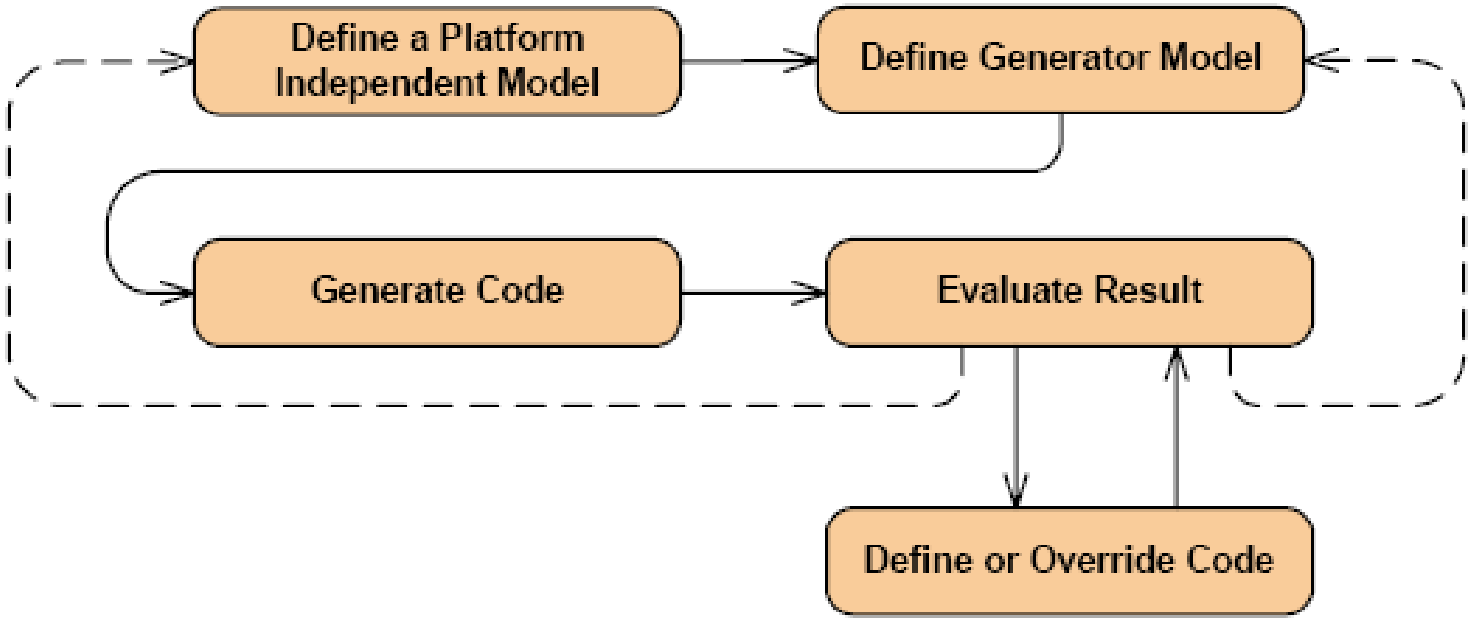
EMF

Eclipse Modeling Framewokr

What's EMF?

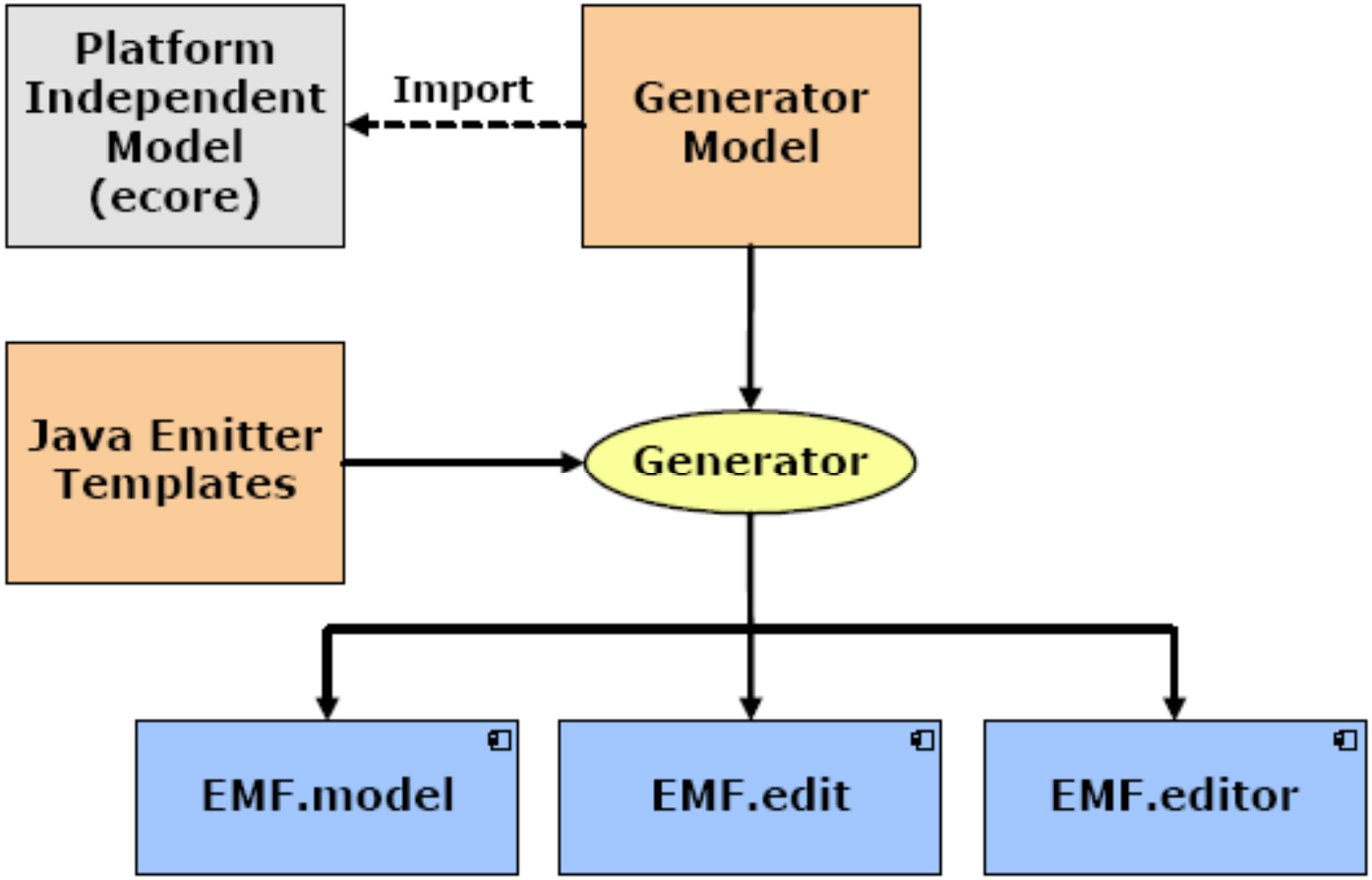
- EMF is part of the ***tools project*** for Eclipse
- The answer to "What is EMF?" depends on who you ask
- EMF is a ***modeling & data integration framework***
  - The foundation for storing metadata and metamodels
- EMF is a ***code generation framework*** for building plug-ins for Eclipse
  - Used to create Eclipse editors

How to Work with EMF



- The above diagram shows a ***simplified*** view of how to use EMF

EMF Toolset


Platform Independent Model (ecore)
Import
Generator Model
Java Emitter Templates
Generator
EMF.model
EMF.edit
EMF.editor

The *ecore* Model

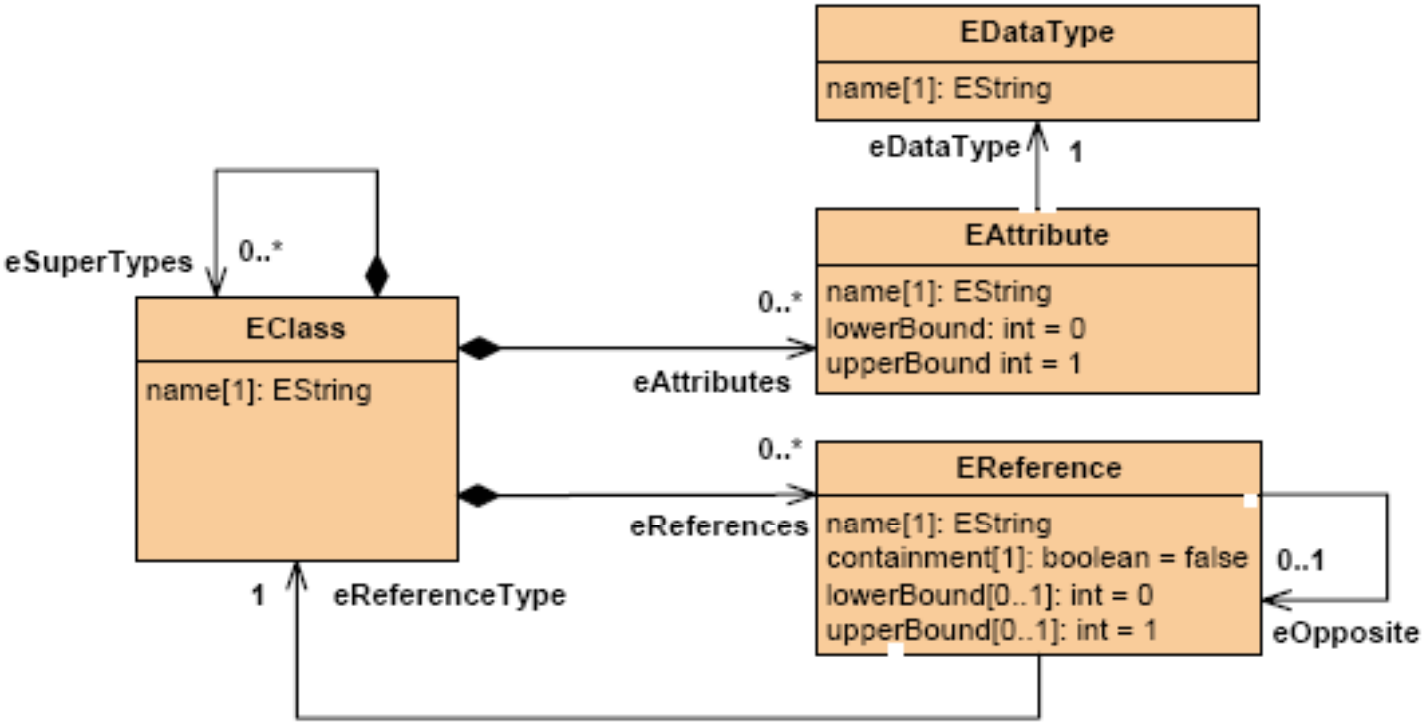
What's *ecore*?

- ecore is the EMF language for defining models
  - A ***metalanguage***
- It allows instantiation of object-oriented models
- Standards based
  - Inspired by OMG's MOF 1.4
  - Resubmitted and in line with OMG's Essential Meta Object Facility (***MOF 2.0 EMOF***)
- ecore is used to ***define the Platform Independent Models***
  - Foundation for code generation
  - Standard for modeling metadata

What's the Purpose of *ecore*?

