CONTENTS INCLUDE:

Essential EMF

By Ed Merks and James Sugrue

ABOUT THE ECLIPSE MODELING FRAMEWORK

The Eclipse Modeling Framework (EMF) is a powerful framework and code generation facility for building Java applications based on simple model definitions. Designed to make modeling practical and useful to the mainstream Java programmer, EMF unifies three important technologies: Java, XML, and UML. Software is focused on manipulating data that can be modeled, hence, models drive software development. This refcard will get you started with the Eclipse Modeling Framework.

Hot Tip Get the latest release of EMF for Eclipse 3.4 by pointing your Install Manager to http://download.eclipse.org/modeling/emf/updates/releases/

GENERATING A MODEL – QUICK START

Here is a step-by-step overview for creating an EMF model quickly in Eclipse.

Step	Instructions
Create an empty EMF project	Click on **New/Project.../Eclipse Modeling Framework/Empty EMF Project.**
Create your initial model	Using either annotated Java, XML Schema, or UML, define an initial model of your application. This is an optional step. You can create a new Ecore model directly using **New/Other.../Eclipse Modeling Framework/Ecore Model**.
Create EMF model	Click on **New/Other.../Eclipse Modeling Framework/EMF Model** and choose your model importer, e.g., annotated Java, Ecore, Rose, UML, or XML Schema.
Generate model code	From the Generator editor opened on the *.genmodel created in the previous step, use the context menu to invoke **Generate Model Code**. From here you can also generate your test, edit and editor code. Note that the test project includes a simple stand-alone example application for reading and writing instances of your model.
Manipulating the Ecore model	The Ecore model can be changed at any time. The *.genmodel can be reloaded, or resynchronized with changes to the Ecore model, by right-clicking on it in the explorer or navigator and choosing the **Reload...** menu option.
Editing the Ecore model graphically	The Ecore Tools project provides a graphical editor for manipulating Ecore models just like a UML class diagram. From the context menu on the *.ecore invoke **Initialize Ecore Diagram File**.

REGENERATION AND MERGING

The EMF generator produces files that are intended to be a combination of generated pieces and handwritten pieces. You are expected to edit the generated classes to add methods and instance variables. You can regenerate your model as required, and you can preserve the changes that you have made by modifying the `@generated` marker.

Regeneration and Merging, continued

Only declarations with the `@generated` marker will be regenerated. If you leave out the `@generated` annotation, or specify `@generated NOT`, a pattern that is encouraged when modifying generated methods directly, then your changes will not be overwritten.

A useful alternative is to redirect a generated method by adding a Gen suffix to the method name, leaving the `@generated` annotation intact. You can then implement your own version of the method with the original signature and can call the "Gen" version, which will continue to be regenerated in the future.

You can use `@generated NOT` on a class or interface to disable merging for the entire class.

Within a Javadoc comment, you can include handwritten content between the `<!--begin-user-doc-->` and `<!--end-user-doc-->` tags, i.e., within the user documentation section. This content will be merged into the user documentation section of the generated comment during merging.

THE ECORE MODEL

During development, the Ecore model is the primary source of information for the EMF generator, which produces the code that we use to manipulate instances of the model. It's even possible to create instances of an Ecore model, before generating code: from the context menu of any class, invoke *Create Dynamic Instance*. Think of an Ecore model as the intersection where Java, XML Schema, and UML overlap augmented with some of the most powerful features specific to each. Ecore is closely aligned

The Ecore Model, continued

with the Object Modeling Group's Essential Meta Object Facility (EMOF), so EMF is able to read and write Ecore instances in a format that conforms to the standard XML Metadata Interchange (XMI) serialization of EMOF.

As a metamodel, Ecore is of course a simple model for describing models and includes support for:

- Classification of objects
- Attributes of those objects

The Ecore Model, continued

- Relationships or associations between those objects
- Operations on those objects
- Simple constraints on those objects, and their attributes and relationships

As a graphic demonstration of its simplicity, the bulk of the Ecore model can be represented as a UML-like diagram that fits a single page. The Ecore Tools project was used to render this diagram!

