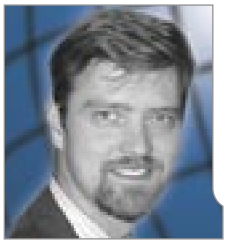
A high-level overview of its anatomy and artifacts

Eclipse Modeling Framework

BY PETTER GRAFF



Petter Graff is vice president of InferData, Ltd. He has more than 20 years of experience building object-oriented solutions. The last 10 years he has been teaching and consulting on enterprise architectures for Fortune 500 companies worldwide. pgraff@inferdata.com

The Eclipse Modeling Framework (EMF) is an open source code generation tool distributed under the Eclipse umbrella. It is a tool created in the spirit of the OMG's Model Driven Architecture (MDA) and an excellent example of the power of MDA.

EMF is capable of creating sophisticated editors from abstract business models. These editors are implemented as plugins for Eclipse. EMF creates feature complete implementations including persistence, business model implementation, editing framework and editors.

(*Note:* WebSphere Studio supports Eclipse plugins. The Eclipse IDE may also be used to directly develop in WebSphere.)

In this article, we'll look at the anatomy of EMF and what it produces by building a simple Eclipse editor for keeping track of music collections.

EMF and MDA

The process of using EMF is compatible with the MDA approach of OMG, however it is missing some of the essential properties of an MDA tool.

- ***EMF is not a general-purpose code-generation tool.*** It generates Eclipse plugins. It would be difficult to make EMF generate, let's say, a .NET application.
- ***EMF generates code from models.*** Strictly speaking, an MDA tool should generate platform-specific models (PSM) before generating code.

Even though it may not line up with OMG 100%, EMF is one of the most powerful ambassadors for the MDA approach.

Process of Using EMF

In Figure 1 we have outlined the typical process used when developing plugins with EMF.

First, we need to create a business model (OMG's PIM). In EMF, this model is called the *ecore* model. The ecore model contains structural requirements for the implementation of the editor.

Second, we need to configure the code-generation options. These options are stored in a model called the *genmodel*. The genmodel decorates the ecore model with information specific to the solution domain (in our case Java and Eclipse). The third step is to generate the code. This step uses the ecore model and the genmodel to generate the implementation for the editor plugins.

The generated code can now be tested. It is feature complete, but a rather naive implementation of an editor. In most cases, a developer has to take over the implementation of sections of the generated code. EMF has a feature that allows the developer to flag the code he/she wants to enhance.

After the code has been generated, a developer typically starts an iterative process where various discrepancies are fixed. The discrepancies can be separated into three categories:

- ***Errors in the business model:*** Requires update to the ecore model
- ***Suboptimal configuration of the code generation options:*** Requires changes to the genmodel
- ***Requirements for a more sophisticated implementation than that generated:*** Requires that the developer takes control of sections of the generated code and modifies the code.

EMF has excellent support for this iterative development process, allowing the developers to fix and regenerate.

Creating the Models

Let's illustrate the various stages and artifacts in EMF by creating a simple application to keep track of a music collection.

We first need to create a business model for the information content the editor will manipulate. In EMF, the business requirements are captured in ecore models. The ecore model is really just an XML file with an ".ecore" extension, hence, we could manipulate the XML file directly. However, for convenience EMF lets you define the ecore model in many ways (see Figure 2):

- A built-in primitive editor (this editor is actually an EMF generated editor!)
- Import from XML Schema
- Import from Rational Rose Models
- Import specially annotated Java

Interfaces (actually, this option is a synchronized option. When the Java interfaces change, the ecore model is synchronized and vice versa).

The ecore model for the Music library contains the structural rules for the implementation. These rules can be visualized in a UML class diagram (see Figure 3).

When creating class diagrams, only a few constructs are used:

- ***Classes:*** Represents the object types that we need to persist and manipulate.
- ***Attributes:*** Attributes represents properties persisted on the objects.
- ***Associations:*** Associations represent relationship between objects. The associations have a few different dimensions:
  - *Containment:* Associations may represent containment relationships. The semantic of this in the ecore model is to specify that an object has a lifetime dependency to its container; for example, if an artist is removed, so is all of their work.
  - *Directionality:* Associations may be unidirectional or bidirectional. If an association is unidirectional, it is only possible to navigate the object model from one of the types. E.g., in our example, it is possible to retrieve the works if we know the artist, but it is not possible to find the artist from some work. An association can also be bidirectional allowing navigation in both directions.

