code for the `Employee` interface will extend the `Person` interface and that the `EmployeeImpl` class will extend the `PersonImpl` class.
- The changeable="false" attribute (see yearsOfService) will mean that no `set()` method will be generated for the attributes and references to which it applies.
- The volatile="true" attribute (see yearsOfService) will mean that no storage will be reserved for the attributes and references to which it applies and also that the generated `get()` and `set()` methods will throw an `UnsupportedOperationException`.
- The transient="true" attribute (see yearsOfService) will mean that the attributes and references to which it applies will not be serialized.
- The data type of each attribute and reference is specified through the eType attribute. For primitive types, the value of eType is a type that is defined in the ecore model (e.g. "ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"). In the case of the employmentType and dateOfHire attributes, the value of eType are types that are defined in the enterprise package (e.g."#//EmploymentType" and "#//Date".)
- The defaultValueLiteral attribute (see manager and employmentType) provide the initial value that the attribute will have if it has not been explicitly set.
- The unsettable="true" attribute (see leaveOfAbsenceStart) will mean that the generated interface will include `unset()` and `isSet()` methods for the attribute to which it applies. (The use of unsettable="true" incurs some runtime overhead due to the fact that the implementation for this will require an additional field to remember whether or not the attribute has been set.)
- The eOperations element (which defines the initiateLeave method) means that the indicated method will be will generated in the interface and implementation class. The generated EmployeeImpl class will include a stub implementation of this method. The stub implementation will throw an `UnsupportedOperationException.`
-

3.1.2.7 The EmploymentType Enumeration Element in the Ecore Document

An enumeration is specified as an ecore:EEnum element in an XMI document. For example, the XMI that defines the *`EmploymentType`* enumeration from the *`Enterprise`* model is:

```
<eClassifiers xsi:type="ecore:EEnum" name="EmploymentType">
  <eLiterals name="FullTime"/>
  <eLiterals name="PartTime" value="1"/>
  <eLiterals name="OnLeave" value="2"/>
</eClassifiers>
```

Note the following:

- The `eLiterals` elements identify the literals that comprise this enumeration.
- Each `value` attribute should be unique. (The literal named `FullTime` uses the default for `value` which is `"0"`.)

3.1.2.8 The Date Datatype Element in the Ecore Document

A datatype is specified as an `ecore:EDataType` element in an XMI document. For example, the XMI that defines the *`Date`* datatype from the *`Enterprise`* model is:

```
<eClassifiers xsi:type="ecore:EDataType" name="Date"
          instanceClassName="java.util.Date"/>
```

Note the following:

- The `instanceClassName` attribute identifies the java interface or class to which the datatype maps.

3.1.3 Code Generation Using Annotated Java Interfaces

If you prefer to use Java interfaces to specify your model, all you need to do is to write a Java **`interface`** declaration to represent each class in your model and a Java **`class`** declaration to define each enumeration in your model.

Within each **`interface`** you will need to specify a `get()` method for each attribute or relationship in the model and within each **`class`** you will need to specify a field to represent each enumeration literal.

Each of these **`interface`** statements, **`class`** statements, `get()` methods, and fields should be preceded by a javadoc comment that includes a `@model` tag. This tag is used to tell the code generation utility that the construct represents an element of your model.

The code generation utility will automatically expand your **`interface`** declarations to include any other methods that are needed to represent and access the classes in your model. All the necessary implementation classes will also be generated automatically.

Much of the information that is needed to generate code can be gleaned from the Java **`interface`** specification. For example, the name of the package that a class belongs to is derived from the **`package`** statement that appears in the corresponding **`interface`** declaration. Also, the names of all attributes are derived by stripping off the prefix “`get`” from the method names. For single-valued attributes and references, the type is the return type of the `get()` method. Multi-valued attributes and references are identified by methods that have a return type of `List` or `EList`.

Ecore properties that cannot be derived from the Java source code can be expressed via the `@model` tags. Each property is specified in the form:

```
/**
 * @model [<property>=<value>...]
 */
```

A full list of the possible properties can be found in Section 4.1, “Ecore Properties and Codegen Specifications,” on page 56.

Examples of the `@model` tags that are needed to specify the *`enterprise`* model illustrated in Figure 1, “UML for enterprise model,” on page 8 can be found in the following sections:

- “Java Specification for the Enterprise Package” on page 28
- “Java Specification for the Company Class” on page 29
- “Java Specification for the Department Class” on page 31
- “Java Specification for the Person Class” on page 31
- “Java Specification for the Employee Class” on page 32
- “Java Specification for the EmploymentType Enumeration” on page 33

3.1.3.1 Java Specification for the Enterprise Package

In certain cases, it may be useful to provide a interface to define a package. Note that usually, this declaration is not required at all. The classes and enumerations that belong to the package in your model are automatically identified based on the **`interface`** and **`class`** declarations that are in a java package. The datatypes that belong to your package are identified by attributes and methods that use types that are not classes in your model.

The only situation where it may be necessary to provide the declaration shown here is when you wish to override some the default settings for the package or when you wish to define a datatype that is not actually referenced in your model.

```
package org.eclipse.emf.samples.enterprise;
public interface EnterprisePackage extends EPackage{
  String eNAME = "enterprise";

  String eNS_URI = "enterprise.xmi";

  String eNS_PREFIX = "enterprise";

  /**
   * @model instanceClass="java.util.Date"
   */
  EDataType getDate();
} //EnterprisePackage
```

Note the following:

- The `eNAME`, `eNS_URI`, and `eNS_PREFIX` shown here are not actually required in this case because the indicated values are in fact the default values that would normally be generated based on the **`package`** statement.
- A get method in the package interface that has an `@model` tag and that has a return type of `org.eclipse.emf.ecore.EDataType` represents a datatype in your model. The `instanceClass` attribute on the `@model` tag identifies the java interface or class to which the datatype maps.
- Note that the classes and enumerations that are part of the package do not have to be specified explicitly. The EMF code generation utility will automatically determine the rest of the contents of the package based on the other interfaces and classes that are processed.

3.1.3.2 Java Specification for the Company Class

A class in your model is specified as a Java interface. The name of the class is the name of the interface. The attributes and references in the class are represented by `get()` methods in your interface that are preceded by a `@model` tag.

For example, the Java interface that defines the *`Company`* class from the *`Enterprise`* model is:

```
package org.eclipse.emf.samples.enterprise;
import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.ecore.EObject;
/**
 * @model
 */
public interface Company extends EObject{
  /**
   * @model
   */
  String getName();
  /**
   * @model type="Department" opposite="company" containment="true"
   */
  EList getDepartment();
  /**
   * @model resolveProxies="false"
   */
  Employee getEmployeeOfTheMonth();
  /**
   * @model opposite="subsidiary"
   */
  Company getParent();
  /**
   * @model type="Company" opposite="parent"
   */
  EList getSubsidiary();
} // Company
```

Note the following:

- When the `get()` method returns a single object, the cardinality of the attribute or reference is single-valued. (For example, see the `getName()`, `getEmployeeOfTheMonth()`, and `getParent()` methods.) This means that code generation will generate both `get()` and `set()` methods for these features.
- When the `get()` method returns a `EList`, the cardinality of the attribute or reference is multi-valued. (For example, see the `getDepartment()` and `getSubsidiary()` methods). This means that no `set()` method will be generated. Note that the `type` attribute on the `@model` tag is required in this case to indicate the type of object that is contained in the `EList`.
- The `opposite` attribute on the `@model` tag identifies the opposite end of a relationship that is navigable in both directions. (For example, the `getDepartment()` method specifies an `opposite` attribute while the `getEmployeeOfTheMonth()` method does not.) The `opposite` attribute is needed so that the generated code for the implementation of the method will ensure that when one end of a relationship is modified, the other end will be updated accordingly.
- When the `resolveProxies` attribute on the `@model` tag is set to `"false"`, (see the `getEmployeeOfTheMonth()` method), the generated implementation of the `get()` method will assume that the target object is never a proxy, and therefore will not attempt to resolve the target. This improves the performance of the `get()` method, but it should only be used if you are sure that the target of the reference will never be stored in a different document from the source.

3.1.3.3 Java Specification for the Department Class

The Java interface that defines the `Department` class from the `Enterprise` model is:

```
package org.eclipse.emf.samples.enterprise;
import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.ecore.EObject;
/**
 * @model
 */
public interface Department extends EObject{
  /**
   * @model
   */
  int getNumber();
  /**
   * @model opposite="department"
   */
  Company getCompany();
  /**
   * @model type="Employee" opposite="department" containment="true"
   */
  EList getEmployee();
} // Department
```

3.1.3.4 Java Specification for the Person Class

The Java interface that defines the `Person` class from the `Enterprise` model is:

```
package org.eclipse.emf.samples.enterprise;
import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.ecore.EObject;
/**
 * @model abstract="true"
 */
public interface Person extends EObject{
  /**
   * @model type="String"
   */
  EList getComments();
  /**
   * @model
   */
  String getName();
} // Person
```

Note the following:

- The `abstract="true"` attribute on the `@model` tag will cause code generation to add the **`abstract`** keyword to the generated implemenation class.

3.1.3.5 Java Specification for the Employee Class

The Java interface that defines the `Employee` class from the `Enterprise` model is:

```
package org.eclipse.emf.samples.enterprise;
/**
 * @model
 */
public interface Employee extends Person{
  /**
   * @model default="false"
   */
  boolean isManager();
  /**
   * @model
   */
  String getEmail();
  /**
   * @model default="FullTime"
   */
  EmploymentType getEmploymentType();
  /**
   * @model dataType="enterprise.Date"
   */
  Date getDateOfHire();
  /**
   * @model transient="true" changable="false" volatile="true"
   */
  int getYearsOfService();
  /**
   * @model unsettable="true" dataType="enterprise.Date"
   */
  Date getLeaveOfAbsenceStart();
  /**
   * @model opposite="employee"
   */
  Department getDepartment();
  /**
   * @model parameters="org.eclipse.emf.samples.enterprise.Date"
   */
  void initiateLeave(Date startDate);
} // Employee
```

Note the following:

- The `extends Person` specification on this interface will mean that the `EmployeeImpl` class will extend the `PersonImpl` class.
- The `changable="false"` attribute on the `@model` tag (see `getYearsOfService()`) will mean that no `set()` method will be generated for the attributes and references to which it applies.
- The `volatile="true"` attribute on the `@model` tag (see `getYearsOfService()`) will mean that no storage will be reserved for the attributes and references to which it applies and

also that the generated implementations for the `get()` and `set()` methods will throw an `UnsupportedOperationException`.