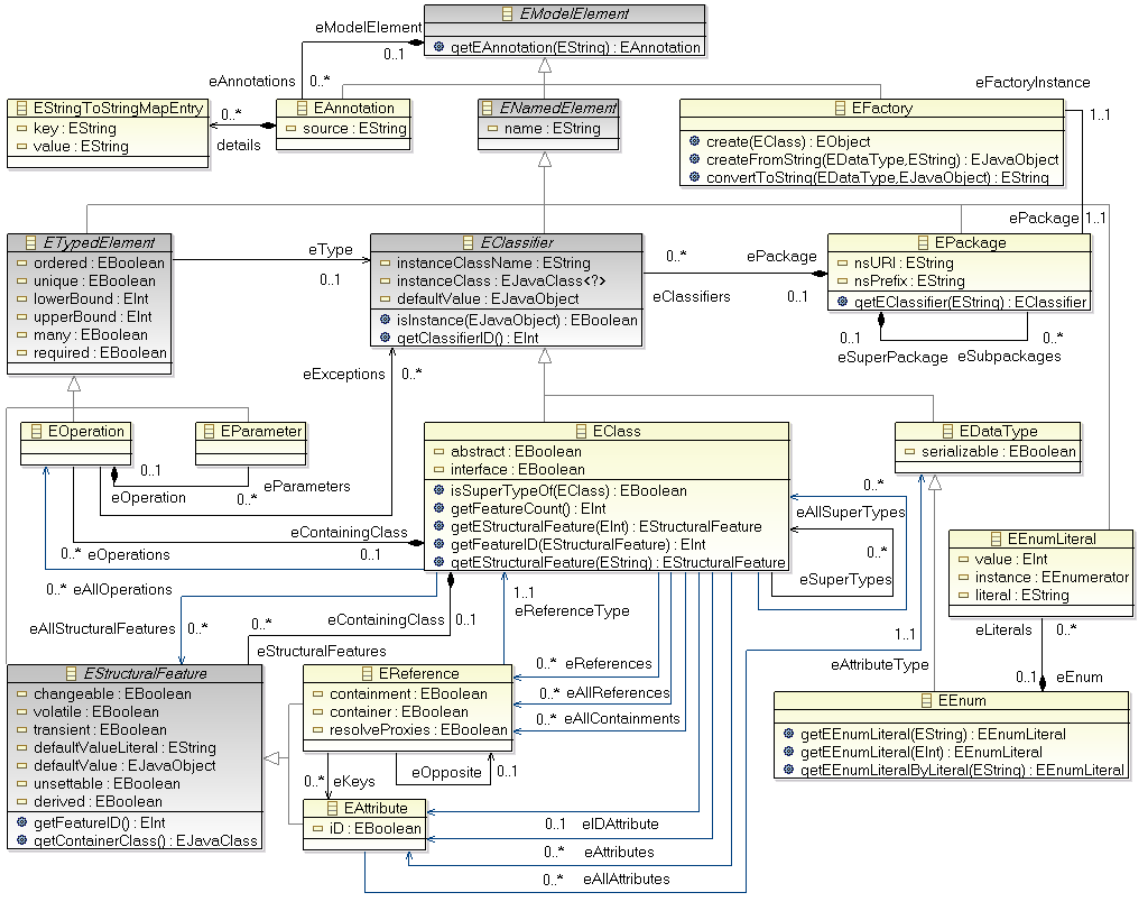


Figure 1: Putting it all together – a definitive view of the relations, attributes and operations of Ecore model elements

Class	Description
`EClass`	Models classes, which are the nodes of an object graph. Classes are identified by name and can contain a number of features, i.e. attributes and references. To support inheritance, a class can refer to a number of other classes as its super-types. A class can be abstract, in which case an instance can't be created, and can even be just an interface, in which case an implementation class is not generated.
`EAttribute`	Models attributes, which are the leaf components of an object's data. They are identified by name and they have a type. Lower and upper bounds are specified in the attribute for multiplicity.
`EDataType`	Models simple types whose structure is not modeled. Instead they act as a wrapper that denotes a primitive or object type fully defined in Java. They are identified by name and are most commonly used as attribute types.
`EReference`	Models one end of an association between two classes. They are identified by name and type, where that type represents the class at the other end of the association. Bidirectionality is supported by pairing a reference with its opposite, i.e., a reference in the class representing the other end of the association. Lower and upper bounds are specified in the reference for multiplicity. A reference can support a stronger type of association called *containment*. For a multi-valued reference, a subset of attributes of the referenced class can be identified as the key, i.e., as uniquely identifying an instance among the references.

`EModelElement`	Models the elements of an Ecore model. It's simply the abstract root of Ecore's hierarchy and supports annotations.
`EPackage`	Models packages, containers for classifiers, i.e. classes and data types. A package's name need not be unique; its namespace URI is used to uniquely identify it. This URI is used in the serialization of instance documents, along with the namespace prefix, to identify the package for the instance.
`EFactory`	Models factories for creating instance objects. The factory provides creation operations to instantiate classes and to convert data values to and from strings. Specialized mappings between data type values and their serialized form are specified by changing the implementation of the correspondingly named `createFromString()` and `convertToString()` methods in the generated factory implementation class.
`EAnnotation`	Models annotations for associating additional information with any model element. The source of the annotation is generally a URI to identify the intended meaning of the additional information, and there is support for details, i.e., a mapping of string key/value pairs.
`EClassifier`	Models the types of values in the object graph. It's the common base class of data type and class that serves as the type of any typed element, which in turn is the common base type for attributes, references, operations and parameters.
`ENamedElement`	Models elements that are named. Most elements in the Ecore model are identified by name and hence extend this class.