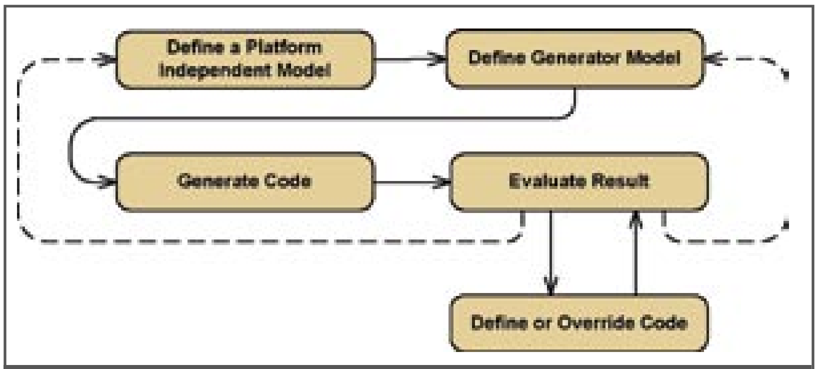


FIG 1: PROCESS OVERVIEW

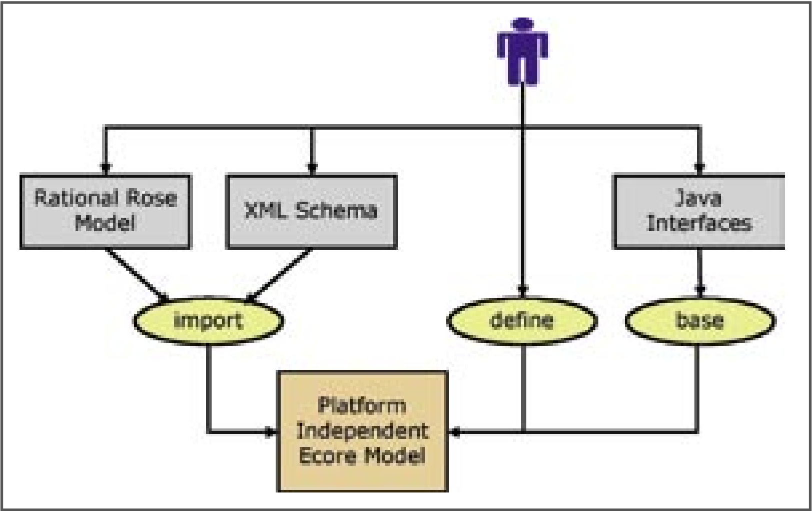


FIG 2: OPTIONS FOR CREATING ECORE MODELS

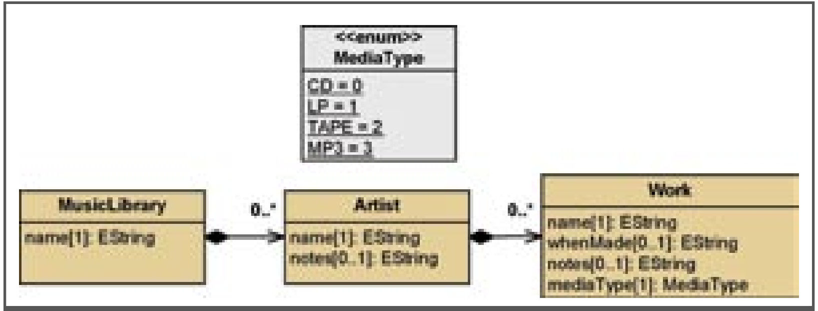


FIG 3: UML MODEL FOR THE MUSIC LIBRARY

It is possible to create the ecore model using the UML notation. There are tools on the market that directly create ecore models (e.g., Omondo-UML). Another alternative is to import Rational Rose models.

The UML diagram shown in Figure 3 defines the ecore model shown in Listing 1. The ecore model can be browsed by the build-in ecore editor shown in Figure 4.

We can now create a genmodel based on the ecore model file. The genmodel adds properties to the various constructs in the ecore file (see Figure 5).

