

# Mastering Eclipse Modeling Framework

**Vladimir Bacvanski**  
**Petter Graff**

*vladimir@inferdata.com*  
*petter@inferdata.com*

InferData Ltd.



# Outline

- Introduction to EMF
- The ecore Model
- The Generator Model
- Code Generation
- EMF.model
- EMF.edit
- EMF.editor
- Summary and Conclusions

# Introduction to EMF



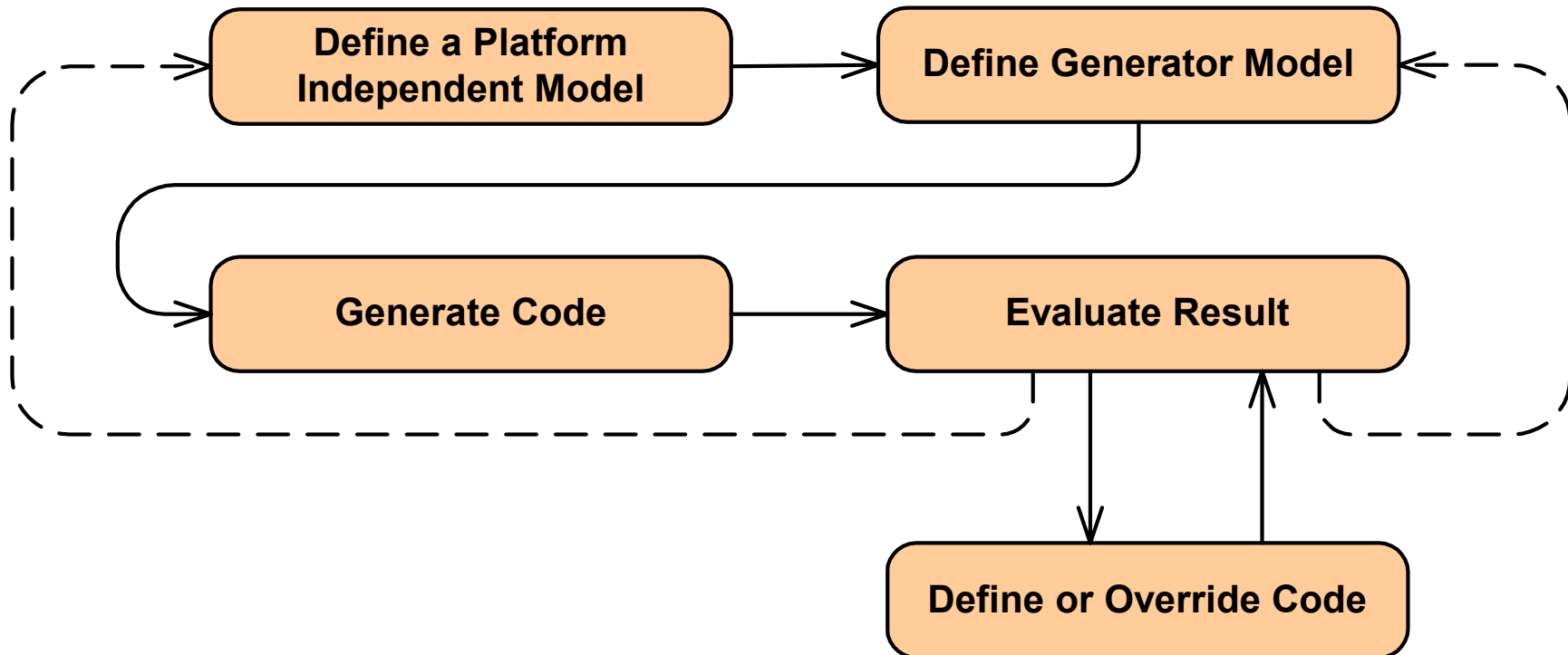
# What is Eclipse Modeling Framework (EMF)?