The Ecore Model, continued

`ETypedElement`	Models elements that are typed, e.g., attributes, references, parameters, and operations. All typed elements have an associated multiplicity specified by their `lowerBound` and `upperBound`. The unbounded `upperBound` is specified by -1, or the symbolic constant `ETypedElement.UNBOUNDED_MULTIPLICITY`.
`EStructural-Feature`	Models the value-carrying features of a class. It is the common base class for attribute and reference. The following Boolean attributes are used here to characterize attributes and references. ▪ **Changed** whether the value of the feature can be modified. ▪ **Derived** whether the value of the feature is to be computed from those of other related features. ▪ **Transient** whether the value of the feature is omitted from the object's persistent serialization. ▪ **Unsettable** whether the value of the feature has an unset state distinguishable from the state of being set to any specific value. ▪ **Volatile** whether the feature has no storage field generated in the implementation class.
`EOperation`	Models the operations that can be invoked in a given class. An operation is identified by a name and a list of zero or more typed parameters representing the overall signature. Like all typed elements, an operation specifies a type, which represents the return type; it may be null to represent no return type. An operation may also specify zero or more exceptions specified as classifiers which represent the types of exceptions that may be thrown.
`EParameter`	Models an operation's input parameters. A parameter is identified by name, and like all typed elements, specifies a type representing the type of a value that may be passed as an argument corresponding to that parameter.
`EEnumLiteral`	Models the members of enumeration type's set of literal values. An enumeration literal is identified by name and has an associated integer value as well as literal value used during serialization, which if null, defaults to the name.
`EEnum`	Models enumeration types, which specify enumerated sets of literal values.

GENERICS IN ECORE

To support the style of generics introduced by Java 5.0, Ecore was augmented in an analogous way. Parameterized types and operations can be specified, and types with arguments can be used in places where previously only regular types could be used. The same notion of Java 5.0 erasures applies, i.e., the older Ecore API can be regarded as what's left over when all use of generics has been erased. The changes are described by a single simple diagram to augment the basic API.

`EClassifiers` can specify an instance type name, where the classic instance class name is the Java erasure.	Example: `'java.util.List<java.lang.Integer>' versus 'java.util.List'`
`EClassifiers` and `EOperations` can specify `ETypeParameters`, where each type parameter is an `ENamedElement` that can optionally specify a sequence of `EGenericTypes` representing the bounds.	Example: the T in `'interface X<T extends Type> {}'.`
Each reference to `EClassifier` in the classic API is augmented by a corresponding containment reference to `EGeneric`Type.	Example: `ETypedElement.eType, EOperation.eExceptions, EClass.eSuperTypes, and EClass.eAllSuperTypes.`
An `EGenericType` represents an explicit reference to either an `EClassifier` or an `ETypeParameter` (but not both) where the raw type, or erasure, of the generic type is a derived reference corresponding to the specified classifier in classic API.	Example: it's either the classifier itself, or the first bound of the type parameter.

When a generic type references a classifier with type parameters, it's generally expected to specify type arguments in one-to-one correspondence to those type parameters. These type arguments recursively are also generic types. In this context, the generic type can have all its references unspecified to represent a wildcard, i.e., *'?', and* an upper or lower bound (but not both) can be specified to represent *'? extends Type'* or *'? super Type'*, respectively. In this way, anything expressible in Java, can be directly expressed in Ecore.

Generics in Ecore, continued

Hot Tip Ensure that 'Sample Ecore Editor/Show Generics' is enabled

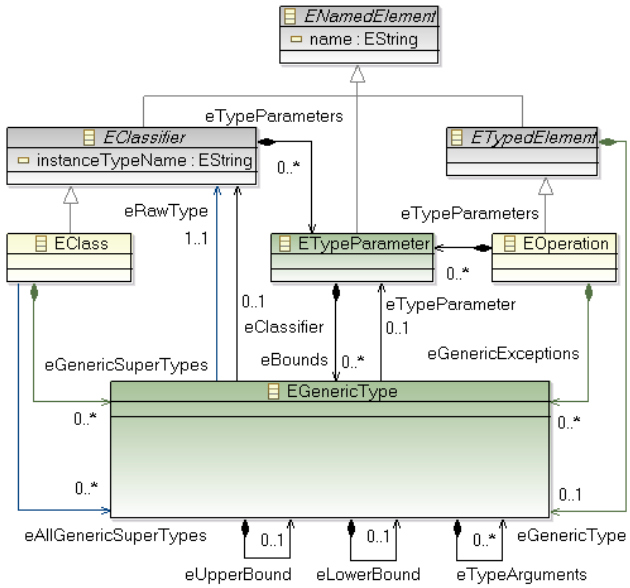


Figure 2: Extending Ecore to support generics

STRUCTURAL FEATURE CONTROL FLAGS

There are a number of flags that can be set on a model feature to control the generated code pattern for that feature as well as to direct its dynamic behavior. Typically, the default settings of these flags (shown in bold below) will be appropriate, so you shouldn't need to change them very often.