The genmodel is also an XML file. The genmodel decorates the ecore model with a set of attributes defining specific configuration for generating Eclipse plugins. It may seem strange to have two models (i.e., why not just add the properties to the ecore model?), but this is essential from the perspective of MDA. If we wanted to create another implementation based on the same business model, we could do so without changing the ecore model.

EMF comes with a point-and-click wizard for creating genmodels from an ecore model, making this task trivial.

Generating Code

Code is generated from the generator model (see Figure 6). The code generator reads the ecore model, the generator models, and a set of code definition templates defined in a template language called JET. The

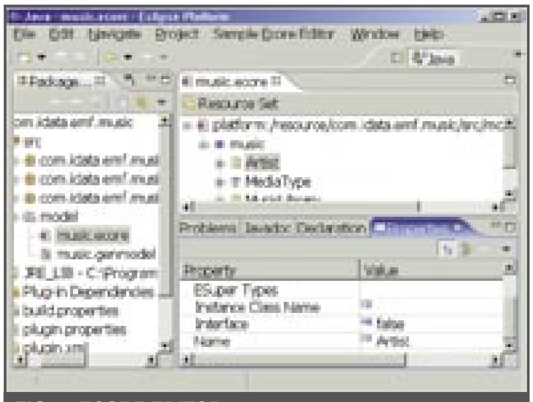

Java - music.ecore - Eclipse Platform
File Edit Navigate Project Sample Ecore Editor Window Help
Package...
music.ecore
Resource Set
music
Artist
MediaType
Problems Javadoc Declaration
Property
Value
ESuper Types
Instance Class Name
Interface
false
Name
Artist

FIG 4: ECORE EDITOR

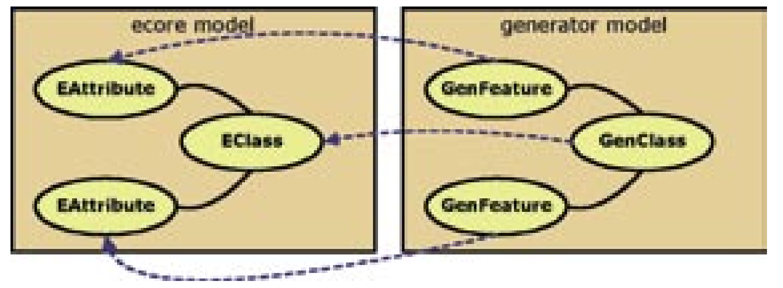


FIG 5: ECORE AND GENMODEL

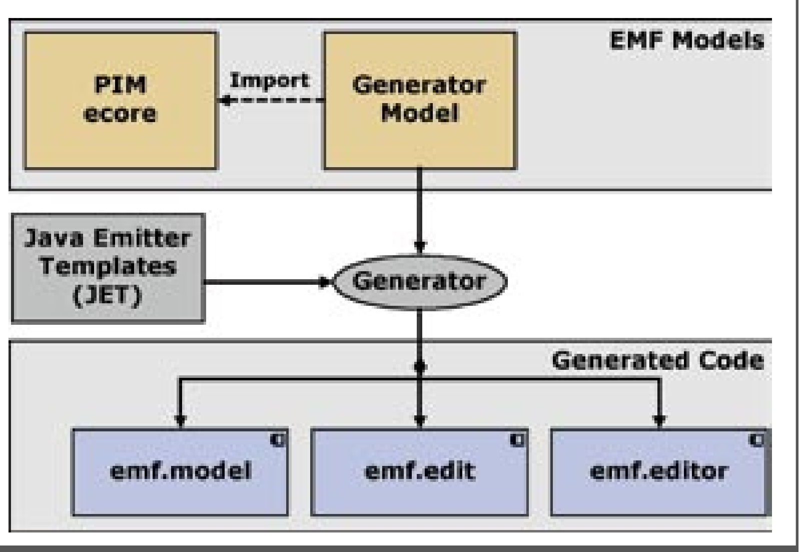


FIG 6: CODE GENERATION FRAMEWORK

JET template is not likely to be modified by most of the EMF users; however, a short description of what they are and the role they play may help to understand the EMF framework.