- The `transient="true"` attribute on the `@model` tag (see `getYearsOfService()`) will mean that the attributes and references to which it applies will not be serialized.
- The `default` attribute on the `@model` tag (see `isManager()` and `getEmploymentType()`) provide the initial value that the attribute will have if it has not been explicitly set.
- The `unsettable="true"` attribute on the `@model` tag (see `getLeaveOfAbsenceStart()`) will mean that the generated interface will include `unset()` and `isSet()` methods for the attribute to which it applies. (The use of `unsettable="true"` incurs some runtime overhead due to the fact that the implementation for this will require an additional field to remember whether or not the attribute has been set.)
- The `initiateLeave(Date startDate)` will be treated as an operation. The generated EmployeeImpl class will include a stub implementation of this method.

3.1.3.6 Java Specification for the EmploymentType Enumeration

An enumeration in your model is specified as a **`public final class`** that extends `org.eclipse.emf.common.util.AbstractEnumerator`. You need to identify the names and values of the enumeration literals and EMF code generation will automatically fill in the implementation details. For example, the Java class that defines the *`EmploymentType`* enumeration from the *`Enterprise`* model is:

```
package org.eclipse.emf.samples.enterprise;
import org.eclipse.emf.common.util.AbstractEnumerator;
/**
 * @model
 */
public final class EmploymentType extends AbstractEnumerator
{
  /**
   * @model name="FullTime"
   */
  public static final int FULL_TIME = 0;

  /**
   * @model name="PartTime"
   */
  public static final int PART_TIME = 1;

  /**
   * @model name="OnLeave"
   */
  public static final int ON_LEAVE = 2;

  private EmploymentType(int value, String name)
  {
    super(value, name);
  }


} //EmploymentType
```

Note the following:

- Each `public static final int` field defines an enumeration literal. The initial values for each field should be unique.
- The name of each literal is given by the `name` attribute on the `@model` tag.

3.2 Generating your model

The steps for invoking the EMF code generation utility are described in the document called "*Tutorial: Generating an EMF Model*", which can be found in the "Documents" section of the EMF web site. Please see http://www.eclipse.org/emf/ for details.

3.3 Configuring your EMF Runtime Environment

There is some setup that may be needed before you can start working with EMF objects.

In some cases there are three alternative mechanisms you can use to do the necessary setup. The choice of which mechanism to use will depend on whether or not your application runs from within the Eclipse workbench and whether the applicable configuration option applies globally or locally.

If your application runs as a plugin within the workbench, you can use your plugin.xml file to specify many of the configuration options you need. Otherwise, you will need to invoke APIs that initialize and register the prerequisite objects. The specific API that you need to use will depend on whether the customization is meant to apply globally or locally.

The following sections list the setup actions that you may need to take. Where appropriate, each section describes the alternative setup mechanisms for specifying each customization.

- If you are using a generated package, you will need to make sure the package is either initialized or registered before you begin. See “Registering/Initializing a Package” on page 35.
- At runtime, the contents of a generated package are accessed through a singleton instance of a generated Package class and instances of classes in the package are created using a singleton instance of a generated Factory class. Your application may need to establish a reference to one or both of these singleton objects. See “Accessing the Package and Factory classes” on page 37.
- A `Resource` corresponds to a collection of objects that are serialized in a single persistent stream. If you are creating new objects or loading objects from an existing stream, you will need to know how to create a `Resource`. See “Creating a `Resource`” on page 37.
- A `ResourceSet` is a collection of `Resource` objects. You will need a `ResourceSet` if you have `Resource`s that have cross references or have common customizations. See “Creating a `ResourceSet`” on page 38.
- A `Resource.Factory` is used by the EMF runtime to create a new `Resource` whenever one is needed. If you need to provide your own implementation of the `Resource` interface (e.g., if you want to serialize in a format other than XMI) then you will also have to implement and register a Resource.Factory to instantiate your `Resource` class. See “Registering a `Resource.Factory`” on page 39.
- A `URIConverter` is a class that determines how a relative `URI` is resolved to an absolute `URI`. If you need to override the default processing, you will need to implement and register your own implementation of the `URIConverter` interface. See “Registering a URIConverter” on page 41.
- `Adapter` objects handle events that are triggered by a `Notifier`. One possible mechanism for establishing the association between Adapter objects and Notifier objects is to attach an AdapterFactory to a ResourceSet. See “Registering an AdapterFactory” on page 41.

3.3.1 Registering/Initializing a Package

For generated packages, before you can access the classes of a package, you need to ensure that the package has been registered and initialized. If you are running within the EMF workbench, you can register packages through the `plugin.xml` file. (See “Registering and Initializing a Generated Package in a Plugin” on page 36.) Otherwise, you need to explicitly invoke a method that will initialize the package. (See “Registering and Initializing a Generated Package Using APIs” on page 36.)

For dynamic packages, i.e., packages that are created by your application at runtime, your application is responsible for the initialization and registration of the package. (See “Registering a Dynamic Package” on page 36.)

3.3.1.1 Registering and Initializing a Generated Package in a Plugin

To preregister a package or packages, you would include the `org.eclipse.emf.ecore.generated_package` extension point in your `plugin.xml` file. For example, assume that you generated the Enterprise package into a Java package called “org.eclipse.emf.samples.enterpise”. Your plugin may contain the following extension point element:

```
<extension point="org.eclipse.emf.ecore.generated_package">
    <package uri = "enterpise.xmi"
        class = "org.eclipse.emf.samples.enterpise.EnterpisePackage"/>
</extension>
```

3.3.1.2 Registering and Initializing a Generated Package Using APIs

If you are running outside of the workbench, you will need to explicitly initialize each package that you require. You do this by invoking the static `init()` method that is defined on each package implementation. For example, a method to initialize the EnterpisePackage would look like this:

```
protected void initializeEnterprisePackage() {
       EnterpisePackageImpl.init();
}
```

3.3.1.3 Registering a Dynamic Package

If you have a dynamic package (i.e., a package that is created by your application at runtime rather than being generated) you will need to ensure that your package is correctly registered.

After you create your package you must ensure that it is registered. One way to do this is using the following method:

```
public static void registerDynamicPackage(String uri, EPackage pkg)
{
  EPackage.Registry.INSTANCE.put(uri, pkg);
}
```

Where:

- `uri` is the string under which the package is registered
- `pkg` is the package itself

A dynamic package must be an instance of a class that implements the `EPackage` interface. Note the default implementation of `EPackage`, `EPackageImpl`, has constructors that take a `packageURI` string as one of their arguments register the new package automatically. Therefore if you use one of these constructors to create your new package, you do not have to do anything else to ensure that the package is registered. However, if you use one of the other `EPackageImpl` constructors you need to register the package as shown above.

3.3.2 Accessing the Package and Factory classes

If you need to access the meta data for your package, you will need to acquire a reference to the generated `Package` class. Similarly, if you need to create instances of the classes in your model, you will need to do so using the generated `Factory` class. If you access these objects frequently, you may find it convenient to cache the references to them. For example, a method to look up and cache the `EnterpisePackage` and `EnterpiseFactory` might look like this:

```
EnterpisePackage enterpisePackage=null;
EnterpiseFactory enterpiseFactory=null;
protected void lookupPackageAndFactory() {
       Map registry = EPackage.Registry.INSTANCE;
       String enterpiseURI = EnterpisePackage.eNS_URI;
       enterpisePackage = (EnterpisePackage)registry.get(enterpiseURI);
       enterpiseFactory = enterpisePackage.getEnterpiseFactory();
}
```

3.3.3 Creating a `Resource`

A `Resource` should be created through either a `Resource.Factory` or a `ResourceSet`. (Actually, the `createResource()` method on the `ResourceSet` class is implemented using `Resource.Factory`, so ultimately, every `Resource` object is created through a `Resouce.Factory`.)

`Resource` objects may also be created automatically. If you reference an object that is defined in a `Resource` that has not yet been loaded, the `Resource` will be automatically loaded.

For example, to create a `Resource` from a `ResourceSet`, you could use the following method.

```
public static Resource createResourceFromResourceSet(ResourceSet resSet,
                                                     String uri)
{
   Resource r = null;
   r = resSet.createResource(URI.createURI(uri));

   return r;
}
```

Where:

- `resSet` is the `ResourceSet` that will contain the new `Resource`
- `uri` is the URI for the `Resource` to be serialized

To create a `Resource` from a `Resource.Factory`, you could use the following method. Note that if you do this, you will eventually need to add the `Resource` to a `ResourceSet` explicitly.

```
public static Resource createResourceFromDefaultFactory(String uri)
{
   Resource r = null;
   Resource.Factory resFactory =
      Resource.Factory.Registry.INSTANCE.getFactory(
         URI.createURI(uri));
   r = resFactory.createResource(URI.createURI(uri));

   return r;
}
```

Where:

- `uri` is the URI for in which the Resource to be serialized

3.3.4 Creating a `ResourceSet`

You can create a `ResourceSet` simply by invoking **`new`** on an implementation of the `ResourceSet` interface. The default implementation is in `org.eclipse.emf.ecore.resource.impl`.

The `createResourceSet()` method defined below provides the convenience of being able to initialize the `ResourceSet` with a specified `Resource.Factory.Registry` or `URIConverter`. If these are needed but do not exist at the time the `ResourceSet` is created, they can be added later.

```
public static ResourceSet createResourceSet(Resource.Factory.Registry r,
                                            URIConverter c) {
   ResourceSet resSet = new ResourceSetImpl();
   if (c!=null) resSet.setURIConverter(c);
   if (r!=null) resSet.setResourceFactoryRegistry(r);
   return resSet;
```

Where:

- `r` is the `Resource.Factory.Registry`, if any, that will be used by the new `ResourceSet`.
- `c` is the `URIConverter`, if any, that will be used by the new `ResourceSet`.

3.3.5 Registering a `Resource.Factory`

You have the option of substituting your own implementation of the `Resource` interface for the default implementation provided by EMF. This enables you to control the format used to serialize your data.

In order to specify which implementation of `Resource` to use, you need to register a `Resource.Factory` that can create an instance of the desired `Resource`. The default `Resource.Factory` used to create XMI streams is `org.eclipse.emf.ecore.xmi.XMIResourceFactoryImpl`.

You can register a `Resource.Factory` by either protocol or file extension. Once a `Resource.Factory` is registered, anytime a `Resource` is generated, if the URI matches one of the registered protocols or extensions, the specified `Resource.Factory` will be used. As a special case, “*” can be used as a wild card to register an extension `Resource.Factory` as applying to all extensions. (Protocols take precedence, so if a URI matches both a registered protocol and a registered extension, the protocol will be used. Specific extensions take precedence over the wild card.)