Flag	Value	Use

Unsettable	true | **false**	A feature that is declared to be unsettable has a notion of an explicit unset or no-value state. For example, a boolean attribute that is not unsettable can take on one of two values: true or false. If, instead, the attribute is declared to be unsettable, it can then have any of three values: true, false, or unset.
		The get accessor for a feature that is not set will return its default value, but for an unsettable feature, there is a distinction between this state and when the feature has been explicitly set to the default value. Since the unset state is outside of the set of allowed values, we need to generate additional methods to change a feature to the unset state and to determine if it is in that state. For example, if the label attribute in class Node is declared to be unsettable, then we'll get two more generated accessor methods, e.g.,
		`boolean isSetLabel();`
		`void unsetLabel();`
		in addition to the original two, e.g.,
		`String getLabel();`
		`void setLabel(String value);`
		The isSet method returns true if the feature has been explicitly set. The unset method changes an attribute that has been set back to its unset state.
		When unsettable is false, we don't get the generated isSet or unset methods, but we still get implementations of the reflective versions: `eIsSet()` and `eUnset()` which every `EObject` must implement. For non-unsettable attributes, `eIsSet()` returns true if the current value is different from the default value, and `eUnset()` sets the feature to the default value, more like a reset.

Structural Feature Control Flags, continued

Containment	true| **false**	Containment applies only for references. A containment reference always has an implicit opposite, even if there is no explicit opposite and that opposite is effectively a view on `EObject.eContainer()`. Whenever an object is added to a containment reference, it will be removed from any other containment reference currently holding it. That's because an object can only have one container, so adding it to a new container must remove it from the old container.
ResolveProxies	**true** | false	ResolveProxies only applies to both containment or non-containment references, but the generator respects them in the latter case only if 'Containment Proxies' are set to true in the generator. ResolveProxies implies that the reference may span documents, and therefore needs proxy checking and resolution in the get accessor. You can optimize the generated get pattern for references that you know will never be used in a cross document scenario by setting resolveProxies to false. In that case, the generated get method will be optimally efficient.
Unique	**true** | false	Unique only applies to multiplicity-many attributes, indicating that such an attribute may not contain multiple equal objects. References are always treated as unique.
Changeable	true | **false**	A feature that is not changeable will not include a generated set method, and the reflective `eSet()` method will throw an exception if you try to set it. Declaring one end of a bi-directional relationship to be not changeable is a good way to force clients to always set the reference from the other end, but still provide convenient navigation methods from either end. Declaring one-way references or attributes to be not changeable usually implies that the feature will be set or changed by some other (user-written) code.
Volatile	true | **false**	A feature that is declared volatile is generated without storage fields and with empty implementation method bodies, which you are required to complete. Volatile is commonly used for a feature whose value is derived from some other feature, or for a feature that is to be implemented by hand using a different storage and implementation pattern.
Derived	true | **false**	The value of a derived feature is computed from other features, so it doesn't represent any additional object state. Framework classes, such as `EcoreUtil.Copier`, that copy model objects will not attempt to copy such features. The generated code is unaffected by the value of the derived flag. Derived features are typically also marked volatile and transient.
Transient	true | **false**	Transient features are used to declare (modeled) data whose lifetime never spans application invocations and therefore doesn't need to be persisted. The (default XMI) serializer will not save features that are declared to be transient.
ID	true | **false**	ID applies only for attributes. The value of the ID must uniquely identify the object in its containing document and hence the ID will be used to reference the object in the (default XMI) serializer.

EOBJECT REFLECTION, INTROSPECTION, AND NOTIFICATION

All EMF modeled objects, i.e., instances of `EClasses` are `EObjects` and support reflection, i.e., you can ask an object for its class, from that class determine all the features, and then use eGet, eSet, eIsSet, and eUnset to introspect their values. An object knows its container and even the containment reference of that container which holds it. You can easily walk the contents, the children referenced by containment, and the cross references, the objects referenced by non-containment references. It's even possible to visit all the contents, i.e., the entire containment tree, via a tree iterator.