JET is based on the JavaServer Pages (JSP) syntax (in fact, the JET implementation was based on the Tomcat implementation). A JET template contains a template for files containing Java implementation files. The framework "expands" the various JET files to generate the Java source files.

What Is Generated?

EMF can create three plugins (see Figure 7):

- ***emf.model:*** The emf.model contains an implementation that is closely aligned with the ecore model. The emf.model plugin contains an object-oriented API that allows programmers to manipulate and persist objects based on the business model.
- ***emf.edit:*** The emf.edit plugin contains user-interface independent editor code. The implementation in this plugin contains adapters that shield the model code from the presentation code. The code also provides a sophisticated command framework with unlimited command stacks to support undo and redo.
- ***emf.editor:*** The emf.editor plugin contains the presentation code for the editor.

The three plugins make up a complete implementation of an editor, allowing users to create and maintain music libraries.

The implementation is based on best practices for developing Eclipse plugins. It uses a set of design patterns to ensure separation and decoupling of concerns. The implementation is highly efficient and its functionality supported far beyond what is typically implemented (e.g., drag-and-drop, unlimited undo/redo, etc).

The Final Result

After generating the code, we can now test or deploy our plugins. The new editor allows us to edit files with a particular extension. The extension name is determined by the code generation options. In our case, we've used the ".music" extension. The new editor recognizes files with a ".music" extension. EMF has also included a wizard for creating new music files (see Figure 8).

Figure 8 shows the running editor. It presents the object structure as a tree. The attribute and associations are modified in a property editor. This is not the only option for presentation. The genmodel allows us to set up various other preferences for how to create the model.

To play around with all the models and the generated plugins, you may download the ecore model, the genmodel and the generated plugins from www.inferdata.com/downloads/emf/emf_intro.

Uses of EMF

The most dominant use of EMF is inside IBM, where it is used for creating editors for their flagship development tool WSAD.

At InferData, we have been using EMF for the following tasks:

- Create persistence implementation for various in-house products
- Create standalone products for the Eclipse platform
- Create quick prototypes to validate complex business models

The use of EMF to create editors is obvious and its benefits are immediate. The last use, to prototype business models, may not be as obvious.

Often, in building large-scale business systems the validation of the business requirements may yield great payback. It is very expensive to find errors late in development. Building formal UML models helps avoid this problem, but there are a limited number of domain experts available that can read the UML models. It has been our experience, that a quick prototype of a business model may widen the understanding of complex business models even though the final target implementation may be on a completely different technology.