The `Resource.Factory` that you register can be any type that implements the `org.eclipse.emf.ecore.resource.Resource.Factory` interface. So, for example, if you want to save your documents in a format other than XMI, you would implement a `Resource` that loads and saves the format you choose and then you would implement and register a `Resource.Factory` that creates an instance of your `Resource` implementation.

3.3.5.1 Registering a `Resource.Factory` for a Plugin

You can use the plugin extension points “org.eclipse.emf.ecore.extension_parser” and “org.eclipse.emf.ecore.protocol_parser” to register an implementation of a `Resource.Factory`. (The term “parser” is used here because the specified `Resource.Factory` determines which type of `Resource` is used which in turn determines how an `InputStream` will be parsed.)

For example, if you have defined an implementation of `Resource.Factory` called `org.eclipse.dtd.impl.DTDResourceFactoryImpl` which creates a Resource that can be used to parse and serialize DTD files, and you want this to apply to any file that has an extension of “.dtd”, you could do the following:

```
<extension point = "org.eclipse.emf.ecore.extension_parser">
    <parser type="dtd"
          class="org.eclipse.dtd.impl.DTDResourceFactoryImpl"/>
</extension>
```

On the other hand, if you want your `DTDResource` implementation to be used for any URI that has a protocol of “abc”, you could do the following:

```
<extension point = "org.eclipse.emf.ecore.protocol_parser">
    <parser protocolName="abc"
          class="org.eclipse.dtd.impl.DTDResourceFactoryImpl"/>
</extension>
```

3.3.5.2 Registering a `Resource.Factory` Globally

The following method registers a `ResourceFactory` in the global `Resource.Factory.Registry` under a specified key.

```
public static void registerGlobalResourceFactory(
    Resource.Factory f,
    String key,
    boolean isExtension)
{
    Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
    Map m;
    if (isExtension) m=reg.getExtensionToFactoryMap();
    else m=reg.getProtocolToFactoryMap();
    m.put(key, f);
}
```

Where:

- `f` is the Resource.Factory to be registered.
- `key` is the String under which the factory is registered.
- The `isExtension` flag indicates if the key represents an extension (**`true`**) or protocol (**`false`**)

3.3.5.3 Registering a `Resource.Factory` locally

The following method registers a `ResourceFactory` in the local `Resource.Factory.Registry` for a given `ResourceSet` under a specified key.

```
   public static void registerLocalResourceFactory(
      ResourceSet resSet,
      Resource.Factory f,
      String key,
      boolean isExtension)
   {
      Resource.Factory.Registry reg = resSet.getResourceFactoryRegistry();
      if (reg==null) {
         reg = new ResourceFactoryRegistryImpl();
         resSet.setResourceFactoryRegistry(reg);
      }
      Map m;
      if (isExtension) m=reg.getExtensionToFactoryMap();
      else m=reg.getProtocolToFactoryMap();
      m.put(key, f);

}
```

Where:

- `resSet` is the `ResourceSet` that defines the context for which the registration is in effect.
- `f` is the `Resource.Factory` to be registered.
- `key` is the `String` under which the factory is registered.
- The `isExtension` flag indicates if the key represents an extension (**`true`**) or protocol (**`false`**)

3.3.6 Registering a URIConverter

If you need a customized `URIConverter` you will need to define the implementation, create an instance of the implementation, and then attach it to a `ResourceSet`. There is no mechanism for registering a `URIConverter` for a plugin.

```
   public static void setURIConverter(ResourceSet resSet, URIConverter c) {
      if (c!=null) resSet.setURIConverter(c);
      return;
   }
```

Where:

- `resSet` is the `ResourceSet`.
- `c` is the `URIConverter`, that will be used by the `ResourceSet`.

3.3.7 Registering an AdapterFactory

An `AdapterFactory` is used to create `Adapter` objects and associate them with `Notifier` objects. You need to register one or more `AdapterFactory` objects with a `ResourceSet`.

```
public static void setURIConverter(ResourceSet resSet, AdapterFactory af) {
   if (af!=null) resSet.getAdapterFactories().add(af);
   return;
}
```

Where:

- `resSet` is the `ResourceSet`.
- `af` is the `AdapterFactory`, that will be added to the `ResourceSet`.

3.4 Running your application

3.4.1 Creating Instance Data

The following method illustrate the construction of two resources that contain instances of classes that are defined in the enterprise model.

```
/**
 * createInstances
 *
 * Creates two resources that contain instances of classes from the
 * enterprise package and adds the resources to the specified resource set.
 *
 */
static void createInstances(ResourceSet resSet) {
   // Access the factory (needed to create instances)
   Map registry = EPackage.Registry.INSTANCE;
   String enterpriseURI = EnterprisePackage.eNS_URI;
   EnterprisePackage enterprisePackage =
      (EnterprisePackage) registry.get(enterpriseURI);
   EnterpriseFactory enterpriseFactory =
      enterprisePackage.getEnterpriseFactory();

   // Create the resources
   Resource res1 =
      resSet.createResource(URI.createURI("megacorp.enterprise"));
   Resource res2 =
       resSet.createResource(URI.createURI("acme.enterprise"));

   // Create the first company and add it to a resource
   Company c1 = enterpriseFactory.createCompany();
   c1.setName("Mega Corp");
   Department d1 = enterpriseFactory.createDepartment();
   d1.setNumber(99);
   Employee e1 = enterpriseFactory.createEmployee();
   e1.setName("Jane Doe");

   c1.getDepartment().add(d1);
   d1.getEmployee().add(e1);
   res1.getContents().add(c1);

   // Create the second company and add it to a resource
   Company c2 = enterpriseFactory.createCompany();
   c2.setName("ACME");
   Department d2 = enterpriseFactory.createDepartment();
   d2.setNumber(101);
   Employee e2 = enterpriseFactory.createEmployee();
   e2.setName("John Smith");

   c2.getDepartment().add(d2);
   d2.getEmployee().add(e2);
   res2.getContents().add(c2);
   c1.getSubsidiary().add(c2);
}
```

This code performs the following tasks:

- Accesses the factory for the enterprise package. This involves first going to the EPackage.Registry to find the package that is registered under the URI that is assigned to the enterprise package and then using the package to access the factory.
- Creates two resources.
- Creates the objects that are in the resources and links them together.

Note that the only objects that are added directly to the resources are the instances of the Company class. The other classes are connected to the Company class through containment relationships, and therefore they should not be added to the resources.

The XMI documents that contain the contents of the resources are displayed in the following section ("Serializing Your Instance Data" on page 44).

3.4.2 Serializing Your Instance Data

The following method illustrate how the resources that were constructed in Section 3.4.1, "Creating Instance Data," on page 42 can be serialized into XMI files.

```
   /**
    * Creates and initializes the resrouce set and then saves the
    * resources contained in that resource set.
 */
   public static void createAndSave() {
      // Initialize the enterprise package
      EnterprisePackageImpl.init();

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

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

      // Create resources and instances; add the resources to the resource set
      createInstances(resSet);

      // Save each resource
      Iterator r = resSet.getResources().iterator();
      while (r.hasNext()) {
         Resource res = (Resource) r.next();
         Map options = new HashMap();
         options.put(XMIResource.OPTION_DECLARE_XML, Boolean.TRUE);
         try {
            res.save(options);
         } catch (IOException e) {
            System.out.println(e);
         }
      }
   }
```

This code performs the following tasks:

- Initializes the enterprise package, which causes the package to be registered so that it can later be looked up by its URI.
- Registers the XMI resource factory for the .enterprise extension.This will cause all documents with this extension to be treated as XMI documents.
- Obtains a resource set.

- Creates resources and instances using the method define in Section 3.4.1, “Creating Instance Data,” on page 42. This method will add all the resources it creates into the specified resource set.
- Saves the resources.

The result of the createIstances method is to produce two resources called megacorp.enterprise and acme.enterprise, which each contain one instance of a Company, a Department, and an Employee.

The contents of the megacorp.enterprise resource is:

```
<?xml version="1.0" encoding="ASCII"?>
<enterprise:Company xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:enterprise="enter-
prise.xmi" name="Mega Corp">
  <department number="99">
    <employee name="Jane Doe"/>
  </department>
  <subsidiary href="acme.enterprise#/"/>
</enterprise:Company>
<?xml version="1.0" encoding="ASCII"?>
```

And the contents of the acme.enterprise resource is:

```
<enterprise:Company xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:enterprise="enter-
prise.xmi" name="ACME">
  <department number="101">
    <employee name="John Smith"/>
  </department>
  <parent href="megacorp.enterprise#/"/>
</enterprise:Company>
```

3.4.3 Loading Instance Data

The following method illustrates how the XMI files that were generated in the example in Section 3.4.2, “Serializing Your Instance Data,” on page 44 can be loaded back into memory.

```
/**
 * load
 *
 * loads and prints the contents of a resource set
 *
 */
public static void load() {
   // Initialize the enterprise package
   EnterprisePackageImpl.init();

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

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

   // Load one of the resources into the resoruce set.
   Resource res = resSet.getResource(
      URI.createURI("megacorp.enterprise"),true);

   // Print all the resources inthe resource set.
   // Note: the process of printing the contents of the first resource
   // will cause the second resource to be demand loaded.
   List resList = resSet.getResources();
   for (int i=0; i<resList.size(); i++) {
      res = (Resource) resList.get(i);
      System.out.println("\n-----------------------------------------");
      System.out.println("\nContents of resource "+res.getURI());
      System.out.println("\n-----------------------------------------\n");
      UGRefPrint.print(res.getContents());
   }

}
```

This code performs the following tasks:

- Initializes the enterprise package, which causes the package to be registered so that it can later be looked up by its URI.
- Registers the XMI resource factory for the .enterprise extension.This will cause all documents with this extension to be treated as XMI documents.
- Obtains a resource set.
- Loads one of the resources explicitly.
- Prints out both of the resources using the print utility that is described in Section 3.6, “Using Reflective APIs,” on page 47.

The output of this method is shown below. Note that only megacorp.enterprise resource is explicitly loaded by this code, but the output includes both megacorp.enterprise and acme.enterprise. The megacorp.enterprise resource includes a reference to an object in the acme.enterprise document. Therefore, the process of printing out megacorp.enterprise forces acme.enterprise to be demand loaded. In the above code above, `resList` initially contains one resource, but during the call to `UGRefPrint`.print, the second resource is added to this list.

```
-----------------------------------------
Contents of resource megacorp.enterprise
-----------------------------------------

Company:
   name: Mega Corp
   department:
      number: 99
      company:
         name: Mega Corp
      employee:
         name: Jane Doe
         department:
            number: 99
   subsidiary:
      name: ACME

-----------------------------------------
Contents of resource acme.enterprise
-----------------------------------------

Company:
   name: ACME
   department:
      number: 101
      company:
         name: ACME
      employee:
         name: John Smith
         department:
            number: 101
   parent:
      name: Mega Corp
```

3.5 Handling notifications

3.5.1 Defining Observers

<tbd>To be done</tbd>

3.5.2 Attaching Observers to Your Objects

<tbd>To be done</tbd>

3.6 Using Reflective APIs

EMF provides APIs that enable you to access your data reflectively. This means that you can view and manipulate EMF data without having any prior knowledge of the model.