- *ecore* allows you to define ***structural models***
- These models are often found in organizations as:
  - ***UML class diagrams***
  - ***XML Schema Definitions***
  - Entity Relationship Diagrams
- Why one more essential modeling structure?
  - Ecore is focusing only on the essential information
  - EMF provides tools that support
    - ***Code generation***
    - Import/export to/from various other forms
    - It has ***IBM*** support

Some Key *ecore* Types

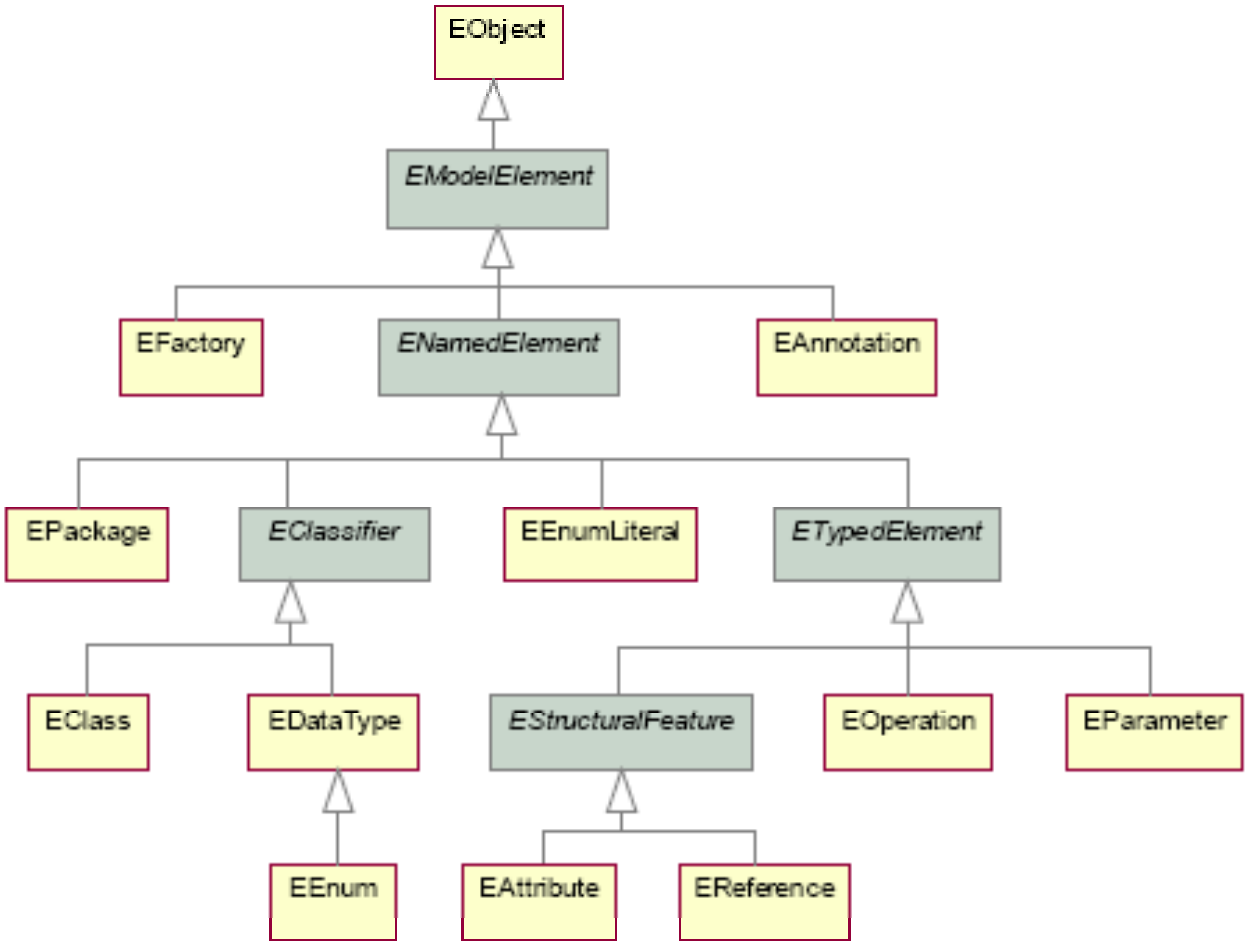


- The simplified ***metamodel*** above represents the minimum set you must understand to come to terms with ***ecore***

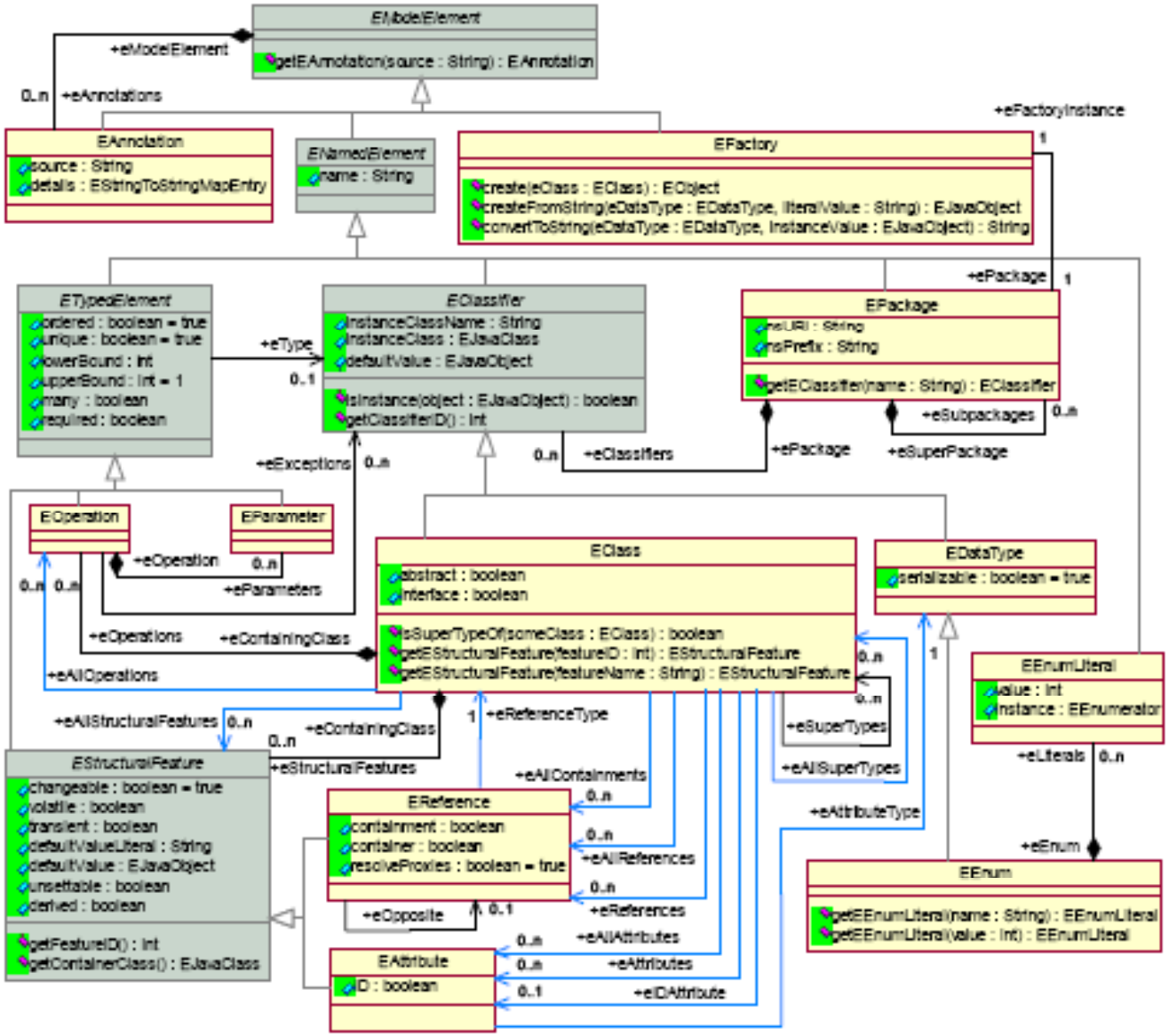
Key Concepts in *ecore*

- ***EClass***
  - Represents a type
  - A type may define:
    - Any number of supertypes
    - Any number of references (aka, associations)
    - Any number of attributes
- ***EAttribute***
  - Represents a typed attribute
- ***EReference***
  - Represents an association end
  - Optionally points to the opposite association end
  - Defines the referenced type

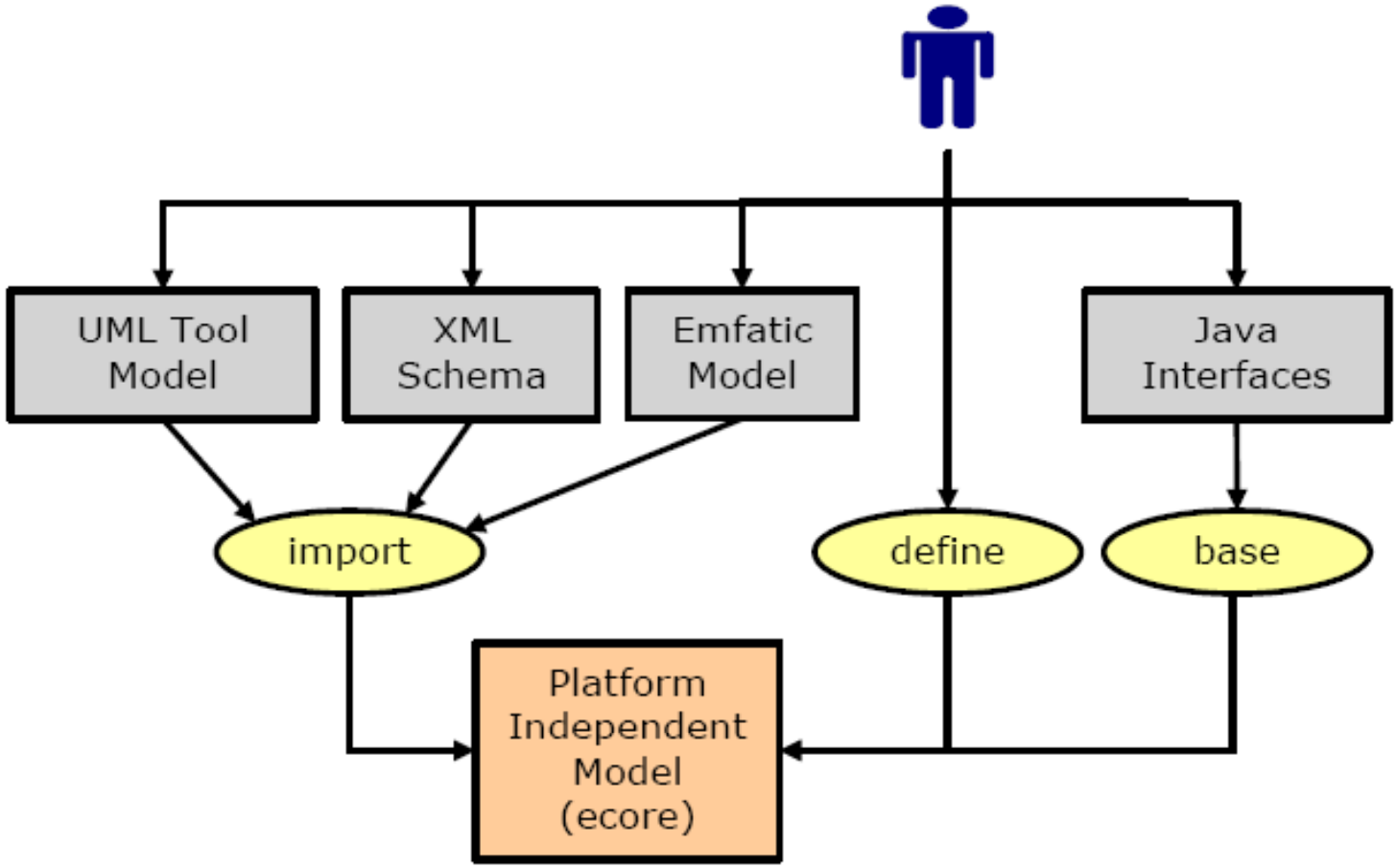
ecore Hierarchy



ecore Implementation



Defining a PIM (*ecore* model)