EObject
`eClass():EClass`
`elsProxy():EBoolean`
`eResource():EResource`
`eContainer():EObject`

EObject Reflection, Introspection, and Notification, continued

`eContainmentFeature():EReference`
`eContents():EEList`
`eAllContents():ETreeIterator`
`eCrossReference():EEList`
`eGet(EStructuralFeature):EJavaObject`
`eGet(EStructuralFeature,EBoolean): EJavaObject`
`eSet(EStructuralFeature,EJavaObject)`
`eIsSet(EStructuralFeature):EBoolean`
`eUnset(EStructuralFeature)`

Figure 3: Reflective operations supported by all EObjects

An EObject is a Notifier, so you can listen to any changes made to an object as follows:

```
eObject.eAdapters().add(
  new AdapterImpl() {
     @Override
     public void notifyChanged(Notification notification) {
       // Listen for changes to features.
     }});
```

MANIPULATING/PERSISTING EMF INSTANCES

EMF's persistence framework is based on the principles of Representation State Transfer (REST). Objects are stored in resources that are identified by Uniform Resource Identifiers (URIs). A URI typically consists of /-separated components of the form `[scheme:][//authority][/path][?query][#fragment]', e.g.,

```
http://www.eclipse.org/modeling/emf/?project=emf#related
file://c:/workspace/project/file.extension#id
platform:/resource/project/file.extension#id
```

An absolute URI starts with a scheme; it's recommended to **always** use absolute URIs to identify resources. Relative URIs are useful within resources for referring to other resources collocated in the same authority; this supports easier relocation of groups of related resources. For example:

```
#id
../directory/file.extension
file.extension
```

Deresolving is the process of converting an absolute URI against a base absolute URI to yield the equivalent absolute URI e.g., deresolving `platform:/resource/a/foo.html against platform:/resource/b/bar.html yields../a/foo.html`. This is used when serializing a resource to produce relative URIs in the serialized result. When a resource is deserialized, resolving a relative URI against a base absolute URI of the referencing document yields the absolute URI relative to that base, e.g., resolving `../a/foo.html` against `platform:/resource/b/bar.html` yields `platform:/resource/a/foo.html`. By consistently assigning absolute URIs to resources, you'll ensure that relative URIs will be used in their serialization whenever possible, and that will help make related collections of resources more portable.

Here is the prototypical pattern for creating a model instance and serializing it.

```
// Create a resource set to hold the resources.
ResourceSet resourceSet = new ResourceSetImpl();
```

Manipulating/Persisting EMF Instances, continued

```
// Create a new empty resource.
Resource resource =
  resourceSet.createResource
    (URI.createFileURI("c:/My.tree"));

// Create and populate instances.
Node rootNode = TreeFactory.eINSTANCE.createNode();
rootNode.setLabel("root");
Node childNode = TreeFactory.eINSTANCE.createNode();
childNode.setLabel("child");
rootNode.getChildren().add(childNode);

// Add the root object to a resource and save it.
resource.getContents().add(rootNode);
resource.save(null);
```

Note that when running stand-alone, rather than when running as an Eclipse application, it's important to keep in mind that Eclipse's convenient plugin extension registrations will not be available and hence explicit registrations will be required, e.g., registration of the package and the resource factory will be needed, if it's not handled elsewhere already.

```
// Register the appropriate resource factory
// to handle all file extensions.
resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().
  put
    (Resource.Factory.Registry.DEFAULT_EXTENSION,
     new XMIResourceFactoryImpl());

// Register the package to make it available during loading.
resourceSet.getPackageRegistry().put
  (TreePackage.eNS_URI,
   TreePackage.eINSTANCE);
```

Here's the prototypical pattern to load an instance.

```
// Demand load the resource into the resource set.
Resource resource =
  resourceSet.getResource
    (URI.createFileURI("c:/My.tree"), true);

// Extract the root object from the resource.
Node rootNode = (Node)resource.getContents().get(0);
```

Once the resource is loaded, the instance can be processed using the generated API. Note that a ready-to-run example just like the above is generated when you invoke Generate Test Code. It's a good place to write code to start exploring your generated API and the capabilities of EMF.

Hot Tip A wealth of utilities are available in `EcoreUtil`, so be sure to investigate that.

EMF MODEL USING ANNOTATED JAVA

Many Java developers prefer to define their initial model through Java interfaces. Leveraging the power of Javadoc annotations, you can switch between your Ecore model and your Java code seamlessly. Your code should be annotated with the `@model` annotation as follows:

```
/**
 * @model
 */
public interface Node {
  /**
   * @model opposite="parent" containment="true"
   */
  List<Node> getChildren();

  /**
   * @model opposite="children"
   */
  Node getParent();
}
```

EMF Model Using Annotated Java, continued

@model Properties for Classes

The Java specification for an **EClass** is a Java interface preceded by an `@model` tag.

Property	Value	Usage
abstract	true | **false**	The abstract attribute of the EClass is set to the specified value.
interface	true | **false**	The interface attribute of the EClass is set to the specified value.

@model Properties for Typed Elements

An **ETypedElement** is specified as a method in the interface corresponding to the **EClass** that contains the typed element.