(See “Examining EObject Instances using Reflection” on page 48 and “Modifying EObjects using reflection” on page 53.) Also, you can dynamically create new classes (See “Creating New Dynamic Classes” on page 53.) or extend classes that have been generated (See “Extending Generated Classes with Dynamic Classes” on page 53.).

3.6.1 Examining EObject Instances using Reflection

The methods described in this section can be used to print out the contents of an `EObject` without having any prior knowledge of the structure of that `EObject`. The methods defined here are all static and are assumed to be in the same class.

The only public method in this class is print, which takes a collection of `EObject` objects and displays the contents of the objects in System.out.

The `print`() method invokes `printObject`() to display each object in the `Collection`. The `printObject()` method prints the name of the object and then displays the contents of the object by invoking `printAllAttributes` and `printAllReferences`.

3.6.1.1 `print`

The print method invokes `printObject`() to display each object in the `Collection`. This can be any `Collection` that contains `EObject` objects. For example, it might be the contents of a `Resource`.

```
static public void print(Collection list) {

    Iterator iter = list.iterator();
    while (iter.hasNext()) {
       Object object = iter.next();
       if (object instanceof EObject)
       printObject()(0, (EObject)object, null, true);
    }
}
```

Where:

- `list` is the collection of `EObject` instances to be printed. (For example, this could be the collection returned by the `getContents()` method of a `Resource` object.

3.6.1.2 `printObject`

The `printObject()` method can be called either on a root object or it can be called to display the target of a reference. When a root object is printed, the name for the object will be the name of the object’s class. When a reference is printed, the name will be the name of the reference.

All the attributes of the object are displayed by calling `printAllAttributes`().

The `printReferences` argument that is passed to `printObject`() is a flag to indicate whether or not to display the references that belong to the object. In the case of a root object, the references are always displayed. For a non-root object, the references will be displayed if the object is being printed as part of its container. (This is needed to prevent the possibility of infinite recursion when invoking `printObject`().)

If the `printReferences` flag is true, the references are displayed by calling `printAllReferences()`.

```
static private void printObject(
    int tabIndex,
    EObject eObject,
    EReference referenceObj,
    boolean printReferences) {
    if (tabIndex != 0) {
        System.out.println();
        for (int i = 0; i < tabIndex; i++)
            System.out.print("\t");
    }
    ENamedElement nameObj =
        (referenceObj == null)
            ? (ENamedElement) eObject.eClass()
            : referenceObj;
    System.out.println(nameObj.getName() + ": " );

    printAllAttributes()(tabIndex + 1, eObject);

    if (printReferences)
        printAllReferences()(tabIndex, eObject);
}
```

Where:

- `tabIndex` is an integer that controls the indentation of the output line
- `eObject` is the object to be printed.
- `referenceObj` is the `EReference` that was traversed to access `eObject`. If `referenceObj` is **`null`**, the `eObject` is a root object.
- The `printReferences` flag indicates whether the output for the `eObject` should include the objects that the `eObject` references. (The references are only printed for containments.)

3.6.1.3 `printAllAttributes`

The `printAllAttributes()` method first accesses the meta object for a given object and then accesses and traverses the list of attributes that belong to the meta object. The `printAttribute()` method is invoked for each attribute to print out the appropriate value, if it exists.

```
static private void printAllAttributes(int tabIndex, EObject eObject) {
   EClass eMetaObject = eObject.eClass();
   if (eMetaObject == null)
      return;

   Collection attrs = eMetaObject.getEAllAttributes();
   if (attrs == null)
      return;
   Iterator iAttr = attrs.iterator();

   while (iAttr.hasNext()) {
      EAttribute eAttr = (EAttribute) iAttr.next();
      printAttribute()(tabIndex, eObject, eAttr);
   }
}
```

Where:

- `tabIndex` is an integer that controls the indentation of the output line
- `eObject` is the object to be printed.

3.6.1.4 `printAttribute`

The `printAttribute()` method displays the value for a single attribute, if it exists. The value of the attribute is obtained by calling the reflective method `EObject.eGet(EStructuralFeature)`. Note that if the attribute is a single-valued attribute, the value will be a single `Object`. Otherwise it will be a `Collection` of objects.

```
static private void printAttribute(
   int tabIndex,
   EObject eObject,
   EAttribute eAttr) {
   if (!eObject.eIsSet(eAttr)) {
      return;
   }

   Object value = eObject.eGet(eAttr);

   if (eAttr.isVolatile() || (value == null))
      return;

   String valueS = "";
   if (eAttr.isMany()) {
      Iterator vals = ((Collection) value).iterator();
      while (vals.hasNext()) {
         if (valueS.length() > 0)
            valueS += ", ";
         valueS += vals.next().toString();
      }
   }
   else
      valueS = value.toString();
   for (int i = 0; i < tabIndex; i++)
      System.out.print("\t");

   System.out.println(eAttr.getName() + ": " + valueS);
   return;
}
```

Where:

- `tabIndex` is an integer that controls the indentation of the output line
- `eObject` is the object to be printed.
- `eAttr` is the attribute to be printed.

3.6.1.5 `printAllReferences`

The `printAllReferences()` method first accesses the meta object for a given object and then accesses and traverses the list of references that belong to the meta object. The `printReference()` method is invoked for each reference to print out the appropriate object, if it exists.

```
static private void printAllReferences(int tabIndex, EObject eObject) {
   EClass eMetaObject = eObject.eClass();
   if (eMetaObject == null)
      return;

   Collection refs = eMetaObject.getEAllReferences();

   if (refs == null)
      return;
   Iterator iRef = refs.iterator();

   while (iRef.hasNext()) {
      EReference ref = (EReference) iRef.next();
      printReference()(tabIndex, eObject, ref);
   }
}
```

Where:

- `tabIndex` is an integer that controls the indentation of the output line
- `eObject` is the object to be printed.

3.6.1.6 `printReference`

The `printReference()` method displays the value for a single reference, if it exists. The value of the reference is obtained by calling the reflective method `EObject.eGet(EStructuralFeature)`. Note that if the reference is a single-valued reference, the value will be a single `Object`. Otherwise it will be a `Collection` of objects.

The target of the reference is printed out by calling `printObject()` recursively. Note that for containment references, we want to print the contained object plus all of its references while for non-containment references, we only want to print the object. This will prevent the possibility of infinite recursion.

```
static private void printReference(
   int tabIndex,
   EObject eObject,
   EReference ref) {
   Object value = eObject.eGet(ref);
   if (ref.isVolatile() || (value == null))
      return;

   if (ref.isMany()) {
      Iterator vals = ((Collection) value).iterator();
      while (vals.hasNext()) {
         EObject eValue = (EObject)vals.next();
         if (eValue==null)
            return;
         boolean printNestedReferences =
            eValue.eContainer() == eObject;
         printObject(tabIndex + 1, eValue, ref, printNestedReferences);
      }
   }
   else {
      EObject eValue = (EObject)value;
      boolean printNestedReferences = eValue.eContainer() == eObject;
      printObject(tabIndex + 1, eValue, ref, printNestedReferences);
   }
}
```

Where:

- `tabIndex` is an integer that controls the indentation of the output line
- `eObject` is the object to be printed.
- `ref` is the reference to be printed.

3.6.2 Modifying EObjects using reflection

<tbd> to be done </tbd>

3.6.3 Creating New Dynamic Classes

<tbd> to be done </tbd>

3.6.4 Extending Generated Classes with Dynamic Classes

<tbd> to be done </tbd>

3.7 Customizing EMF

3.7.1 Creating Keys to Access the Contents of a `Resource`

<tbd> to be done </tbd>

3.7.2 Cross File References and Proxies

<tbd> to be done </tbd>

3.7.3 Customizing the `Resource` for non-XMI Serialization

<tbd> to be done </tbd>

3.7.4 Handling XMI Documents Serialized from a Different Version of Your Model

<tbd> to be done </tbd>

3.7.5 Customizing a `URIConverter`

Suppose you have special rules for resolving relative URIs. You can implement those rules by creating your own implementation of the `org.eclipse.emf.ecore.resource.URIConverter` interface and attaching it to the `ResourceSet` that will be used to load and save the resource.

For example, suppose you would like all relative URIs to resolve to a specific location on your file system. Your implementation of `URIConverter` could look like this:

```
public class UGURIConverterImpl extends URIConverterImpl
{
  private URI baseURI=null;

 /**
  * Construct a UGURIConverterImpl from a specified base uri
  */
  public UGURIConverterImpl(String base)
  {
   if (base!=null) baseURI=URI.createURI(base);
  }

  /**
   * Normalize the uri.
   * <p>
   * If the uri is relative and if the baseURI has been specified,
   * simply resolve the uri against the base.
   * Otherwise defer to the super classs's implementation.
   */
  public URI normalize(URI uri)
  {
    if (uri.isRelative() && baseURI!=null) {
       return uri.resolve(baseURI);
    }
    return super.normalize(uri);
  }

  /**
   * Creates an output stream and returns it.
   * <p>
   * If the normalized uri is a file scheme, use the normalized uri to
   * construct the output stream directly. Otherwise defer to the super classs's
   * implementation.
   */
  public OutputStream createOutputStream(URI uri) throws IOException
  {
    URI converted = normalize(uri);
    String scheme = converted.scheme();
    if ("file".equals(scheme))
    {
      return createFileOutputStream(converted.toFileString());
    }
    return super.createOutputStream(uri);
  }

  /**
   * Creates an input stream and returns it.
   * <p>
   * If the normalized uri is a file scheme, use the normalized uri to
   * construct the input stream directly. Otherwise defer to the super classs's
   * implementation.
   */
  public InputStream createInputStream(URI uri) throws IOException
  {
    URI converted = normalize(uri);
    String scheme = converted.scheme();
    if ("file".equals(scheme))
    {
      return createFileInputStream(converted.toFileString());
    }
    return super.createInputStream(uri);
  }
} // URIConverterImpl
```

4.0 Quick Reference

The following sections provide reference information:

- See “Ecore Properties and Codegen Specifications” on page 56 for an description of all the properties that you may need to use when generating code.
- See “EMF APIs” on page 76 for a link to information on using the EMF APIs.