- Many options for creating an *ecore* model

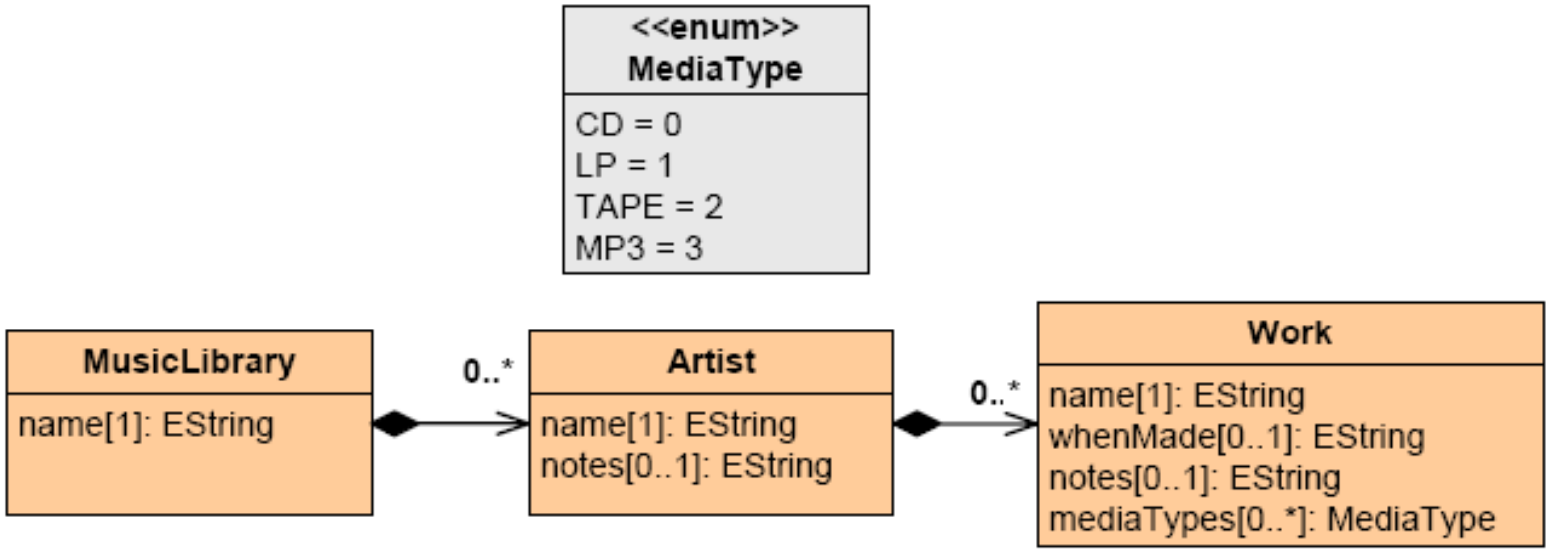
Options for Defining an *ecore* Model

- ***XML editor***
  - ecore files are XML files, hence you may use any XML editor
- ***ecore editor***
  - The EMF tool comes with a simple *ecore* editor
- ***UML Tools: Rational Rose, EclipseUML***
  - Marked up Rational Rose models can be converted to *ecore* files
  - EclipseUML provides native EMF support
- ***XML Schema Definition (XSD)***
  - An *ecore* model can be generated from a schema definition

Options for Defining an *ecore* Model

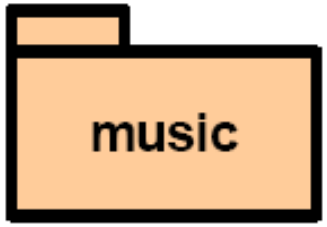
- ***Java Interfaces***
  - An *ecore* model can be generated from annotated Java
- ***Emfatic***
  - A Java-like language designed to express ecore

Music Library Example



- We'll work through an example to illustrate the ecore model
- A music library tracking ***Artists*** and their ***Work***

Defining a Package




```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <ecore:EPackage
3       xmi:version="2.0"
4       xmlns:xmi="http://www.omg.org/XMI"
5       xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
6       name="music">
7   </ecore:EPackage>
```

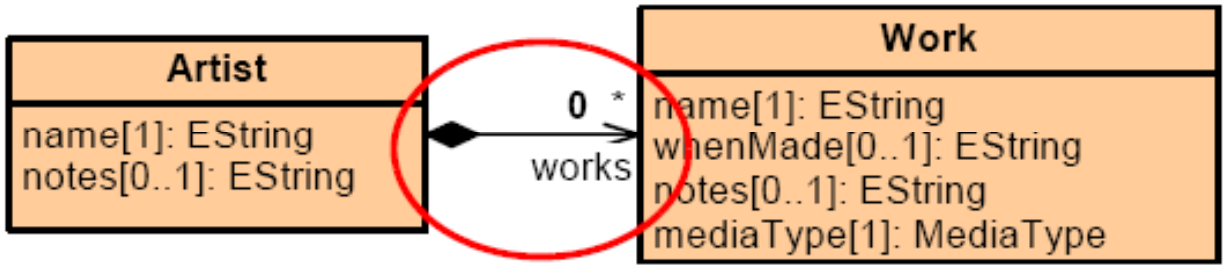
- Every ecore file starts with a ***EPackage*** declaration
  - ***xmi:version***
    Defines which XMI version from OMG is being used
  - ***xmlns:xmi*** and ***xmlns:ecore***
    Defines namespaces for the two XML Schemas being used
  - ***name***
    Defines the name of the package

Defining a Class

```
1   <eClassifiers
2       xsi:type="ecore:EClass"
3       name="Artist">
4       <eStructuralFeatures
5           xsi:type="ecore:EAttribute"
6           name="name"
7           lowerBound="1"
8           eType="ecore:EDataType http://www.eclipse.org/emf/2002/
      Ecore#//EString"/>
9       <eStructuralFeatures
10          xsi:type="ecore:EAttribute"
11          name="notes"
12          eType="ecore:EDataType http://www.eclipse.org/emf/2002/
      Ecore#//EString"/>
13  </eClassifiers>
```

- An ***EClass*** is defined with
  - ***eClassifier*** tag
  - Metareference ***xsi:type="ecore:EClass"***

Definition of an Association

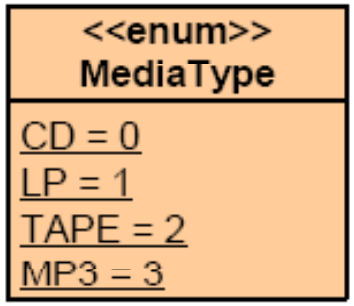



```
1   <eClassifiers xsi:type="ecore:EClass" name="Artist">
2       ...
3       <eStructuralFeatures
4           xsi:type="ecore:EReference"
5           name="works"
6           upperBound="-1"
7           eType="#//Work"
8           containment="true"/>
9       ...
10  </eClassifiers>
```

- Associations defined with ***eStructuralFeatures*** and ***xsi:type="ecore:EReference"***

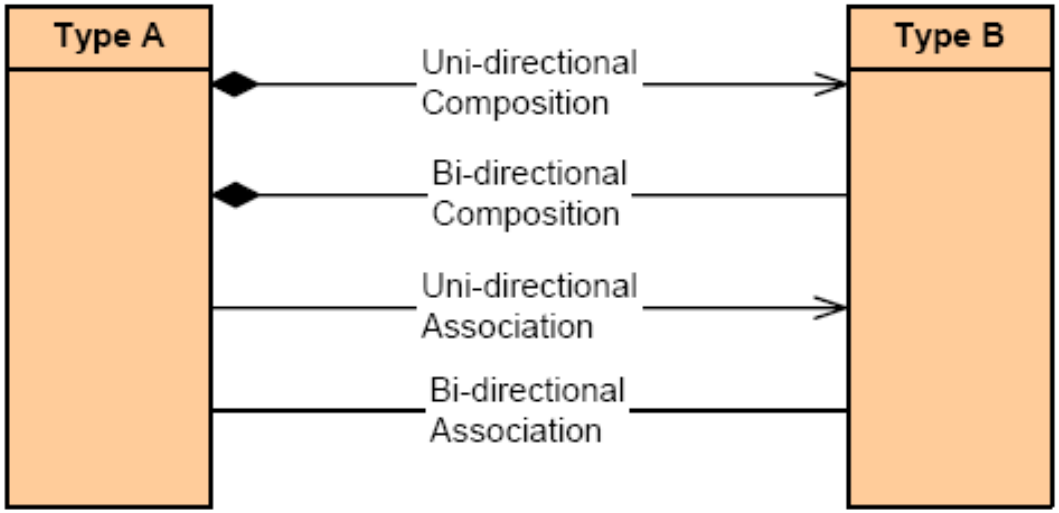
Definition of Enumerated Types

```
1   <eClassifiers
2       xsi:type="ecore:EEnum"
3       name="MediaType">
4
5           <eLiterals name="CD"/>
6           <eLiterals name="LP" value="1"/>
7           <eLiterals name="TAPE" value="2"/>
8           <eLiterals name="MP3" value="3"/>
9
10  </eClassifiers>
```



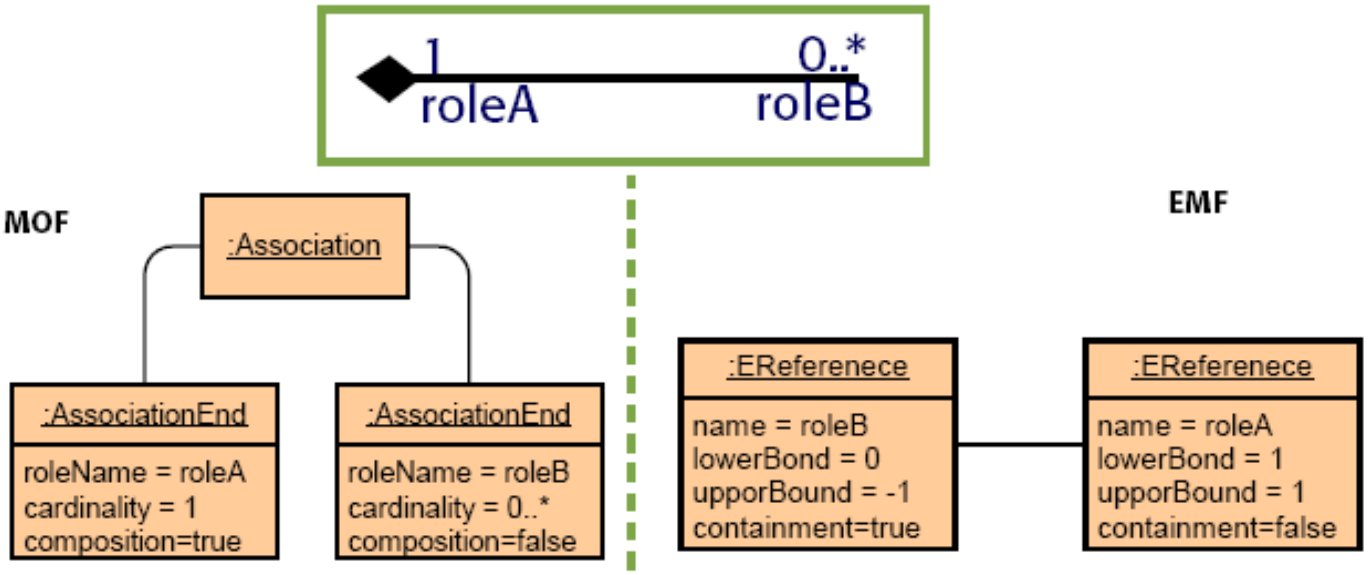
- Enumerated types:
  - ***eClassifier*** tag
  - ***xsi:type="ecore:EEnum"*** metareference
- Notice the definition of enum values by use of
  - ***eLiterals*** tag