- 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

# EMF History

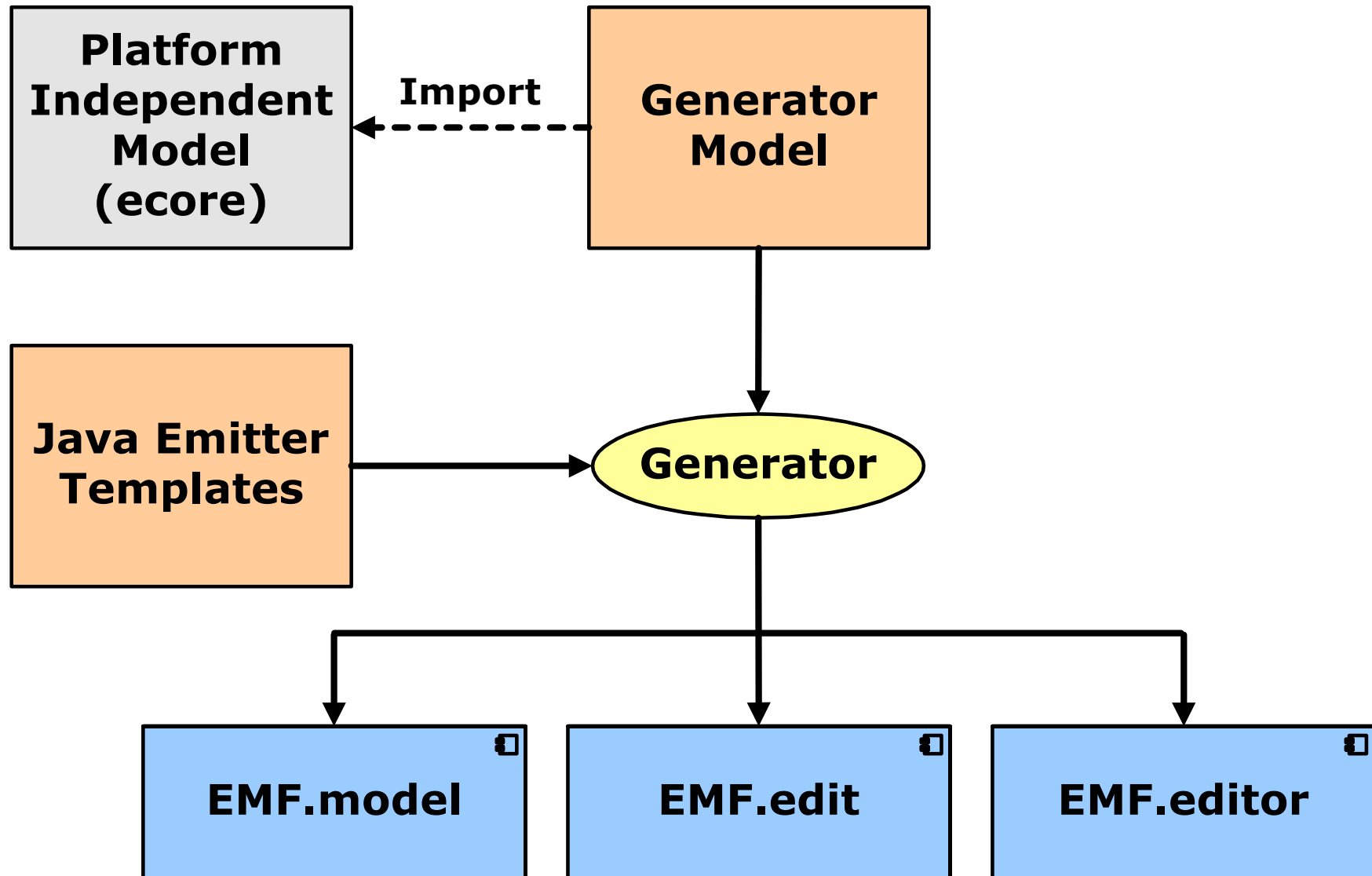
- EMF *evolved from experiences* building editors for WebSphere Studio family of products
  - Later it has been aligned with the OMG's MDA approach
  - Built on MOF 2.0's EMOF

# How to Work with EMF?



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

# EMF Toolset from 30.000 Feet



# The *ecore* Model



# Introduction

- ***Introduction***
  - The purpose of ecore
- Defining an *ecore* model
- Defining *ecore* in XML
- The *ecore* Editor
- Defining *ecore* using UML tools
- Defining *ecore* using Java
- Defining *ecore* using XML Schema
- Defining *ecore* using Emfatic