4.1 Ecore Properties and Codegen Specifications

The code patterns used by the EMF code generation utility are determined by the properties of the packages, classes, attributes, and relationships that you specify in your model. EMF supports three different formats for the specification of a model, namely, UML, XMI, and Java. Whichever format you use, you will need to be aware of how the model properties are specified in that format.

The following sections enumerate all the properties that apply to each element of an Ecore model. Each section has two tables. The first table lists the properties and how they impact the code generation process and the second table shows how each of these properties is specified in each of the three formats.

Here is an overview of the Ecore model elements:

- An *`EPackage`* (See “EPackage Properties” on page 58.) is a collection of *`EClassifier`* objects. Each package has a package URI which is used to uniquely identify the package.
- An *`EClassifier`* is the description of a type in Ecore. Each *`EClassifier`* is either an *`EClass`* or an *`EDataType`*
    - An *`EClass`* (See “EClass Properties” on page 59.) is a description of a fundamental Ecore data element. Every *`EObject`* is an instance of an *`EClass`*.

      An *`EClass`* may be abstract or concrete and it may derive from other classes. It consists of zero or more *`EStructuralFeatures`* and *`EOperations`*.
    - An *`EDataType`* (See “EDataType Properties” on page 64.) is a description of a type whose values are not Ecore objects. This can be a primitive type, a Java Class that is defined outside of the Ecore model, or an *`EEnum`*.
    - An *`EEnum`* (See “EEnum Properties” on page 62.) is a type that is constructed for a specified list of *`EEnumLiterals`* (See “EEnumLiteral Properties” on page 72.).

- An *`EStructuralFeature`* is a component of an *`EClass`* that describe a field that belongs to the Class. Each *`EStructuralFeature`* is either an *`EAttribute`* (See “EAttribute Properties” on page 66.) or an *`EReference`* (See “EReference Properties” on page 69.).

  Each *`EAttribute`* or *`EReference`* has a type (i.e. either an *`EClass`* or an *`EDataType`*) and may also have other properties that define its cardinality, changeability, default value (if any), persistence, etc.

- An *`EOperation`* (See “EOperation Properties” on page 74.) is a component of an *`EClass`* that describes a method belonging to the class.

  Each *`EOperation`* has a type (which may be an *`EClass`*, an *`EDataType`*, or null) and also has zero or more *`EParamter`* objects (See “EParameter Properties” on page 75.).

4.1.1 EPackage Properties

The properties of an `EPackage` in Ecore are:

TABLE 1. Ecore Properties for EPackage

Property	Usage	Default
name	The name of the package. This name is used as the name of the generated package Interface.	No default.
nsURI	The Namespace URI of the package, i.e. the URI that appears in the xmlns tag to identify this package in an XMI document.	nsName with a suffix of ".xmi"
nsPrefix	The Namespace prefix that is used when references to instances of the classes in this package are serialized.	The nsName with the first character converted to upper case.
eClassifiers	The classes, enumerations and datatypes contained in the package. (See "EClass Properties" on page 59, "EEnum Properties" on page 62, and "EDataType Properties" on page 64.)	empty
eSubpackages	The nested packages. This information is used to construct the default names and namespace URIs for the subpackages. Also, a package and its subpackages are treated as a group for the purposes of initialization, so that when one package is initialized, all the other packages in the group will also be initialized.	none
prefix	Used as the prefix for the names of the generated Factory and Package classes.	Same as package name specified in the model
basePackage	The prefix used for the Java package that contains the generated code for the model.	"" (i.e., the empty string)

These properties are specified to EMF code generation in one of the following ways:

- UML - The properties are set via a package object in a UML diagram or via the specification dialog box for the package. (See Section 3.1.1.7, "Ecore Properties for Packages," on page 19 for an example.)
- XMI - Most of these properties are specified as attributes or sub-elements of the ecore:EPackage element in the ecore document. The ecore:EPackage document is typically the root of the document. (See Section 3.1.2.2, "The Enterprise Package Element in the Ecore Document," on page 23 for an example.) Some of the package properties are specified in the genPackage element of the genmodel document (See Section 3.1.2.1, "Genmodel Document for the enterprise Model," on page 22 for an example.)
- Java - The properties are implicitly derived from the java package that contains the various **`interface`** declarations that define the classes in your model.

Alternatively, you can specify an explicit `interface` statement for a package or from the `@model` tag that precedes the `interface` statement. You could do this if you want to override some of the properties of the package. However, you may find that the easiest way to do this is to allow the code generation utility to automatically create the initial version of this `interface` statement, after which you can make the modifications you require. (See Section 3.1.3.1, "Java Specification for the Enterprise Package," on page 28 for an example.)

TABLE 2. Codegen Specifications for EPackage Properties

Property	**UML**	**XMI**	**Java**
name	name of the package in the UML diagram or the *packageName* property[c]	name attribute[a]	Implicitly derived from the Java package[b] .
nsURI	*nsURI* property[c]	nsURI attribute[a]	The initial value of the `eNSURI` field[d]
nsPrefix	*prefix* property[c]	nsPrefix attribute[a]	The initial value of the `eNSPrefix` field[d]
eClassifiers	The classes, enumerations, and datatypes that are contained in the UML Package	eClassifiers element[a]	Derived from the interfaces, classes and datatypes within this Java package[b]
eSubpackages	Nested packages[c]	eSubpackages element[a]	n/a[e]
prefix	*prefix* property[c]	prefix attribute[f]	The prefix part of the name of the Java package[b]
basePackage	*basePackage* property[c]	basePackage attribute[f]	The base part of the Java package[b]

---

[a] Specified on the ecore:Package element in the ecore document

[b] This is the Java package that is specified on the package statement of the interfaces and/or classes contained in the package.

[c] This property is specified on the eCore page of the specification dialog for the UML Package.

[d] This field is a member of the interface that corresponds to the package itself.

[e] Subpackages cannot be specified if you use Java interfaces to specify your model.

[f] Specified on the subPackages element in the genmodel document.

4.1.2 EClass Properties

The properties of an *EClass* in Ecore are:

TABLE 3. Ecore Properties for EClass

Property	Usage	Default
`name`	Used to construct the names of the generated interface and implementation class. (The name of the implementation class has a suffix of “`Impl`”)	no default
`instanceClass`	Used by the EMF runtime to validate the type of objects on a type-safe list. For non-dynamic classes, this is always the generated interface. **null** indicates a dynamic class.	the generated interface
`defaultValue`	The intrinsic default value for a class. This default will be applied to any attributes of the class. *Note: this property cannot be modified for EClass objects. It’s value is always* ***`null`***.	***`null`***
`abstract`	If ***`true`***, the generated implementation class will have the **`abstract`** keyword	***`false`***
`interface`	If ***`true`***, only the java **`interface`** will be generated. There will be no corresponding implementation class and no create method in the factory.	***`false`***
`eAttributes`	The attributes associated with the class. Used to construct the accessor methods for the interface and implementation of the class.[a] (See “EAttribute Properties” on page 66.)	none
`eReferences`	The attributes associated with the class. Used to construct the accessor methods for the interface and implementation of the class.[a] (See “EReference Properties” on page 69.)	none
`eOperations`	The attributes associated with the class. Used to construct the additional methods that are part of the class. (Note: code generation creates stubs for the implementations of these methods.) (See “EOperation Properties” on page 74.)	none
`eSupertypes`	The supertypes for this class. Used to construct the **`extends`** clauses of the generated **`interface`** and **`class`** statements. Note: the generated interface will extend from all the interfaces for all the supertypes. However, the generated implementation class will only extend from the implementation class of the first supertype in the list.	none

---

[a] Depending on the properties of the attribute or reference, the accessor methods may be `get()`, `set()`, `isSet()` and `unset()`. Usually, the implementations of these methods are generated automatically.

These properties are specified to EMF code generation in one of the following ways:

- UML - The properties are set via a UML class object or via the specification dialog box for that class. (See Section 3.1.1.1, "Basic UML Model Elements," on page 10 for an example.)
- XMI - The properties are specified as attributes or sub-elements of an ecore:EClass element in the XMI document. The ecore:EClass is typically one of the eClassifiers sub-elements of the ecore:EPackage object that is at the root of the XMI document. (See Section 3.1.2.3, "The Company Class Element in the Ecore Document," on page 24 for an example.)
- Java - The properties are derived from the **`interface`** statement for a class or from the `@model` tag that precedes the **`interface`** statement. (See Section 3.1.3.2, "Java Specification for the Company Class," on page 29 for an example.)

TABLE 4. Codegen Specifications for EClass Properties

Property	**UML**	**XMI**	**Java**
`name`	The name of the class in the UML diagram	name attribute[a]	The name of the Java **`interface`**.
`instanceClass`	*n/a*[b]	*n/a*[b]	*n/a*[b]
`defaultValue`	*n/a*[c]	*n/a*[c]	*n/a*[c]
`abstract`	The *`abstract`* property on the UML class[d].	abstract attribute[a]	`abstract` property[e]
`interface`	The *`<<interface>>`* stereotype on the UML Class	interface attribute[a]	`interface` property[e]
`attributes`	All the attributes associated the class	attributes element[a]	All the `get()` methods on the **`interface`** that have a `@model` tag and whose return type is a primitive type
`references`	All the relations associated with the class	references element[a]	All the `get()` methods on the **`interface`** that have a `@model` tag and whose return type is an Ecore class

TABLE 4. Codegen Specifications for EClass Properties

Property	UML	XMI	Java
`operations`	All the operations associated with the class	operations element[a]	Any method that is flagged with an `@model` tag and is not the `get()` method for and attribute or reference[f]
`supertypes`	All the generalizations associated with the class	supertypes element[a]	All the classes that are listed in the **`extends`** clause of the **`interface`** statement.

---

[a] Specified on the eClassifiers element that has an xsi:type of ecore:EClass in the ecore document

[b] You do not specify the instanceClass property explicitly. The value is always the generated interface.

[c] You do not specify the default value property explicitly. The value is always null.

[d] This property is set in Rational Rose using the “Abstract” checkbox on the “Details” page of the “Specification” dialog for a class.

[e] The property is specified via the `@model` tag that precedes the **`interface`** statement for the class.

[f] If there is potential ambiguity with a `get()` method, you need to specify the “`parameters=`” attribute to give signature of the method.