Defining Associations



- Ecore supports the following main categories of associations
  - Directionality
    - Uni-directional
    - Bi-directional
  - Composition
    - Composition type
    - Association type

ecore Reference



- An ecore reference represents one end of an association
  - This divert a bit from the MOF world
- A bi-directional association requires two references
  - One for each end of the association
  - Each having defining the other as the opposite

Defining *ecore* using Java – Java Interfaces Annotations

- For programmers, an appealing option is to use Java Interfaces to define the model
- Mechanism:
  - Extension to ***JavaDoc***
    - ***@model*** [properties]
    - The ***properties*** are extensions using name-value pairs
  - ***Naming convention***
    - String getName() --> name: String

Defining a Package/EPackage

- The package used is the immediate package parent of the interface defined, e.g.:

```
1   package com.idata.music;
2
3   /**
4    * @model
5    */
6   interface Artist { ... }
```

- When imported:
  - EPackage::name = ***music***
  - The Artist class is put into the music package

Defining Classes

```
1   package com.inferdata.music;
2
3   /**
4    * @model
5    */
6   interface Artist { ... }
```

- If the JavaDoc comment prior to the interface declaration contains a ***@model*** tag
  - The ***interface*** is mapped to an ***EClass*** object in ecore
  - The ***EClass::name*** attribute is mapped to the ***interface name***

```
1   /**
2    * @model
3    */
4   interface SpecialArtist extends Artist {...}
```

- Standard ***extension*** mechanism between interfaces serves to define ***inheritance*** between ***EClasses***

Defining Attributes

```
1   package com.idata.music;
2
3   /**
4    * @model
5    */
6   interface Artist {
7       /**
8        * @model
9        */
10      String getName();
11  }
```

- Declaration of get methods define attributes

```
1       /**
2        * @model default="My Favourite Band"
3        */
4       String getName();
```

- Attributes may have default values
  - New instances of the type will be initialized with this value
  - If the default value is selected, no storage is required

Defining Associations

```
1   package com.idata.music;
2   import org.eclipse.emf.common.util.EList;
3   /**
4    * @model
5    */
6   interface Artist {
7       /**
8        * @model
9        */
10      String getName();
11
12      /**
13       * @model type="Work" opposite="artist" containment="true"
14       */
15      EList getWorks();
16  }
```

- ***type*** defines the kind of element we are referencing
- ***opposite*** declares a link to the opposite reference
- ***containment*** declares if this association is a composition type
- Two ways associations require two references, both defining

Defining an Enumeration

```
1   package com.idata.music;
2   /**
3    * @model
4    */
5   public final class MediaType {
6       /**
7        *@model
8        */
9       public final static int CD=0;
10      /**
11       *@model
12       */
13      public final static int MP3=1;
14      /**
15       *@model
16       */
17      public final static int TAPE=2;
18  }
```

- Enumerations are defined as ***final classes***
  - The enumerated values as ***finals static int***

The Generator Model

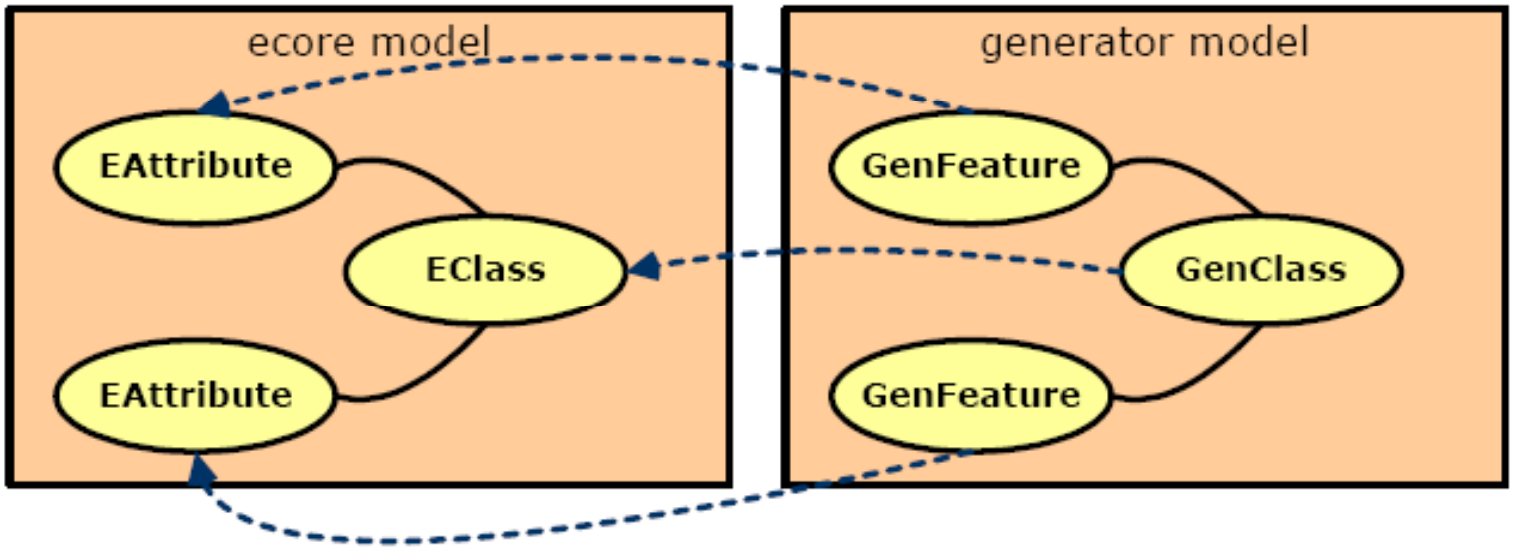
The Role of the *genmodel*

- In addition to the ecore model, we also need a ***genmodel***
- The ***genmodel*** provides the platform specific information
  - As opposed to the *ecore* model that holds only platform independent information
- A ***genmodel*** is required to generate code
- The genmodel allows you to configure how you want your code generated, e.g.:
  - What packages to use
  - How to display the model structure

What Must Be Configured?

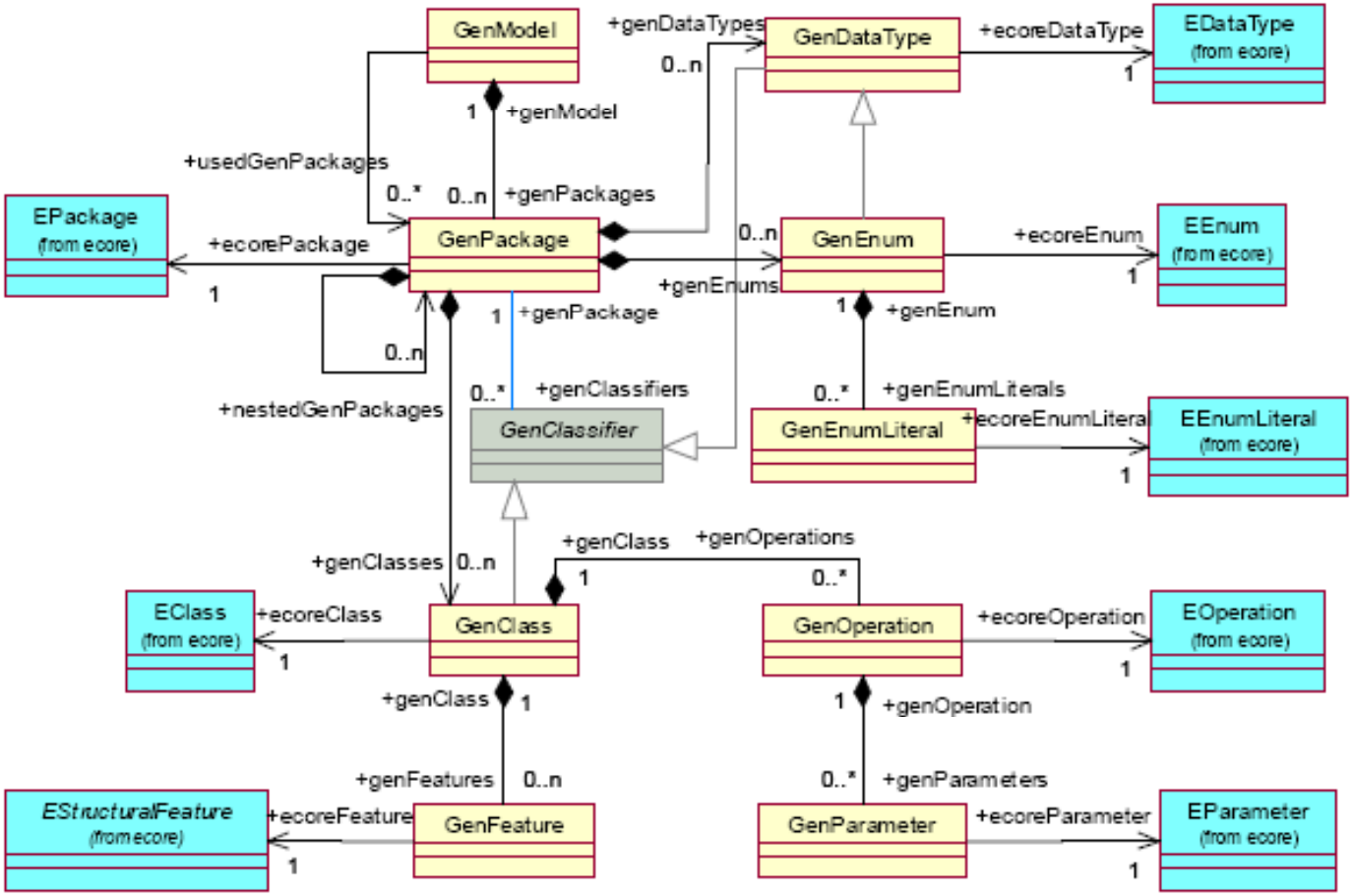
- The genmodel contains a set of options for how to generate a plug-in editor for eclipse based on the ecore model
- Code organization and meta data
  - Copy right text
  - Package information
  - ...
- Presentation options
  - Show as tree?
  - Which attribute makes up the label
  - ....
- ...