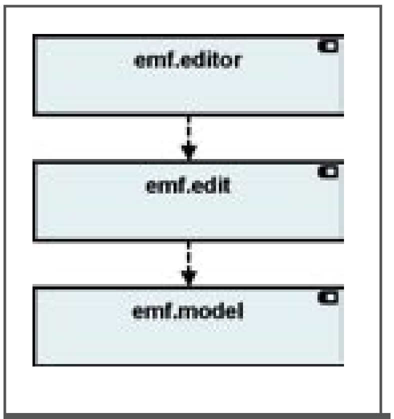


FIG 7: PLUG-IN DEPENDENCY GRAPH

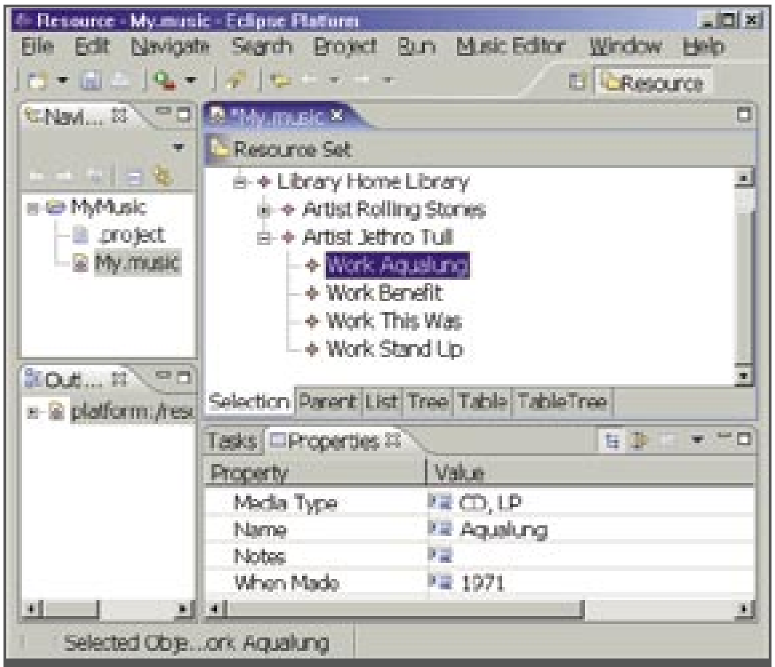
Conclusion

In this article, I've provided a high-level overview of EMF. We've looked at its anatomy and the artifacts it produces.

EMF is a powerful framework for creating Eclipse plugins. Its framework is inspired by the MDA standard emerging from OMG. Even though one may argue that EMF doesn't meet the MDA standard one hundred percent, it is highly practical and provides significant benefit to Eclipse developers.

The main benefits are:

- Significant improvement of productivity.
- High-quality implementation based on best practices for developing Eclipse plugins.
- Excellent separation of concern. Business models remain technology independent; code generation is performed for all that can be code generated and kept separate from the manually developed code.



Resource - My.music - Eclipse Platform
File Edit Navigate Search Project Run Music Editor Window Help
Resource
Navi...
My.music
Resource Set
MyMusic
.project
My.music
Out...
platform:/reso
Selection Parent List Tree Table TableTree
Tasks Properties

- Library Home Library
  - Artist Rolling Stones
  - Artist Jethro Tull
    - Work Aqualung
    - Work Benefit
    - Work This Was
    - Work Stand Up

Property	Value
Media Type	CD, LP
Name	Aqualung
Notes	
When Made	1971

Selected Obje...ork Aqualung

FIG 8: THE MUSIC EDITOR

LISTING 1

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xmlns:ecore="http://www.eclipse.org/emf/2002/
Ecore"
  name="music"
  nsURI="http://www.inferdata.com/emf/music"
  nsPrefix="music">
  <eClassifiers
        xsi:type="ecore:EClass"
        name="MusicLibrary">
        <eStructuralFeatures
                xsi:type="ecore:EAttribute"
                name="name"
                lowerBound="1"
        eType="ecore:EDataTypehttp://www.eclipse.
org/emf/2002/Ecore#//EString"/>
        <eStructuralFeatures
                xsi:type="ecore:EReference"
                name="artists" upperBound="-1"
                eType="#//Artist"
containment="true"/>
  </eClassifiers>
  <eClassifiers
        xsi:type="ecore:EClass"
        name="Artist">
        <eStructuralFeatures
                xsi:type="ecore:EAttribute"
                name="name" lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.
org/emf/2002/Ecore#//EString"/>
        <eStructuralFeatures
                xsi:type="ecore:EAttribute"
                name="notes"
        eType="ecore:EDataType http://www.eclipse.
org/emf/2002/Ecore#//EString"/>
        <eStructuralFeatures
                xsi:type="ecore:EReference"
                name="works" upperBound="-1"
                eType="#//Work"
                containment="true"/>
  </eClassifiers>
  <eClassifiers
        xsi:type="ecore:EClass"
        name="Work">
        <eStructuralFeatures
                xsi:type="ecore:EAttribute"
                name="name"
                lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.
org/emf/2002/Ecore#//EString"/>
        <eStructuralFeatures
                xsi:type="ecore:EAttribute"
                name="whenMade"
                lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.
org/emf/2002/Ecore#//EString"/>
        <eStructuralFeatures
                xsi:type="ecore:EAttribute"
                name="notes"
        eType="ecore:EDataType http://www.eclipse.
org/emf/2002/Ecore#//EString"/>
        <eStructuralFeatures
                xsi:type="ecore:EAttribute"
                name="mediaType"
                lowerBound="1"
                eType="#//MediaType"/>
  </eClassifiers>
  <eClassifiers
        xsi:type="ecore:EEnum"
        name="MediaType">
        <eLiterals name="CD"/>
        <eLiterals name="LP" value="1"/>
        <eLiterals name="TAPE" value="2"/>
        <eLiterals name="MP3" value="3"/>
  </eClassifiers>
</ecore:EPackage>
```