4.1.3 EEnum Properties

The properties of an *`EEnum`* in Ecore are:

TABLE 5. Ecore Properties for EEnum

Property	Usage	Default
`name`	Used to construct the name of the generated **`public final class`**	no default
`instanceClass`	Used by the EMF runtime to validate the type of objects on a type-safe list. For non-dynamic classes, this is always the generated enumeration class. **null** indicates a dynamic class.	the generated enumeration class
`defaultValue`	The intrinsic default value for an enumeration. This default will be applied to any attributes of the enumeration type that do not specify an explicit default. *Note: this property cannot be modified for EEnum objects. It’s value is always the first enumerator.*	first enumerator

TABLE 5. Ecore Properties for EEnum

Property	Usage	Default
`serializable`	Controls whether or not the generated factory will contain `convertToString()` and `createFromString()` methods for a datatype. *Note: this property cannot be modified for EEnum objects. It's value is always **`true`**.*	***`true`***
`eLiterals`	The literals associated with this enumeration. Used to construct the **`final static`** integers and literals that comprise the generated **`class.`** (See "EEnumLiteral Properties" on page 72.)	none

These properties are specified to EMF code generation in one of the following ways:

- UML - The properties are set via a UML class object that has a stereotype of *`<<enumeration>>`*, or via the specification dialog box for that class. (See Section 3.1.1.3, "Attribute Specifications in UML," on page 13 for an example.)
- XMI - The properties are specified as attributes or sub-elements of an ecore:EEnumeration element in the XMI document. The ecore:EEnumeration is typically one of the eClassifiers sub-elements of the ecore:EPackage object that is at the root of the XMI document. (See Section 3.1.2.7, "The EmploymentType Enumeration Element in the Ecore Document," on page 26 for an example.)
- Java - The properties are derived from the **`class`** statement for an enumeration or from the `@model` tag that precedes the **`class`** statement. Note: this **`class`** statement must be preceded by a `@model` tag and should flagged as **`public`** and **`final`** and should **`extend`** the `org.eclipse.emf.common.util.AbstractEnumerator` class. (See Section 3.1.3.6, "Java Specification for the EmploymentType Enumeration," on page 33 for an example.)

TABLE 6. Codegen Specification for EEnum Properties

Property	UML	XMI	Java
`name`	The name of a UML class that has the *`<<enumeration>>`* stereotype	name attribute[a]	The name of enumeration class. (An enumeration class is any java **`class`** that is preceded by a `@model` tag.)
`instanceClass`	*n/a*[b]	*n/a*[b]	*n/a*[b]
`defaultValue`	*n/a*[c]	*n/a*[c]	*n/a*[c]

TABLE 6. Codegen Specification for EEnum Properties

Property	UML	XMI	Java
`serializable`	*n/a*[d]	*n/a*[d]	*n/a*[d]
`eLiterals`	All the attributes of the UML class	eLiterals element[a]	All variables of type **`int`** that are preceded by a `@model` tag. [e]

---

a. Specified on an eClassifiers element that has an xsi:type of ecore:EEnum in the ecore document

b. You do not specify the instanceClass property explicitly. The value is always the generated class.

c. You do not specify the default value property explicitly. The value is always the first entry on the eLiterals list.

d. You do not specify the serializable property explicitly for enumerations. The value of the this property is always true.

e. The `@model` tag may have a `name=` argument, but should not have any other arguments.

4.1.4 EDataType Properties

The properties of an *`EDataType`* in Ecore are:

TABLE 7. Ecore Properties for EDataType

Property	Usage	Default
`name`	Used to construct the name of the `get()` method in the package that accesses the datatype.	no default
`instanceClass`	Used by code generation in constructing the signature of accessor methods that are generated for attributes that are typed to this datatype. Also used by the EMF runtime to validate the type of objects on a type-safe list.	no default
`defaultValue`	The intrinsic default value for a datatype. This default will be applied to any attributes of the datatype that do not specify an explicit default.	For java primitive types, the appropriate Java default for the primitive; Otherwise, ***`null`***
`serializable`	Controls whether or not the generated factory will contain `convertToString()` and `createFromString()` methods for a datatype. *Note: If the serializable flag is **`false`** for a datatype, all attributes of that datatype must be transient.*	*`true`*

These properties are specified to EMF code generation in one of the following ways:

- UML - The properties are set via a UML class object that has a stereotype of `<<datatype>>`, or via the specification dialog box for that class. (See Section 3.1.1.3, “Attribute Specifications in UML,” on page 13 for an example.)
- XMI - The properties are specified as attributes or sub-elements of an ecore:EDataType element in the XMI document. The ecore:EDataType is typically one of the eClassifiers sub-elements of the ecore:EPackage object that is at the root of the XMI document. (See Section 3.1.2.8, “The Date Datatype Element in the Ecore Document,” on page 27 for an example.)
- Java - Any usage of a type that is not an EMF type will be implicitly treated as a datatype. For example, if one of the attributes or methods in your model uses a type that is not defined in your model (i.e. there is not interface with the @model tag to define that type) the type will be treated as a datatype.

  Alternatively, you can define a datatype explicitly by adding a `get()` method to the package that defines the type. The return type of this `get()` method must be `org.eclipse.emf.ecore.EDataType` and the method must be preceded by a `@model` tag. (See Section 3.1.3.1, “Java Specification for the Enterprise Package,” on page 28 for an example.)

TABLE 8. Codegen Specifications for EDataType Properties

Property	**UML**	**XMI**	**Java**
`name`	The name of a UML class that has the *`<<datatype>>`* stereotype	name attribute[a]	The name of `get()` method, without the “`get`” prefix.
`instanceClass`	The name of an attribute of the class which has the *`<<javaclass>>`* stereotype	instanceClass attribute[a]	The `instanceClass` property[b]
`defaultValue`	*n/a*[c]	*n/a*[c]	*n/a*[c]
`serializable`	The *`abstract`* property on the UML class[d].	serializable attribute[a]	The `serializable` property[b]

---

[a] Specified on an eClassifiers element that has an xsi:type of ecore:EDataType in the ecore document

[b] This property is specified via the `@model` tag that precedes the `get()` method that defines the datatype.

[c] You do not specify the default value property explicitly. For java primitive types, the value is the appropriate Java default for the primitive; Otherwise it is null.

[d] This property is set in Rational Rose using the “Abstract” checkbox on the “Details” page of the “Specification” dialog for a class.

4.1.5 EAttribute Properties

The properties of an `EAttribute` in Ecore are:

TABLE 9. Ecore Properties for EAttribute

Property	Usage	Default
`name`	Name used to construct the names of accessor methods	no default
`eType`	The type of the attribute. *Note: this must be an EDatatype*.	no default
`changeable`	Indicates whether the attribute may be modified. If `changable` is **`true`**, a `set()` method is generated for the attribute. Otherwise, no `set()` method is generated.	*true*
`volatile`	Indicates whether the attribute cannot be cached. If `volatile` is **`true`**, the generated class does not contain a field to hold the attribute and the generated `get()` and `set()` methods for the attribute are empty. In this case you should provide your own implementation of the accessor methods. Otherwise, the default implementations for these methods will provide the expected behavior.	*false*
`transient`	Indicates whether the attribute should not be stored. If `transient` is **`true`**, the XMI serializer will not write this attribute out when the class is serialized. Otherwise, the attribute will be serialized.	*false*
`unique`	Indicates whether a many-valued attribute is allowed to have duplicates. If `unique` is **`true`**, the implementation of the list that is used to contain the values will enforce uniqueness.	*true*
`defaultValue`	Determines the value returned by the get method if the attribute has never been set.	*no default*
`lowerBound`	Determines the setting of the `required` property (see below). If `lowerBound` is `0`, the `required` property will be set to **`false`**. Otherwise, the `required` property will be **`true`**.	*0*

TABLE 9. Ecore Properties for EAttribute

Property	Usage	Default
`upperBound`	Determines the setting of the `many` property (see below). If `upperBound` is `1`, the `many` property will be set to **`false`**. Otherwise, the `many` property will be **`true`**.	*1*
`many`	If `many` is **`true`**, there is no `set()` method for the attribute and the `get()` method returns a list that can only contain objects of the appropriate type. Otherwise, both `get()` and `set()` methods are generated and they return and receive a reference to a single object of the appropriate type.	*false*
`required`	Indicates whether the attribute is required. *Note: this property has no impact on code generation or on the EMF runtime. This property is has the potential to be useful for validation.*	*false*
`unsettable`	Indicates that the attribute may be unset. If `unsettable` is **`true`**, an `isSet()` method is generated for the attribute. Note that this requires additional runtime storage for the class	*false*

These properties are specified to EMF code generation in one of the following ways:

- UML - The properties are set by via an attribute belonging to a UML class object or via the specification dialog box for that attribute. (See Section 3.1.1.1, “Basic UML Model Elements,” on page 10 and Section 3.1.1.4, “The eCore Properties Page,” on page 15 for examples.)
- XMI - The properties are specified as attributes or sub-elements of an ecore:EAttribute element in the XMI document. The ecore:EAttribute is typically one of the eAttributes sub-elements of an ecore:EClass object, which in turn is one of the eClassifiers sub-elements of the ecore:EPackage object that is at the root of the XMI document. (See Section 3.1.2.6, “The Employee Class Element in the Ecore Document,” on page 25 for examples.)
- Java - The properties are derived from the `get()` method in the **`interface`** that defines the class to which this attribute belongs. This `get()` method must be preceded by a `@model` tag. (See Section 3.1.3.5, “Java Specification for the Employee Class,” on page 32 for examples.)

TABLE 10. Codegen Specifications for EAttribute

Property	UML	XMI	Java
`name`	the name of the UML attribute	name attribute[a]	name of `get()` method, without the “`get`” prefix
`eType`	the type of the UML attribute	eType attribute[a]	For single-valued attributes, the return type of the get() method. Otherwise, the eType property[c]
`changeable`	*`isChangeable`* property[b]	changeable attribute[a]	`changeable` property[c]
`volatile`	*`isVolatile`* property[b]	volatile attribute[a]	`volatile` property[c]
`transient`	*`isTransient`* property[b]	transient attribute[a]	`transient` property[c]
`unique`	*`isUnique`* property[b]	unique attribute[a]	`unique` property[c]
`defaultValue`	The initial value assigned to the attribute	defaultValue attribute[a]	`defaultValue` property[c]
`lowerBound`	cardinality stereotype[d]	lowerBound attribute[a]	The `lowerBound` property[c]
`upperBound`	cardinality stereotype[d]	upperBound attribute[a]	The `upperBound` property[c], if it exists; Otherwise, the return type of the get method[e]
`many`	*n/a*[f]	*n/a*[f]	*n/a*[f]
`required`	*n/a*[g]	*n/a*[g]	*n/a*[g]
`unsettable`	isUnsettable property[b]	unsettable attribute[a]	`unsettable` property[c]

---

[a] Specified on an eAttributes element in the ecore document

[b] This property is specified on the eCore page of the specification dialog for the UML Attribute

[c] This property is specified via the `@model` tag that precedes the `get()` method that defines the `get()` method that defines the attribute.