Relationship Between *ecore* and *genmodel*



- The *genmodel* holds one element for each element in the *ecore model*
  - The objects in the *genmodel* contains:
    - A reference to a *ecore* element
    - Code generation options

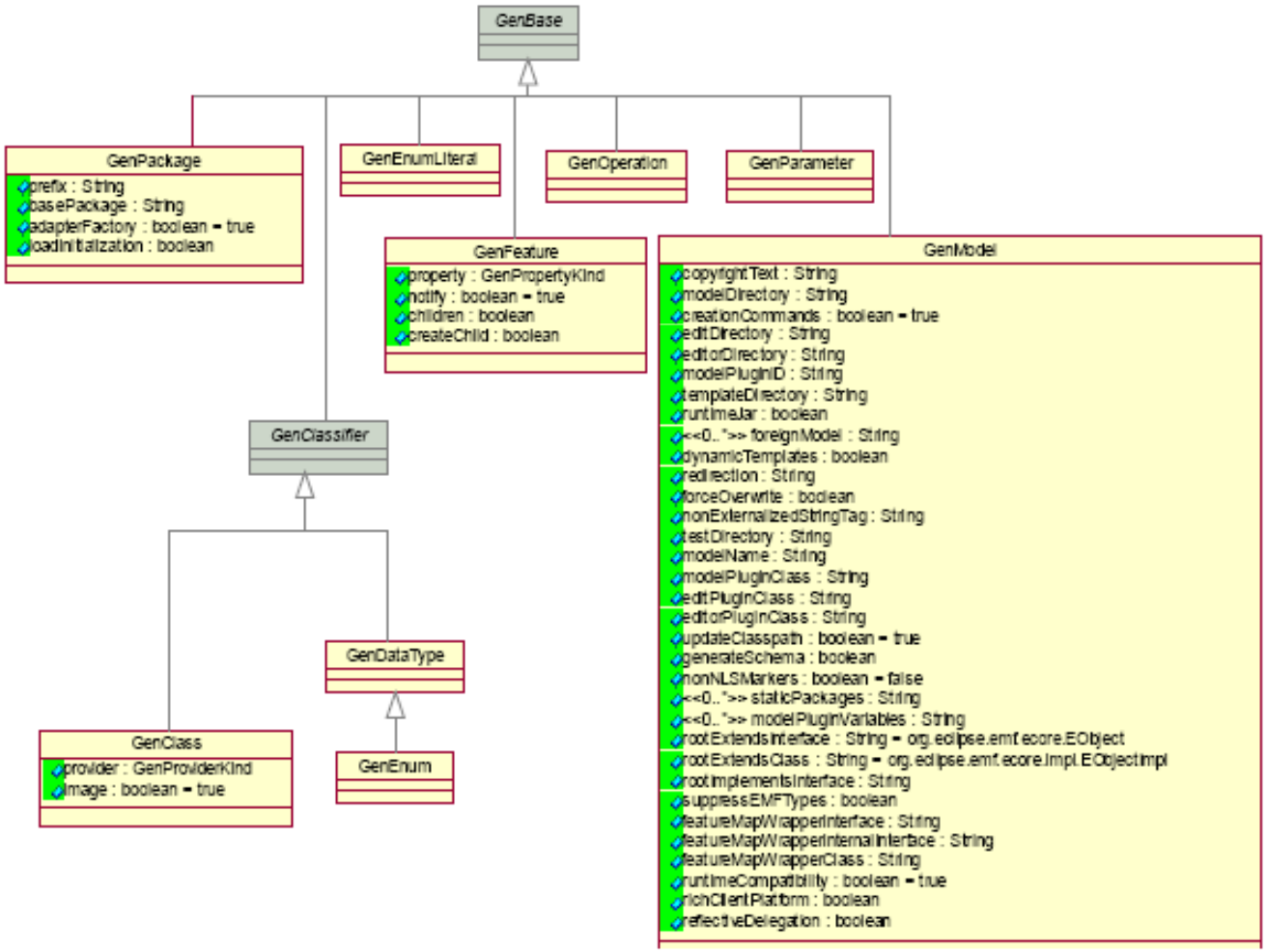
Generation Model



What Can Be Configured

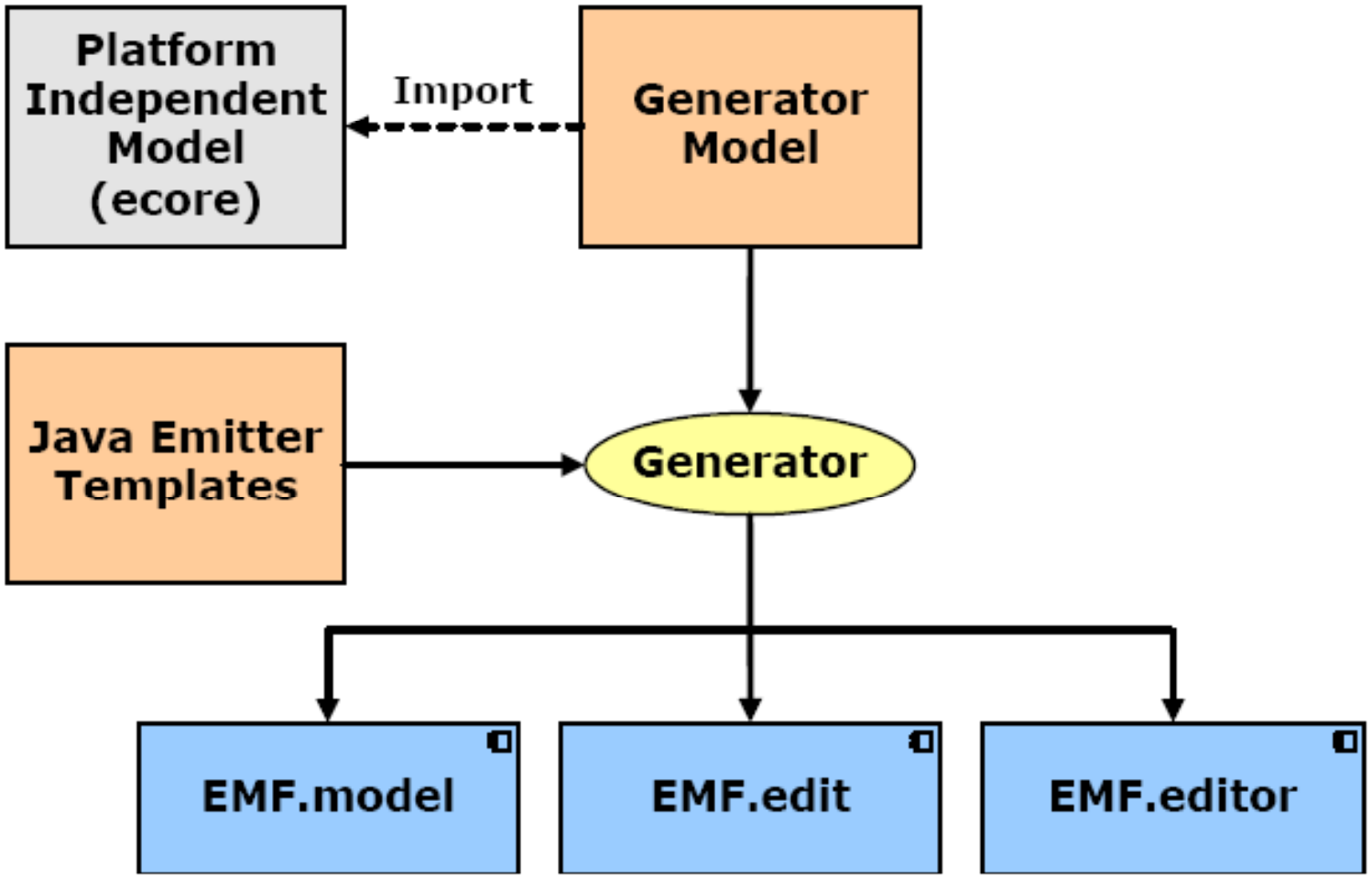
- General configuration options (and subcategories)
  - All
  - Edit
  - Editor
  - Model
  - Model Class Defaults
  - Model Feature Defaults
  - Templates and Merge
- Model elements
  - GenPackage
  - GenClass
  - GenFeature

MetaModel for Configuration

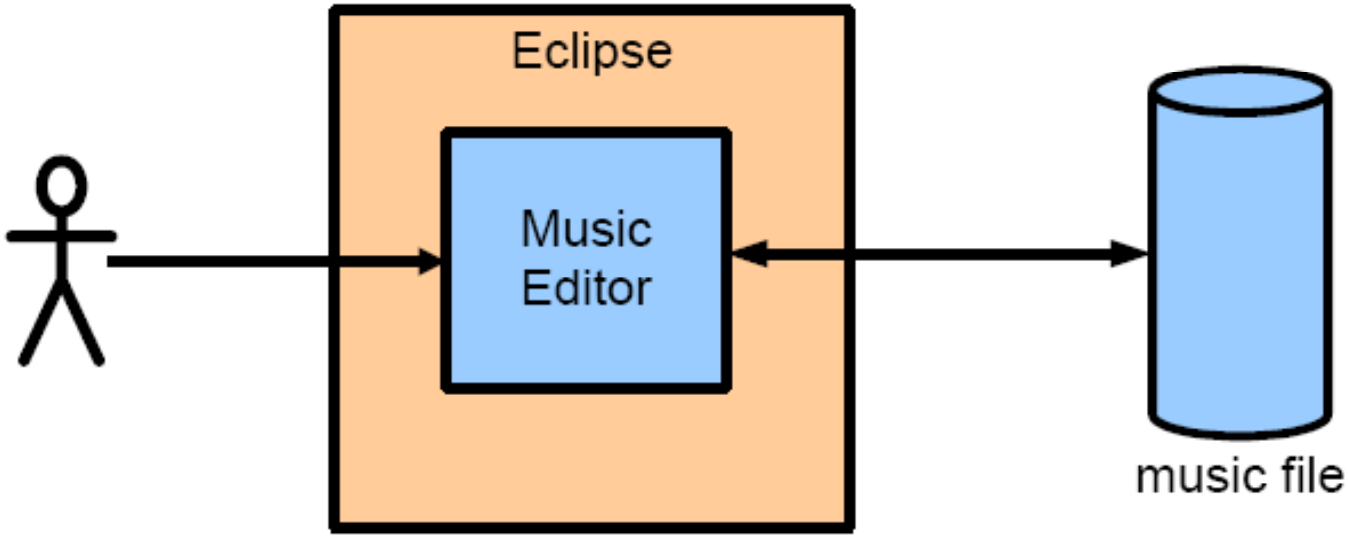


Code Generation

Code Generation Overview

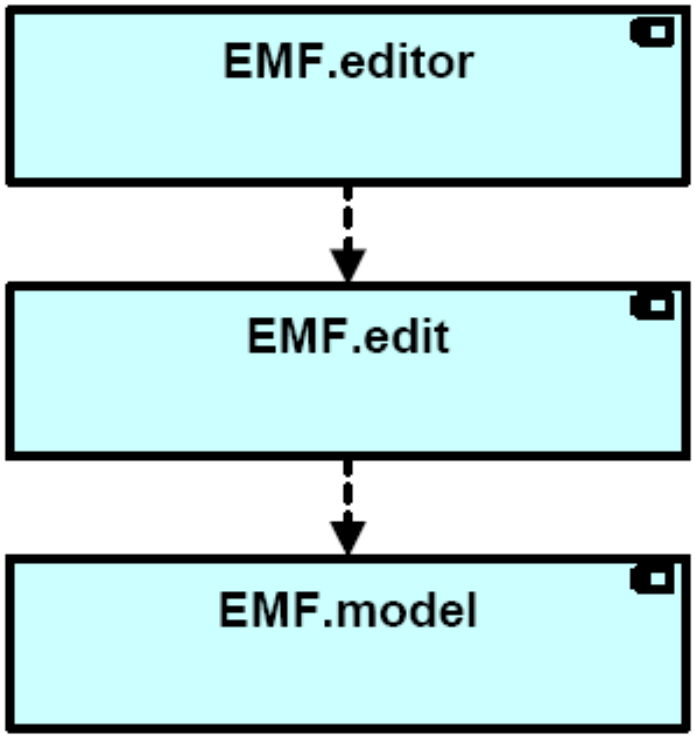


About the Generation Implementation



- The EMF generator creates an editor for content based on the PIM
  - Plug-in for eclipse
  - A default XML serialization