Property	Value	Usage
lower or lowerBound	Integer-value	The lowerBound attribute of the `ETypedElement` is set to the Integer-value, which must be 0 or greater.
many	true | **false**	If true, the upperBound attribute of the `ETypedElement` is set to -1 (unbounded). Otherwise, it is set to 1.
ordered	true | **false**	The ordered attribute of the `ETypedElement` is set to the specified value.
required	true | **false**	If true, the lowerBound attribute of the `ETypedElement` is set to 1. Otherwise, it is set to 0.
type	Type-name	The eType reference of the `ETypedElement` is set to an `EClassifier` corresponding to the Java *type-name*.
unique	**true** | false	The unique attribute of the `ETypedElement` is set to the specified value.
upper or upperBound	Integer-value	The upperBound attribute of the `ETypedElement` is set to Integer-value. The specified value must be greater than 0, or -1 (unbounded).

@model Properties for Structural Features

An **EStructuralFeature** is specified as an accessor method in the interface corresponding to the **EClass** that contains the feature. The properties for `ETypedElement` also apply.

Property	Value	Usage
changeable	true | **false**	The changeable attribute of the `EStructuralFeature` is set to the specified value.
derived	true | **false**	The derived attribute of the `EStructuralFeature` is set to the specified value.
suppressed GetVisibility	true | **false**	If true, the `EStructuralFeature` is annotated with a GenModel-sourced `EAnnotation` that suppresses the `get()` accessor for the feature in the interface.
suppressed IsSetVisibility	true | **false**	If true, the `EStructuralFeature` is annotated with a GenModel-sourced `EAnnotation` that suppresses the `isSet()` accessor for the feature in the interface.
suppressed SetVisibility	true | **false**	If true, the `EStructuralFeature` is annotated with a GenModel-sourced `EAnnotation` that suppresses the `set()` accessor for the feature in the interface.
suppressed UnsetVisibility	true | **false**	If true, the `EStructuralFeature` is annotated with a GenModel-sourced `EAnnotation` that suppresses the `unset()` accessor for the feature in the interface.
transient	true | **false**	The transient attribute of the `EStructuralFeature` is set to the specified value.
unsettable	true | **false**	The unsettable attribute of the `EStructuralFeature` is set to the specified value.
volatile	true | **false**	The volatile attribute of the `EStructuralFeature` is set to the specified value.

@model Properties for Attributes

An **EAttribute** is specified as an accessor method in the interface corresponding to the **EClass** that contains the attribute. The properties for `EStructuralFeature`, and hence, `ETypedElement` also apply.

Property	Value	Usage
dataType	Data-type	The specific `EDataType` named data-type is used as the eType for the `EAttribute`. If not already modeled, an `EDataType` is created with the given name.
default or defaultValue	Default-value	The `defaultValueLiteral` attribute of the `EAttribute` is set to the string value identified by *default-value*.
id	true | **false**	The id attribute of the `EAttribute` is set to the specified value.

EMF Model Using Annotated Java, continued

@model Properties for References

An **EReference** is specified as a method in the interface corresponding to the **EClass** that contains the reference. The properties for `EStructuralFeature`, and hence, `ETypedElement` also apply.

Property	Value	Usage
containment	true | **false**	The containment attribute of the `EReference` is set to the specified value.
opposite	Reference name	The opposite reference of the `EReference` is set to the `EReference` corresponding to the specified reference-name. The opposite `EReference` must belong to the `EClass` that is identified by the eType of this `EReference`.
resolveProxies	true | **false**	The resolveProxies attribute of the `EReference` is set to the specified value. The default value is false when containment is true, and true otherwise.
type	Type-name	The `eType` reference of the `EReference` is set to an `EClass` corresponding to the Java *type-name*.

@model Properties for Operations

An **EOperation** is specified as a method in the Java interface corresponding to the **EClass** that contains the operation. The properties for `ETypedElement` also apply.

Property	Value	Usage
dataType	Data-type	The specific `EDataType` named data-type is used as the eType for the `EOperation`. If not already modeled, an `EDataType` is created with the given name.
exceptions	List-of-types	The list-of-types is a space-separated list of names, each specifying the `EDataType` to be used for the corresponding `eException`. If not already modeled each `EDataType` is created with the given name. To avoid specifying a particular `EDataType` for the corresponding exception, a "-" character can appear as an item in the list.
type	Type-name	The eType reference of the `EOperation` is set to an `EClassifier` corresponding to the Java type-name.

EMF MODEL USING XML SCHEMA

An advantage to using XML schema to define your model is that, when serialized, instances of the model will conform to your XML schema. At a high level, the mapping to Ecore is quite simple:

A schema maps to an `EPackage`
A complex type definition maps to an `EClass`
A simple type definition maps to an `EDataType`
An attribute declaration or element declaration maps to an `EAttribute` if its type maps to an `EDataType`, or to an `EReference` if its type maps to an `EClass`. There is is a special `EClass` called the `DocumentRoot`, to hold global elements and attributes, i.e., those not nested in a complex type.