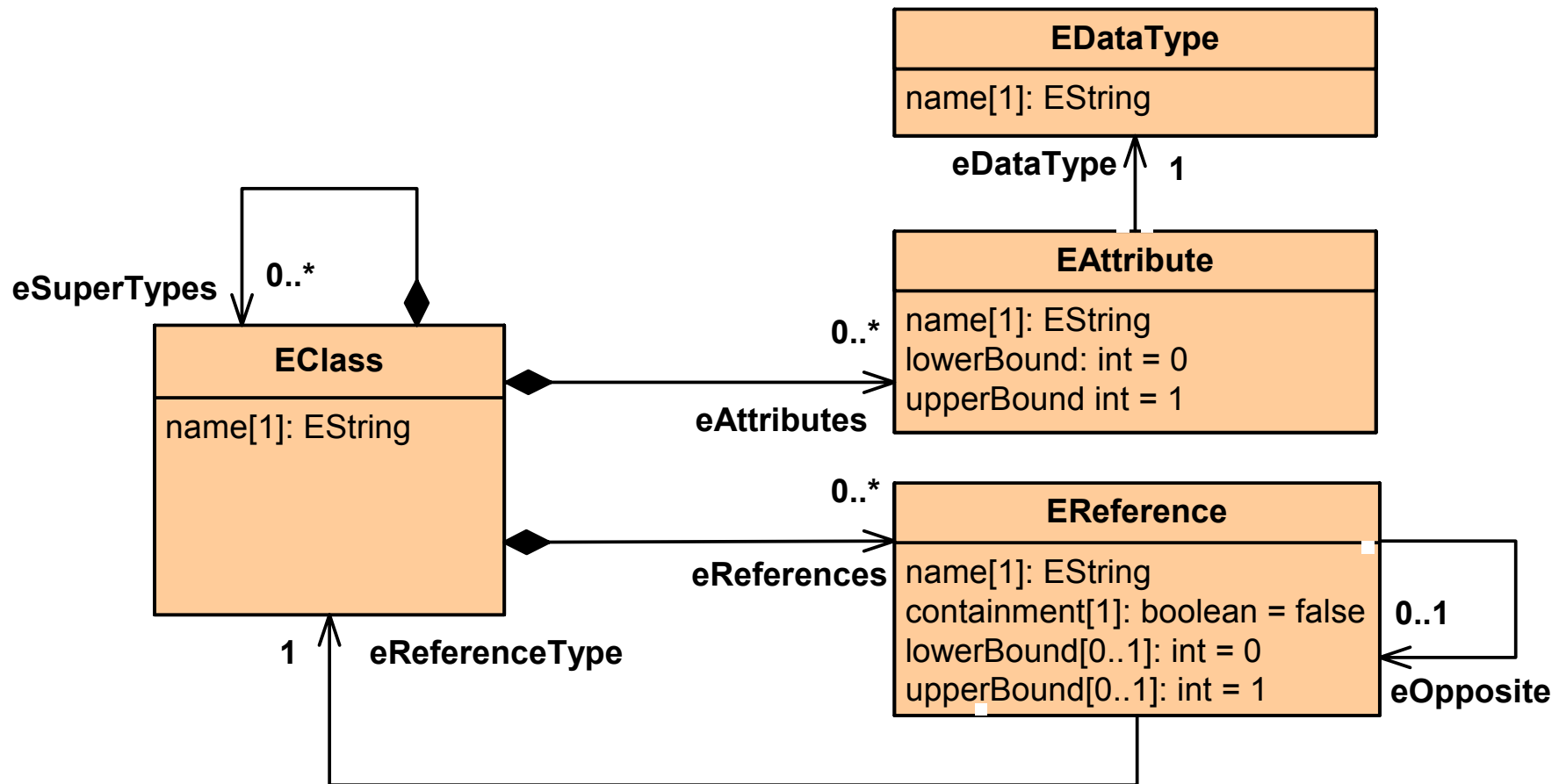
# What is ecore?