Plug-ins Created by EMF



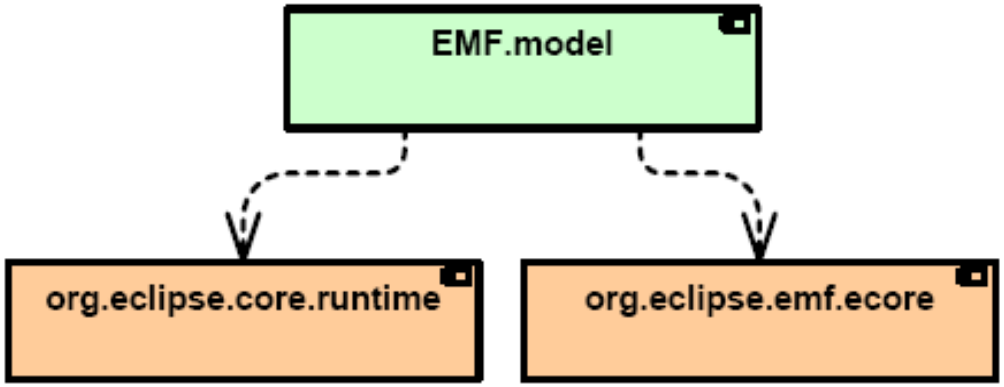
Holds presentation code

Holds presentation independent adaptors

Holds the PIM based model code

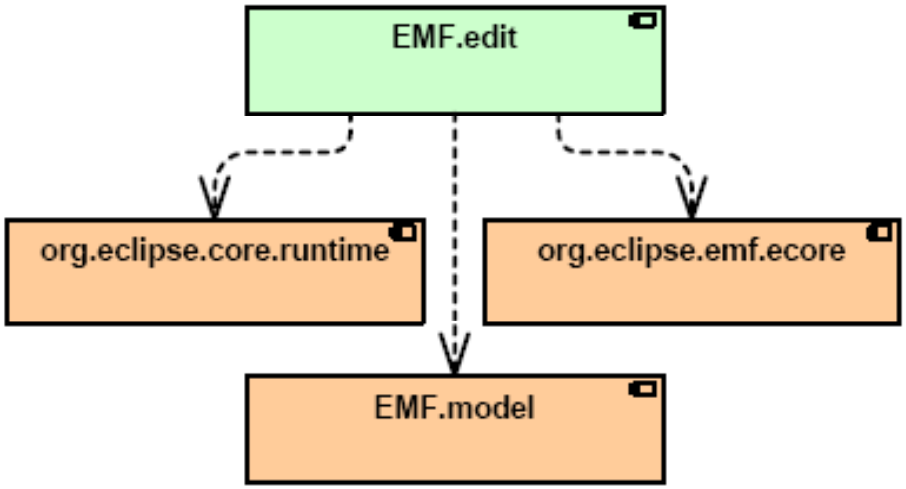
- EMF can create three different plug-ins

The EMF.model



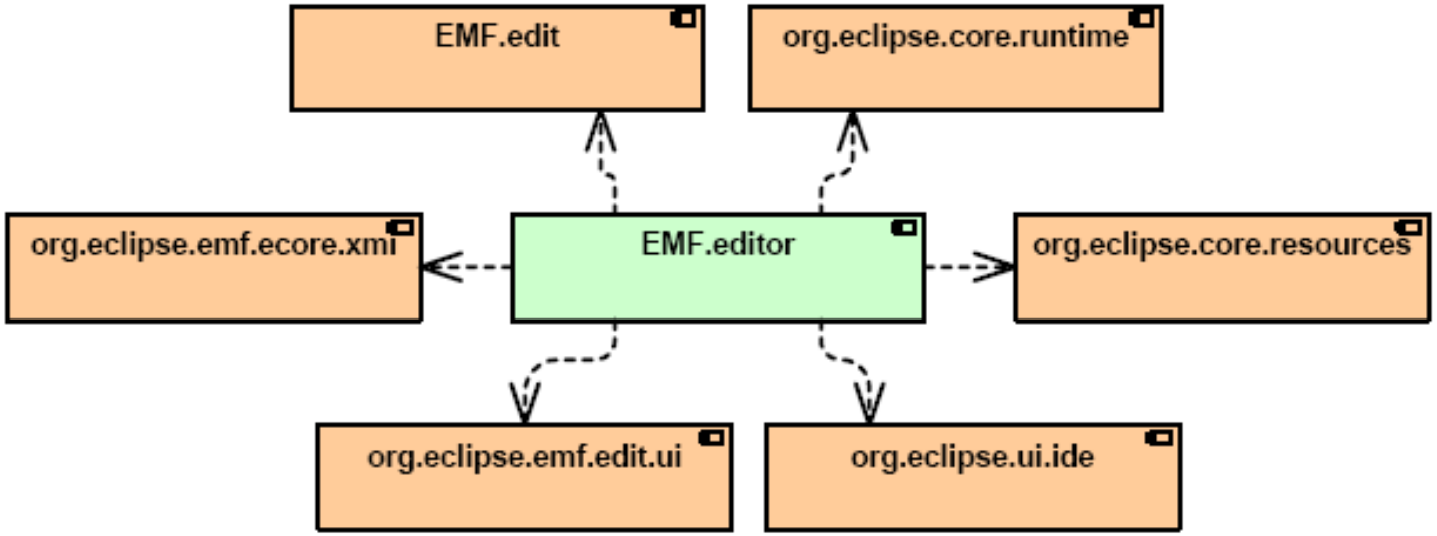
- The model code provides a complete implementation of the PIM
- Highly efficient persistence mechanisms on top of XML files
- 100% continuity from model to code
  - All code generated is predictable
  - Typically, little or no modification of the code required

EMF.edit



- The EMF.edit acts as a presentation independent layer adapting model objects
  - Label providers
  - Tree models
  - Commands
  - ...

EMF.editor



- The EMF.editor provides the code SWT/JFace code that directly interacts with the user

Can Everything be Generated

- ***THE ANSWER CURRENTLY IS: DEFINITELY NOT!!!***
- It is highly likely that you need to make some changes to the generated code
- Usual change patterns:
  - ***EMF.model***
    - Utility operations declared in EMF but manually implemented
  - ***EMF.edit***
    - Changing display behavior
      E.g. concatenating attributes to make up a label
    - Restricting what is shown in the user interface based on some non-declarative rule
  - ***EMF.editor***
    - Often completely rewritten

How to Change the Generated Code?

- All generated code holds the java-doc comment ***@generated***

```
/**
 * This returns the image for the artist type.
 * @generated
 */
public Object getImage(Object object) {
  return getResourceLocator().getImage("full/obj16/Artist");
}
```

- To take over the code from the generator, change the @generator tag

```
/**
 * This returns the image for the artist type.
 * @generated NOT
 */
public Object getImage(Object object) {
  return complexCalculationOfImage( object );
}
```

Integrity of Non-Generated Code

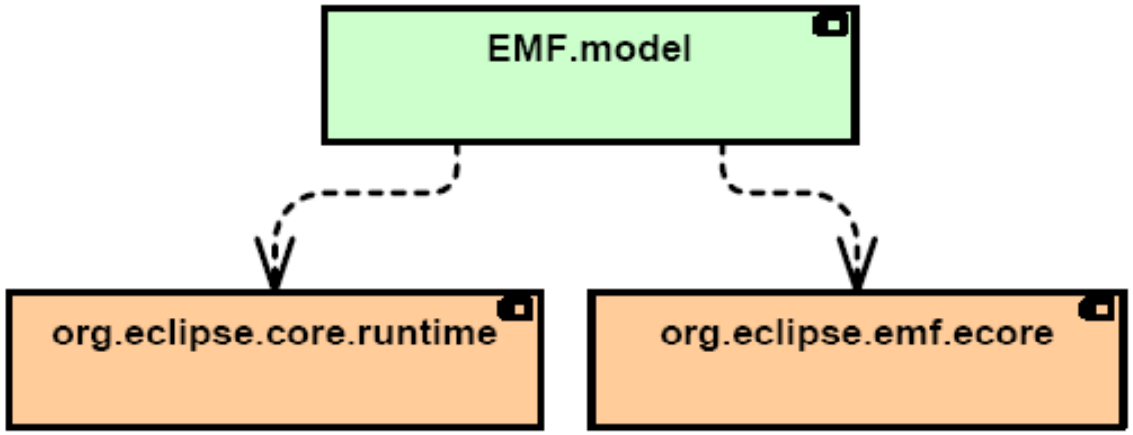
- Only code with @generated tag will be overridden when subsequent code generation is performed

EMF.model

EMF.model

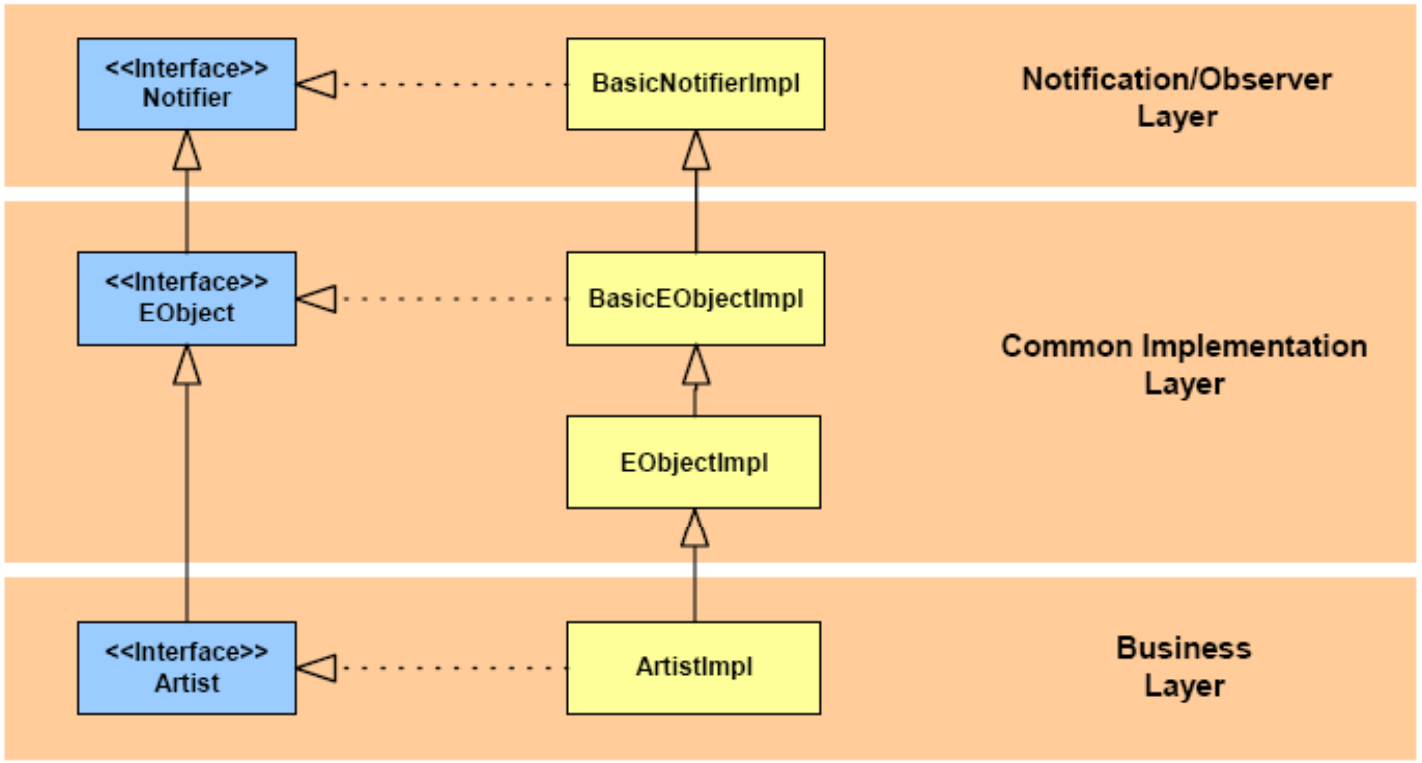
- The EMF.model plug-in contains the code related to:
  - Business domain structure
  - Persistence
- We often go to great length to ***avoid making changes*** to this plug-in
- Typically, only ***EOperations*** are modified
  - EMF does not provide action semantic
- Sometimes useful to reimplement ***toString***
  - To provide a human readable string presentation of a business object