[d] The cardinality stereotype is specified as *`<<lowerBound..upperBound>>`* where *`lowerBound`* is either *`0`* or *`1`* and *`upperBound`* is either *`1`* or *`*`*. If the cardinality stereotype is omitted, *`<<0..1>>`* is assumed.

[e] A return type of `java.util.List` or `org.eclipse.emf.common.util.EList` indicates an upperBound of “-1” (which means there is no upper bound.) Any other type indicates an upperBound of “1”.

[f] You do not specify the *`many`* property explicitly. The value is derived from the *`upperBound`*.

[g] You do not specify the *`required`* property explicitly. The value is derived from the *`lowerBound`*.

4.1.6 EReference Properties

The properties of an `EReference` in Ecore are:

TABLE 11. Ecore Properties for EReference

Property	**Usage**	**Default**
`name`	Name used to construct the names of accessor methods	no default
`eType`	The type of the reference. In the case of single-valued references, the `eType` is the return type of the generated `get()`method. For multi-valued references, the `eType` is the type of objects that are allowed on the type-safe list that is returned by the `get()`method. *Note: for references, the eType must be an EClass*	no default
`changeable`	Indicates whether the reference may be modified. If changeable is *false*, no `set()` method is generated for the reference	*true*
`volatile`	Indicates whether the reference cannot be cached. If volatile is *true*, the generated class does not contain a field to hold the reference and the generated `get()` and `set()` methods for the reference are empty. In this case you should provide your own implementation of the accessor methods.	*false*
`transient`	Indicates whether the reference should not be stored. If transient is *true*, the XMI serializer will not write this reference out when the class is serialized.	*false*
`unique`	Indicates whether a many-valued attribute is allowed to have duplicates. If `unique` is **`true`**, the implementation of the list that is used to contain the values will enforce uniqueness. *Note: The setting of the* `unique` *is always* **`true`** *for references.*	*true*
`defaultValue`	Determines the value returned by the get method if the attribute has never been set. *Note: The* `defaultValue` *property is always* **`null`** *for references. It cannot be modified.*	

TABLE 11. Ecore Properties for EReference

Property	Usage	Default
`lowerBound`	Determines the setting of the *`required`* property (see below). If *`lowerBound`* is *`0`*, the *`required`* property will be set to ***`false`***. Otherwise, the *`required`* property will be ***`true`***.	*0*
`upperBound`	Determines the setting of the *`many`* property (see below). If *`upperBound`* is *`1`*, the *`many`* property will be set to ***`false`***. Otherwise, the *`many`* property will be ***`true`***.	*1*
`many`	Indicates whether the reference is single-valued or multi-valued. If *`many`* is ***`true`***, there is no `set()` method for the attribute and the `get()` method returns a list that can only contain objects of the appropriate type. Otherwise, both `get()` and `set()` methods are generated and they return and receive a reference to a single object of the appropriate type.	*false*
`required`	Indicates whether the reference is required. *Note: this property has no impact on code generation or on the EMF runtime. This property is has the potential to be useful for validation.*	*false*
`containment`	Indicates whether the reference is a containment. If containment is true, the generated accessor methods will enforce containment semantics. (E.g., if you add an object to a new container, that object will be automatically removed from any existing container.	***`false`***
`container`	Indicates whether the reference is a container. This is the opposite of a containment EReference. If container is true, the generated accessor methods will have container semantics.	***`false`***
`resolveProxies`	Indicates whether proxy references should be resolved automatically.	***`true`***
`eOpposite`	Identifies the EReference that represents the opposite end of the relationship. This is used by the EMF runtime to preserve bidirectional referential integrity. (E.g., if you set one end of a relationship, the opposite end will be set automatically.)	***`null`***

These properties are specified to EMF code generation in one of the following ways:

- UML - The properties are set by via a one of the roles of a relation belonging to a UML class object or via the specification dialog box for that role. (See Section 3.1.1.1, “Basic UML Model Elements,” on page 10 and Section 3.1.1.4, “The eCore Properties Page,” on page 15 for examples.)

- XMI - The properties are specified as attributes or sub-elements of an ecore:EReference element in the XMI document. The ecore:EReference is typically one of the eReferences sub-elements of an ecore:EClass object, which in turn is one of the eClassifiers sub-elements of the ecore:EPackage object that is at the root of the XMI document. (See Section 3.1.2.3, “The Company Class Element in the Ecore Document,” on page 24 for examples.)
- Java - The properties are derived from the `get()` method in the **`interface`** that defines the class to which this attribute belongs. The method must be preceded by a `@model` tag. (See Section 3.1.3.2, “Java Specification for the Company Class,” on page 29 for examples.)

TABLE 12. Codegen Specifications for EReference Properties

Property	**UML**	**XMI**	**Java**
`name`	The name of the UML relation.	name attribute[a]	The name of `get()` method, without the “`get`” prefix.
`eType`	The type of the UML relation.	eType attribute[a]	For single-valued attributes, the return type of the get() method. Otherwise, the eType property[c]
`changeable`	*`isChangeable`* property[b]	changeable attribute[a]	The `changeable` property[c]
`volatile`	*`isVolatile`* property[b]	volatile attribute[a]	The `volatile` property[c]
`transient`	*`isTransient`* property[b]	transient attribute[a]	The `transient` property[c]
`unique`	*`isUnique`* property[b]	unique attribute[a]	The `unique` property[c]
`defaultValue`	The initial value assigned to the attribute	defaultValue attribute[a]	The defaultValue property[c]
`lowerBound`	cardinality[d]	lowerBound attribute[a]	The `lowerBound` property[c]
`upperBound`	cardinality[d]	upperBound attribute[a]	The `upperBound` property[c], if it exists; Otherwise, the return type of the get method[e]
`many`	*n/a*[f]	*n/a*[f]	*n/a*[f]
`required`	*n/a*[g]	*n/a*[g]	*n/a*[g]
`containment`	roles that are marked as aggregates and have by-value containment	containment attribute[a]	The `containment` property[c]

TABLE 12. Codegen Specifications for EReference Properties

Property	UML	XMI	Java
`container`	the class on the owning side of a containment relation	container attribute[a]	The `container` property[c]
`resolveProxies`	*`resolveProxies`* property[b]	resolveProxies attribute[a]	The `resolveProxies` property[c]
`eOpposite`	the relation that represents the opposite role, if it exists	eOpposite attribute[a]	The `eOpposite` property[c]

---

[a] Specified on an eReferences element in the ecore document

[b] This property is specified on either the eCoreA or the eCoreB page of the specification dialog for the UML Association

[c] This property is specified via the `@model` tag that precedes the `get()` method that defines the `get()` method that defines the reference.

[d] The cardinality is specified as *`lowerBound..upperBound`* where *`lowerBound`* is either *`0`* or *`1`* and *`upperBound`* is either *`1`* or *`*`*. If the cardinality is omitted, *`0..*`* is assumed.

[e] A return type of `java.util.List` or `org.eclipse.emf.common.util.EList` indicates an upperBound of “-1” (which means there is no upper bound.) Any other type indicates an upperBound of “1”.

[f] You do not specify the *`many`* property explicitly. The value is derived from the *`upperBound`*.

[g] You do not specify the *`required`* property explicitly. The value is derived from the *`lowerBound`*.

4.1.7 EEnumLiteral Properties

The properties of an *`EEnumLiteral`* in Ecore are:

TABLE 13. Ecore Properties for EEnumLiteral

Property	Usage	Default
`name`	The name is used to generate the final static constants in the enumeration class that are used to access the literal. These names are derived by inserting “_” characters to separate the words in the name and converting the name to upper case. One of the final static constants is the result of this conversion and the other one has the suffix of “_LITERAL”	no default

TABLE 13. Ecore Properties for EEnumLiteral

Property	Usage	Default
`value`	Determines the integer value that is associated with this literal	0
`instance`	Identifies the instance of the Enumerator that defines the value of this enumeration literal. This instance may be assigned to any attributes whose type is the enumeration to which this enumeration literal belongs.	For dynamic ”this”; otherwise the instance of the generated Enumerator

These properties are specified to EMF code generation in one of the following ways:

- UML - The properties are set by via an attribute belonging to a UML class object that has the *<<enumeration>>* stereotype, or via the specification dialog box for that attribute. (See Section 3.1.1.3, “Attribute Specifications in UML,” on page 13 for an example.)
- XMI - The properties are specified as attributes or sub-elements of an ecore:EEnumLiteral element in the XMI document. The ecore:EEnumLiteral is typically one of the eLiterals sub-elements of an ecore:EEnum object, which in turn is one of the eClassifiers sub-elements of the ecore:EPackage object that is at the root of the XMI document. (See Section 3.1.2.7, “The EmploymentType Enumeration Element in the Ecore Document,” on page 26 for an example.)
- Java - The properties are derived from the **`public static final int`** field that defines the enumeration literal within the **`class`** that defines the enumeration to which this literal belongs. The type of this field must be **`int`** and the method must be preceded by a `@model` tag. (See Section 3.1.3.6, “Java Specification for the EmploymentType Enumeration,” on page 33 for an example.)

TABLE 14. Codegen Specifications for EEnumLiteral Properties

Property	UML	XMI	Java
`name`	The name of the attribute that represents the literal	name attribute[a]	The `name` property[b], if it is present, otherwise, the name of the field.
`value`	If specified, the initial value of the attribute. Otherwise, the literals are numbered consecutively, starting at 0.	value attribute[a]	The initial value of the field.
`instance`	n/a[c]	n/a[c]	n/a[c]

---

[a] Specified on an eLiterals element in the ecore document

[b] This property is specified via the `@model` tag that precedes the field that defines the datatype.

[c] You do not set the instance property explicitly. This property is automatically filled in when the package to which the Enumeration belongs is initialized.

4.1.8 EOperation Properties

The properties of an *`EOperation`* in Ecore are:

TABLE 15. Ecore Properties for EOperation

Property	Usage	Default
`name`	The name of the generated method.	no default
`eType`	The return type of the me.thod	null
`eParameters`	The signature of the method. (See “EParameter Properties” on page 75.)	none

These properties are specified to EMF code generation in one of the following ways:

- UML - The properties are set by via an operation belonging to a UML class object, or via the specification dialog box for that operation. (See Section 3.1.1.3, “Attribute Specifications in UML,” on page 13 for an example.)
- XMI - The properties are specified as attributes or sub-elements of an ecore:EOperation element in the XMI document. The ecore:EOperation is typically one of the eOperations sub-elements of an ecore:EClass object, which in turn is one of the eClassifiers sub-elements of the ecore:EPackage object that is at the root of the XMI document. (See Section 3.1.2.6, “The Employee Class Element in the Ecore Document,” on page 25 for an example.)
- Java - The properties are derived from a method that is specified within the **`interface`** that defines the class to which this operation belongs. The method must be preceded by a `@model` tag. (See Section 3.1.3.5, “Java Specification for the Employee Class,” on page 32 for an example.)