EMF Extensions to XML Schema

Ecore namespace attributes can be used to tailor the default mapping onto Ecore as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    xmlns:tree="http://www.example.com/tree"
    ecore:nsPrefix="tree"
    ecore:package="com.example.tree"
    targetNamespace="http://www.example.com/tree">
  <xsd:element name="tree" type="tree:Node"/>
  <xsd:complexType name="Node">
    <xsd:sequence>
      <xsd:element ecore:opposite="parent"
          maxOccurs="unbounded" minOccurs="0" name="children"
          type="tree:Node"/>
    </xsd:sequence>
    <xsd:attribute name="label" type="xsd:string"/>
  </xsd:complexType>
</xsd:schema>
```

EMF Model Using XML Schema, continued

Property	Applicability	Value	Usage
`ecore: changable`	`<xsd:element>` `<xsd:attribute>`	true | **false**	The changeable attribute of the `EStructuralFeature` is set to the specified value.
`ecore: constraints`	`<xsd:complexType>` `<xsd:simpleType>`	List-of-Names	The EStructuralFeature is annotated with an Ecore-sourced `EAnnotation` that contains a key with the value 'constraint' whose value is the space-separated list of named constraints.
`ecore: default`	`<xsd:element>` `<xsd:attribute>`	Literal-value	The derived value literal attribute of the `EAttribute` is set to the specified value.
`ecore: derived`	`<xsd:element>` `<xsd:attribute>`	true | **false**	The derived attribute of the `EStructuralFeature` is set to the specified value.
`ecore: documentRoot`	`<xsd:schema>`	Name	The document root `EClass`'s name is set to the specified value.
`ecore: enum`	`<xsd:simpleType>`	**true** | false	If false, rather than mapping the simple type with enumeration facets to an `EEnum`, it maps to a `EDataType` with enumeration extended meta data annotations.
`ecore: featureMap`	`<xsd:element>` `<xsd:sequence>` `<xsd:schoice>` `<xsd:all>`	Name	If the empty string is specified, it disables the mapping to a feature map. Otherwise, it specifies the name of the feature map `EStructuralFeature`.
`ecore: ignore`	`<xsd:attribute>` `<xsd:element>` `<xsd:annotation>` `XSD facets`	**false** | true	When true is used on gobal elements or attributes, annotations or their documentation or appinfo contents, or on simple type facets, it specifies that the corresponding `Ecore` construct should not be produced as normal.
`ecore: implements`	`<xsd:complexType>`	List-of-QName	Specifies additional super types, as denoted by the `QNames`, for the `EClass`.
`ecore: instance Class`	`<xsd:complexType>` `<xsd:simpleType>`	Java-Class	Specifies the instance type name of the `EClassifier`; '{}' can be used in place of '<>'.
`ecore: interface`	`<xsd:complexType>`	true | **false**	The interface attribute of the `EClass` is set to the specified value.
`ecore: key`	`<xsd:appinfo>`	String-value	Specifies the key of the details entry in the `EAnnotation` where the appinfo's contents are the value.
`ecore: lowerBound`	`<xsd:attribute>` `<xsd:element>`	Integer-value	Specifies the value of the lowerBound attribute of the `EStructuralFeature`.
`ecore: many`	`<xsd:attribute>`	true | **false**	When true is used on an attribute with a list-type value, it will instead map to a multiplicity many feature of the corresponding item type.
`ecore: mixed`	`<xsd:complexType>`	true | **false**	The complex type is mapped exactly as if it had a real mixed attribute with the specified value.
`ecore: name`	`XSD named components`	Name	Specifies the name of the `ENamedElement`.
`ecore: nsPrefix`	`<xsd:schema>`	NCName	Specifies the nsPrefix of the `EPackage`.
`ecore: opposite`	`<xsd:attribute>` `<xsd:element>`	Name	Specifies by name an element or attribute in the `EReference`'s type that will pair with the reference as its opposite. When used on a containment reference, a container reference will be created.
`ecore: ordered`	`<xsd:attribute>` `<xsd:element>`	**true** | false	Sets the value of the ordered attribute on the `EStructuralFeature`.
`ecore: package`	`<xsd:schema>`	Name	Specifies the fully qualified Java package name, the last segment of which will be used as the name of the `EPackage`.
`ecore: reference`	`<xsd:attribute>` `<xsd:element>`	QName	When used on an attribute or element of type `IDREF`, `IDREFS`, or anyURI, the QName specifies a complex type corresponding to the `EClass` that that is the type of the `EReference`.