Dependencies



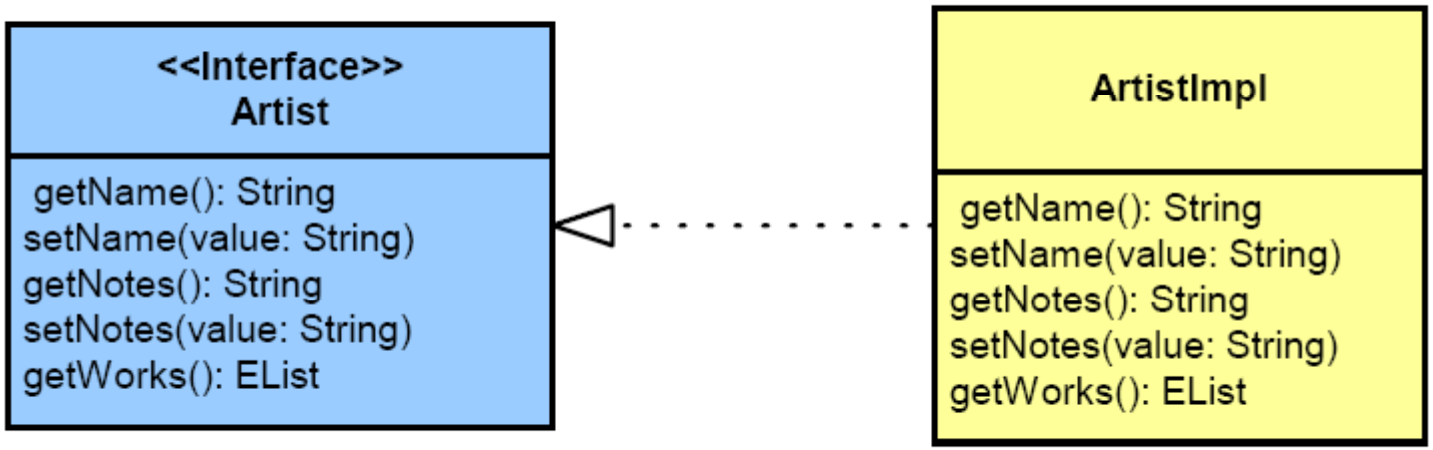
- The EMF.model depends on only two plug-ins
  - org.eclipse.core.runtime
  - org.eclipse.emf.ecore
- It is possible to setup the genmodel options such that the EMF.model can run outside Eclipse

Implementation of EClass



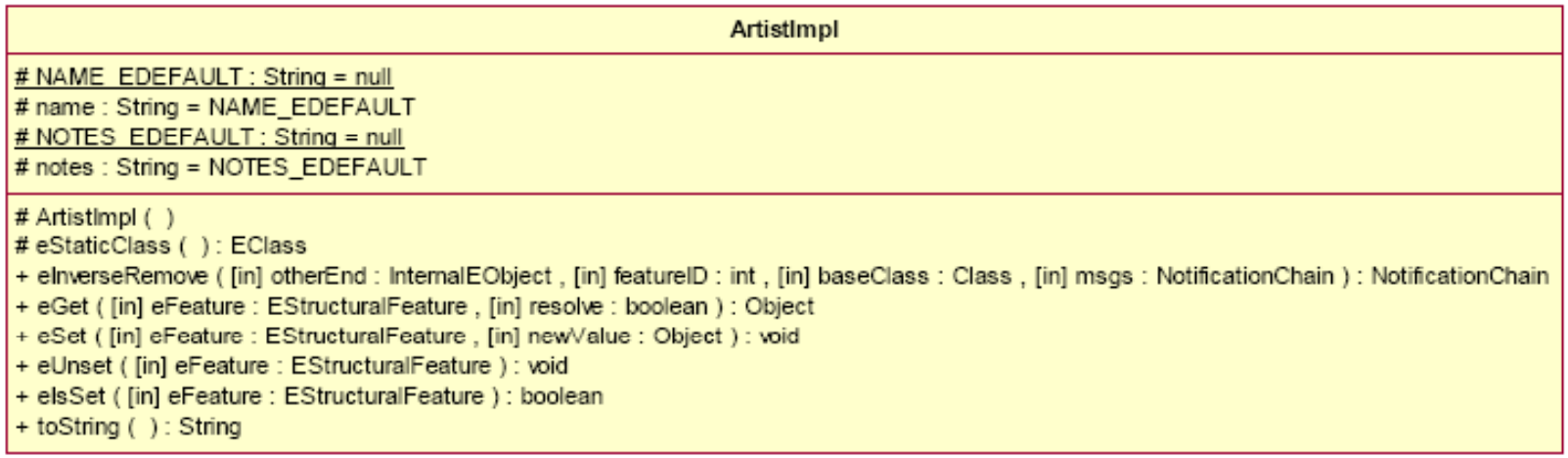
- The generated EMF.model implementation extends a predefined framework

Business Implementation



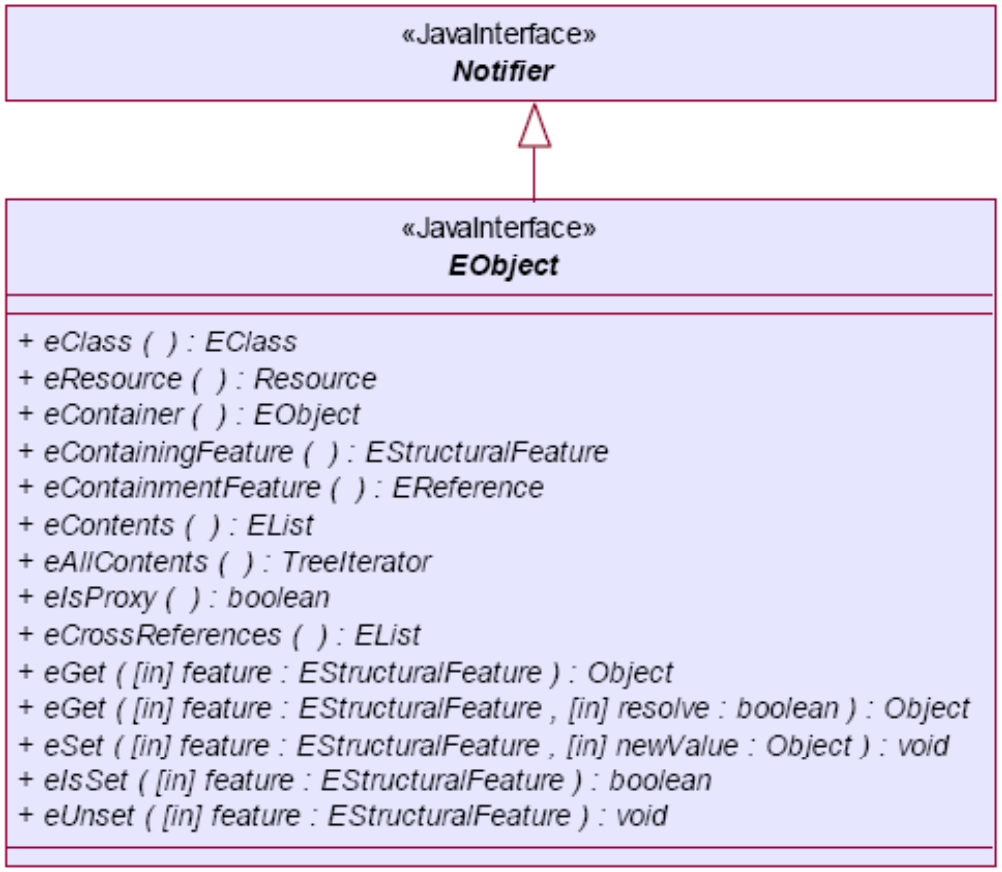
- ***EAttribute*** implementation
  - Get and set methods
- ***EReference*** implementation
  - Get method for many
  - Get and set method for one

Framework Generated Implementation



- Each generated class has a set of methods generated which are there to support the framework
  - Reflective set and get methods
  - Support for EClass (reflection)
  - Initialization and storage of default values

EObject Interface



- All business object interfaces extends EObject

Eattribute Implementation

```
protected static final String NAME_EDEFAULT = null;

protected String name = NAME_EDEFAULT;

public String getName() {
  return name;
}

public void setName(String newName) {
  String oldName = name;
  name = newName;
  if (eNotificationRequired())
    eNotify( new ENotificationImpl(...));
}
```

- The attributes becomes get and set methods
- You typically never override the attribute implementation

EReference Implementation

- The EReference is implemented similar to the EAttribute
- The only difference is that the attribute is now one of the business objects
- Some additional implementation is required if the reference is not of type composition
  - We may have to resolve the reference
    - The object could be in a different resource
  - We may have to ensure the integrity of two-way references
    - Remember the concept of opposite in the ecore model?

EOperation Implementation

- It is possible to define operations in the ecore model
- There is no support for defining the semantic of the operations in ecore
- The idea is to:
  - Declare the business operations in the ecore model
    - Name of the operation
    - Return value type
    - Parameters
  - Implement the semantic in Java
- The code generator will generate hooks for you to implement the semantic

Defining the EOperation

```
public class XImpl extends EObjectImpl implements X {

   /**
    * @generated NOT
    */
   void f() {
     // Provide the implementation
   }
}
```

- When first generated, EMF generates a dummy implementation throwing exceptions
- To take ownership of the code
  - Set the @generated tag to **NOT**
  - Implement the method