  Note: if there is ambiguity with an accessor method (i.e. if the name of the method begins with the prefix “`get`*”), the* `parameters` *property must be specified on the* `@model` *tag to identify an EOperation.*

TABLE 16. Codegen Specifications for EOperation Properties

Property	UML	XM	Java
`name`		name attribute[a]	
`eType`		eType attribute[a]	
`eParameters`		eParameters element[a]	

---

[a] Specified on an eOperations element in the ecore document

4.1.9 EParameter Properties

The properties of an `EParameter` in Ecore are:

TABLE 17. Ecore Properties for EParameter

Property	Usage	Default
`name`	The name of the generated argument	no default
`eType`	The type of the argument	no default

These properties are specified to EMF code generation in one of the following ways:

- UML - The properties are set by via an argument on an operation belonging to a UML class object, or via the specification dialog box for that argument. (See Section 3.1.1.3, "Attribute Specifications in UML," on page 13 for an example.)
- XMI - The properties are specified as attributes of an ecore:EParameter element in the XMI document. The ecore:EParameter is typically one of the eParameters sub-elements of an ecore:EOperation object, which in turn is one of the eOperations sub-elements of an ecore:EClass object, which in turn is one of the eClassifiers sub-elements of the ecore:EPackage object that is at the root of the XMI document. (See Section 3.1.2.6, "The Employee Class Element in the Ecore Document," on page 25 for an example.)
- Java - The properties are derived from the signature of the method to which this parameter belongs. The method must be preceded by a `@model` tag. (See Section 3.1.3.5, "Java Specification for the Employee Class," on page 32 for an example.)

TABLE 18. Codegen Specifications for EParameter Properties

Property	UML	XMI	Java
`name`	the name of the argument that represents this parameter	name attribute[a]	the name of the argument that represents the parameter
`eType`	the type of the argument	eType attribute[a]	the type of the argument

---

[a] Specified on an eParameters element in the ecore document

4.2 EMF APIs

The APIs for the EMF runtime are described in detail in the JavaDoc document for EMF. You can access this document through the “Documents” section of the EMF web site. Please see http://www.eclipse.org/emf/ for details.

5.0 Appendix A - The Ecore Model

FIGURE 11. Ecore Model Class Hierarchy

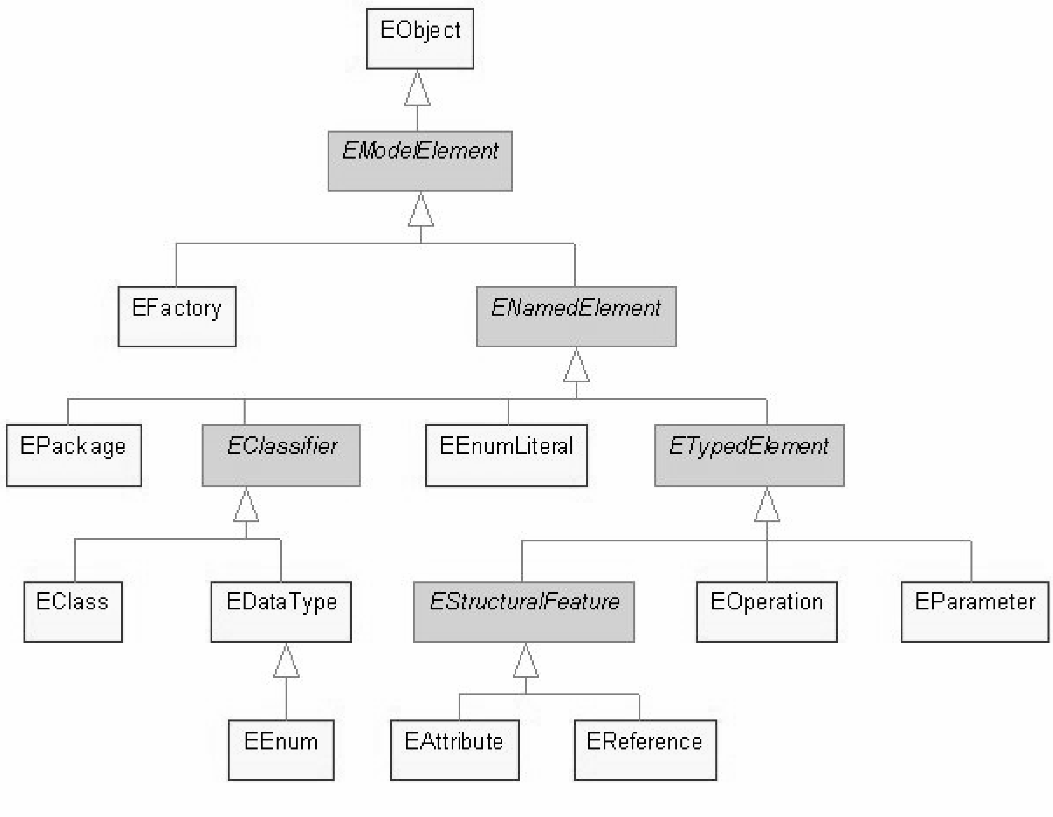
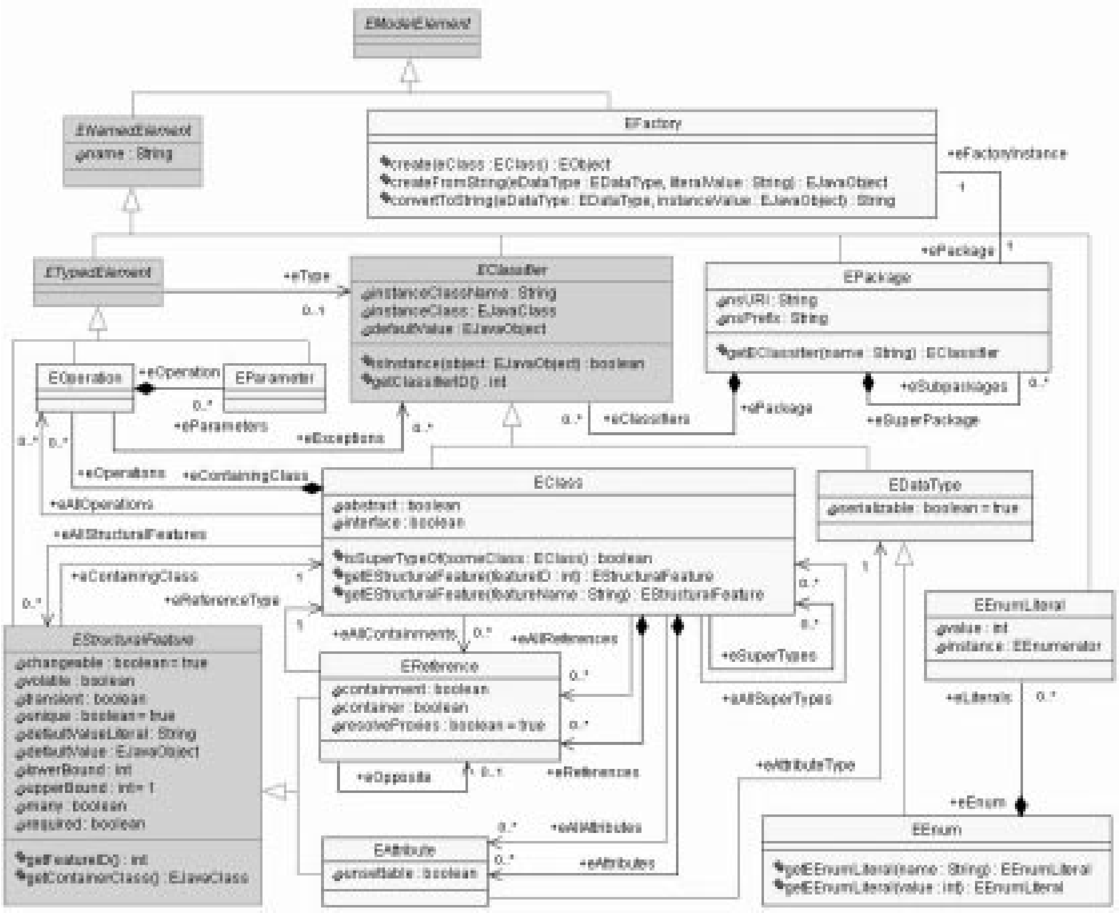


FIGURE 12. Ecore Model Relationships, Attributes, and Operations


EModelElement
ENamedElement
name : String
EFactory
create(eClass : EClass) : EObject
createFromString(eDataType : EDataType, literalValue : String) : EJavaObject
convertToString(eDataType : EDataType, instanceValue : EJavaObject) : String
+eFactoryInstance
1
+ePackage
1
ETypedElement
+eType
0..1
EClassifier
instanceClassName : String
instanceClass : EJavaClass
defaultValue : EJavaObject
isInstance(object : EJavaObject) : boolean
getClassifierID() : int
EPackage
nsURI : String
nsPrefix : String
getEClassifier(name : String) : EClassifier
+eSubpackages
0..*
+eSuperPackage
+ePackage
0..*
+eClassifiers
EOperation
+eOperation
EParameter
0..*
+eParameters
+eExceptions
0..*
0..*
0..*
+eOperations
+eContainingClass
+eAllOperations
EClass
abstract : boolean
interface : boolean
isSuperTypeOf(someClass : EClass) : boolean
getEStructuralFeature(featureID : int) : EStructuralFeature
getEStructuralFeature(featureName : String) : EStructuralFeature
EDataType
serializable : boolean = true
+eAllStructuralFeatures
+eContainingClass
1
+eReferenceType
1
0..*
+eAllContainments
0..*
+eAllReferences
0..*
0..*
+eSuperTypes
+eAllSuperTypes
1
EEnumLiteral
value : int
instance : EEnumerator
EStructuralFeature
changeable : boolean = true
volatile : boolean
transient : boolean
unique : boolean = true
defaultValueLiteral : String
defaultValue : EJavaObject
lowerBound : int
upperBound : int = 1
many : boolean
required : boolean
getFeatureID() : int
getContainerClass() : EJavaClass
EReference
containment : boolean
container : boolean
resolveProxies : boolean = true
0..*
0..*
+eOpposite
0..1
+eReferences
+eAttributeType
+eLiterals
0..*
+eEnum
0..*
+eAllAttributes
EAttribute
unsettable : boolean
0..*
+eAttributes
EEnum
getEEnumLiteral(name : String) : EEnumLiteral
getEEnumLiteral(value : int) : EEnumLiteral