`ecore: resolve Proxies`	`<xsd:attribute>` `<xsd:element>`	true | **false**	Specifies the resolveProxies attribute of the `EReference`.

EMF Model Using XML Schema, continued

ecore: serializable	<xsd:simpleType>	**true** | false	Specifies the serializable attribute of the EDataType.
ecore: suppressed Get Visibility	<xsd:attribute> <xsd:element>	true : **false**	If true, the EStructuralFeature is annotated with a GenModel-sourced EAnnotation that suppresses the get() accessor for the feature in the interface.
ecore: suppressed IsSet Visibility	<xsd:attribute> <xsd:element>	true : **false**	If true, the EStructuralFeature is annotated with a GenModel-sourced EAnnotation that suppresses the isSet() accessor for the feature in the interface.
ecore: suppressed Set Visibility	<xsd:attribute> <xsd:element>	true : **false**	If true, the EStructuralFeature is annotated with a GenModel-sourced EAnnotation that suppresses the set() accessor for the feature in the interface.

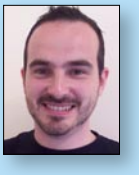
EMF Model Using XML Schema, continued

ecore: suppressed Unset Visibility	<xsd:attribute> <xsd:element>	true : **false**	If true, the EStructuralFeature is annotated with a GenModel-sourced EAnnotation that suppresses the unset() accessor for the feature in the interface.
ecore: transient	<xsd:attribute> <xsd:element>	true : **false**	Specifies the transient attribute of the EStructuralFeature.
ecore: unique	<xsd:attribute> <xsd:element>	**true** : false	Specifies the unique attribute of the EStructuralFeature.
ecore: unsettable	<xsd:attribute> <xsd:element>	true : **false**	Specifies the unsettable attribute of the EStructuralFeature.
ecore: upperBound	<xsd:attribute> <xsd:element>	Integer	Specifies the upperBound attribute of the EStructuralFeature.
ecore: value	<xsd:enumeration>	Integer	Specifies the value of the EEnumLiteral.
ecore: volatile	<xsd:attribute> <xsd:element>	true : **false**	Specifies the volatile attribute of the EStructuralFeature.

ABOUT THE AUTHOR

Ed Merks

Ed Merks leads the top-level Eclipse Modeling Project along with Rich Gronback (Borland), and the Eclipse Modeling Framework subproject. He holds a Ph.D. in Computing Science from Simon Fraser University. Ed has his own small company, Macro Modeling, and his work at Eclipse is funded by itemis AG. Ed is well recognized for his dedication to the Eclipse community, posting literally thousands of newsgroup answers each year.



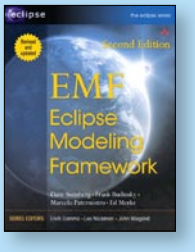
James Sugrue

James Sugrue is a software architect at Pilz Ireland, a company using many Eclipse technologies. James also works as editor on EclipseZone and JavaLobby. He follows the Eclipse projects closely and is interested in their application in many industries.

Publication

EMF: Eclipse Modeling Framework, 2nd Edition, co-author with Dave Steinberg, Frank Budinsky, and Marcelo Paternostro, Addison Wesley Publications

RECOMMENDED BOOK



The second edition contains more than 40% new material. The authors illuminate the key concepts and techniques of EMF modeling, analyze EMF's most important framework classes and generator patterns, guide you through choosing optimal designs, and introduce powerful framework customizations and programming techniques.

BUY NOW
books.dzone.com/books/emf

Get More FREE Refcardz. Visit refcardz.com now!

Upcoming Refcardz:

SOA Patterns

Windows Presentation Foundation

HTML and XHTML

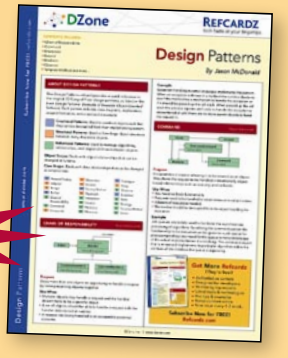
SOA Governance

Agile Methodologies

Available:

Essential EMF
Getting Started with Equinox & OSGi
Core Mule
Core CSS: Part III
Using XML in Java
Essential JSP Expression Language
Getting Started with Hibernate Search
Core Seam
Essential Ruby
Essential MySQL
JUnit and EasyMock
Spring Annotations
Getting Started with MyEclipse
Core Java
Core CSS: Part II
PHP
Getting Started with JPA
JavaServer Faces
Core CSS: Part I
Struts2

FREE



Design Patterns
Published June 2008

Visit refcardz.com for a complete listing of available Refcardz.

DZone

DZone communities deliver over 4 million pages each month to more than 1.7 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. "DZone is a developer's dream," says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-39-4
ISBN-10: 1-934238-39-2
50795
9 781934 238394
$7.95