EMF.edit

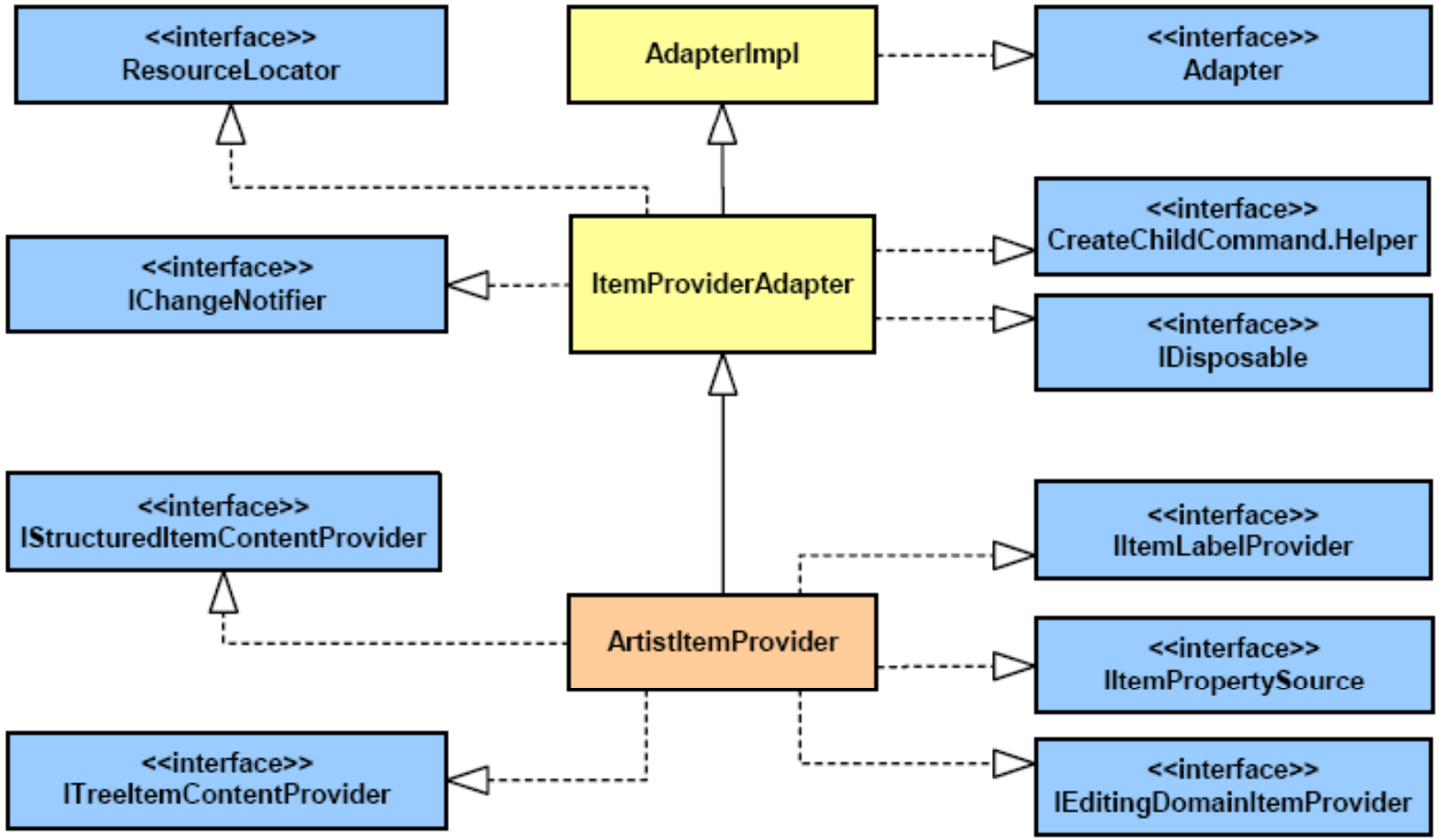
Role of EMF.edit

- The EMF.edit ***separates*** the ***GUI*** from the ***business model***
  - User interface independent implementation of the interaction domain
- Expect to modify the EMF.edit plug-in
- Typical changes include:
  - Modification of the item provider
  - Introducing new commands

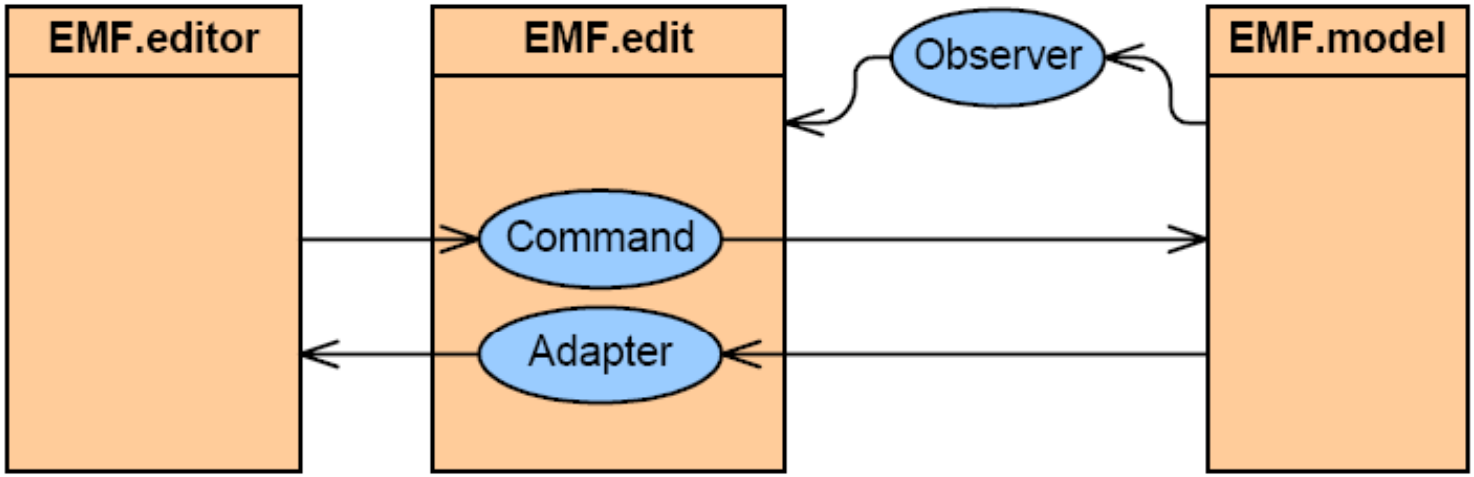
Generator Pattern

- For every business object an adapter is created in the EMF.edit plug-in
  - Called ***ItemProvider***
  - E.g., ArtistImteProvider
- The Item Provider extends ***org.eclipse.emf.edit.provider.ItemProviderAdapter***
  - Contains default implementation for most of the required functionality
  - Extending the edit framework often involves overriding one of its methods

Framework Generator Structure



EMF.edit and Patterns



- To understand the EMF.edit plug-in, it is essential to understand three basic design patterns
  - Observer pattern
  - Command pattern
  - Adapter pattern

Changing the Label

- A typical change to the adapter is to change the item provider
  - The genmodel allows us to select which attribute to use as a label for a model element
  - What if we want to concatenate two fields?
  - E.g. from the music domain, let's say we want to print out the work year and name together
    - ***YEAR: Work Name***
- Requires change to ***WorkItemProvider::getText***
  - Change to the javadoc tag @generated
  - Reimplement the method

Label Change Code

```
/**
 * This returns the label text for the adapted class.
 * <!-- begin-user-doc -->
 * Returns the presentation string for the work
 *     <YEAR>: <Name>
 * <!-- end-user-doc -->
 * @generated NOT
 */
public String getText(Object object) {

  String label = ((Work)object).getName();
  return label == null || label.length() == 0 ?
    getString("_UI_Work_type") :
    getString("_UI_Work_type") + " " + label;


  String label = this.getYear();
  if (label == null || label.size() == 0 )
    label = "????";
  label += ": " + this.getName();
  return label;

}
```

Changing the Icon Representation

- Another common change to the edit model is to change the icon representation for an item
- The genmodel generates a simple icon file for each business object type
  - Located in two directories
    - emf.edit/icons/full/obj16
      Contains an image file for each business object type
    - emf.edit/icons/full/ctool16
      Contains an image file for each creation possibility for a business object type
- The most common change is just to override the image file with your new representation

Changing Icons in Code

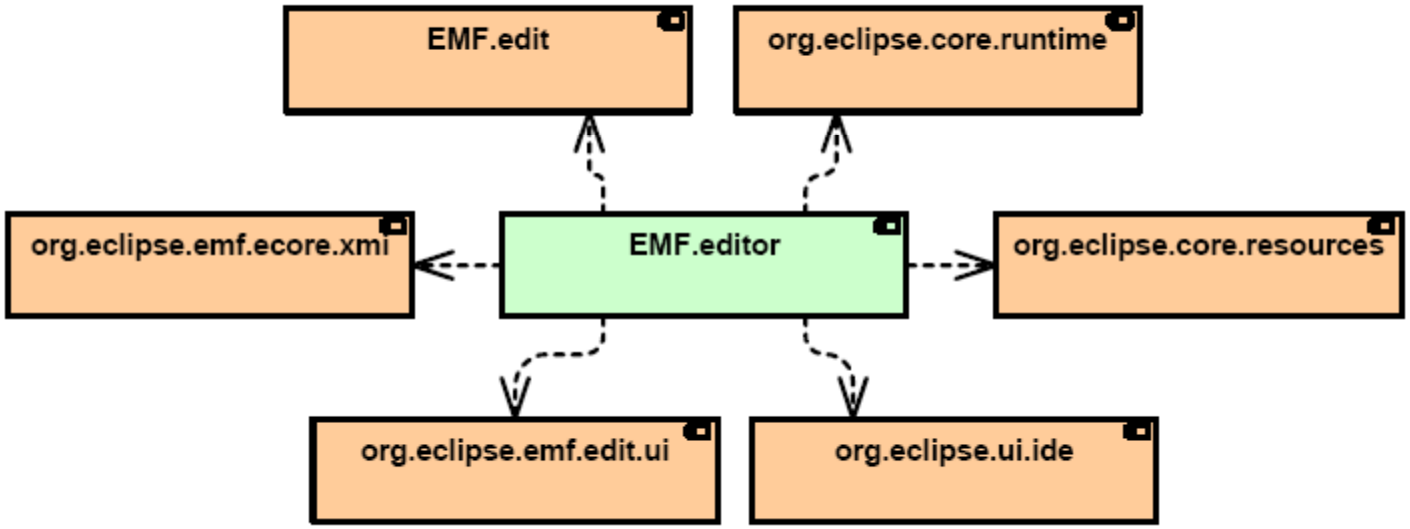
- Sometimes changing the icon is not enough
  - State based icons
  - Context sensitive icons
- We can now change the code as we did for the string representation
- Let's say we want the icon for the work to change based on the media type
  - Reimplement ***WorkItemProvider::getImage()***

Changing the Image by Code

```
/**
 * This returns Work.gif.
 * <!-- begin-user-doc -->
 * Returns an icon based on the media type
 * <!-- end-user-doc -->
 * @generated NOT
 */
public Object getImage(Object object) {
  return getResourceLocator().getImage("full/obj16/Work");
  int mt = ((Work)object).getMediaType().getValue();
  switch (mt) {
    case MediaType.CD:
      return getResourceLocator().getImage("full/obj16/CD");
    case MediaType.LP:
      return getResourceLocator().getImage("full/obj16/LP");
    case MediaType.MP3:
      return getResourceLocator().getImage("full/obj16/MP3");
    case MediaType.TAPE:
      return getResourceLocator().getImage("full/obj16/TAPE");
  }
  return getResourceLocator().getImage("full/obj16/Work");
}
```

EMF.editor

Role of EMF.editor



- The EMF.editor provides the code SWT/JFace code that directly interacts with the user
- Two main options here
  - Leave it, it is good enough
  - Reimplement, it is not even close to what we want

Is It Good Enough?

- If you are using EMF to build stand alone editors, the answer is probably ***NO***
- Must change the new wizard
  - Avoid user selected root elements
  - The category for the wizard should not the "EMF Example New Wizards"
- More than one view
  - The multi-page editor only activates the tree editor properly
  - Change to custom dialog likely

What's Generated?

- ***MusicEditor.java***
  - The main editor code
  - Bridges the U/I Events to U/I Actions
  - Sets up and activates the workbench pages
- ***MusicActionBarContributor.java***
  - Defines and configures menus
- ***MusicModelWizard.java***
  - Implements the new wizard
- ***MusicEditorPlugin.java***
  - Bootstrap code for the plug-in
  - Most of the behavior is in the superclass EMFPlugin

References

- This presentation was adapted from:
  - Backvanski, V. and Graff, P. "*Mastering Eclipse Model Framework*". EclipseCon2005. www.eclipsecon.org/2005/presentations/EclipseCon2005_Tutorial28.pdf.
- Others References
  - Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, J. G. "*Eclipse Modeling Framework: A Developer's Guide*". 2003.
  - Steinberg, D. "*Fundamentals of the Eclipse Modeling Framework*". 2008. www.eclipse.org/modeling/emf/docs/presentations/EclipseCon/EclipseCon2008_309T_Fundamentals_of_EMF.pdf.