- 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 Is 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

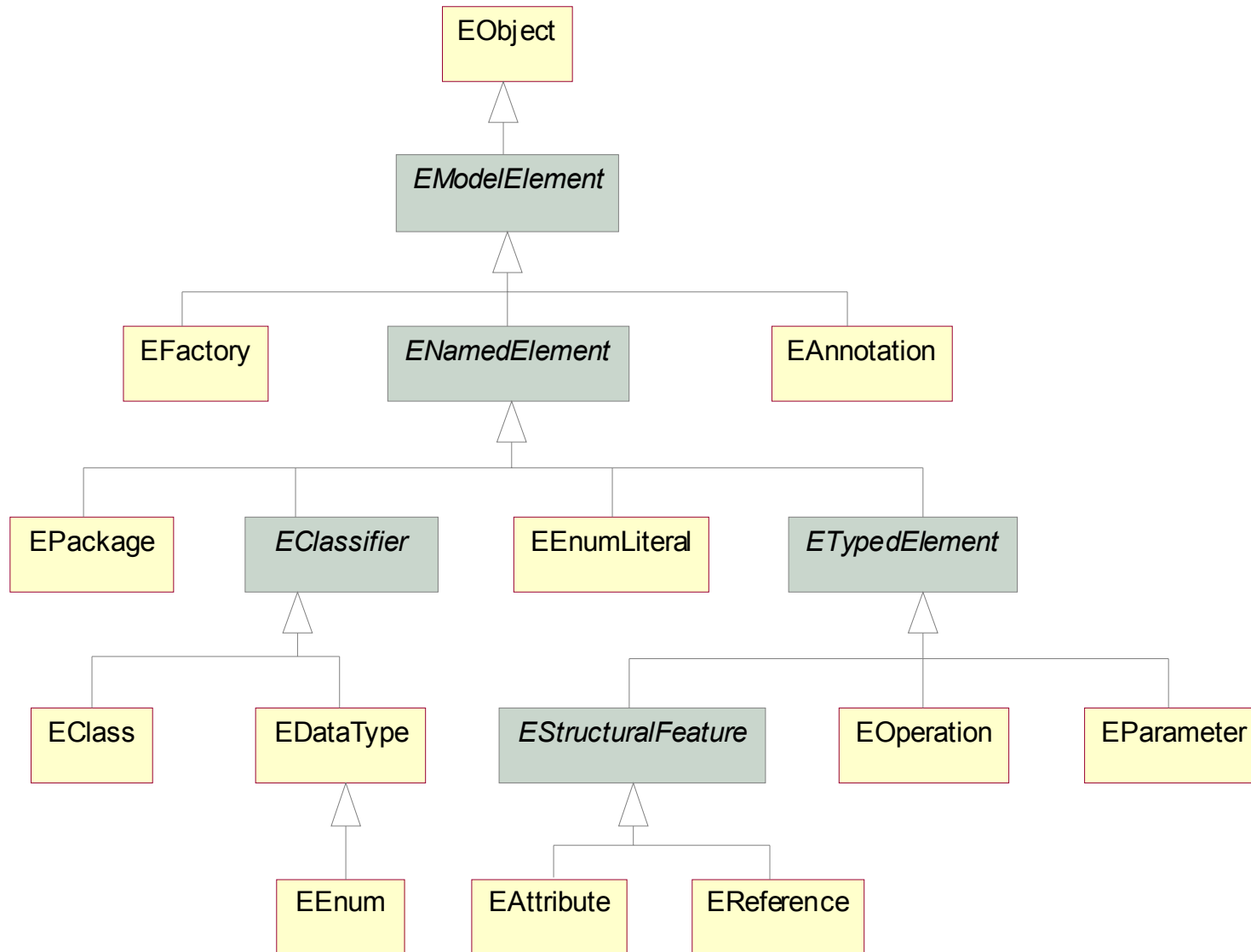
## ■ ***EAttribute***

- Represents a typed attribute

## ■ ***EReference***

- Represents an association end
- Optionally points to the opposite association end
- Defines the referenced type

# ecore Hierarchy

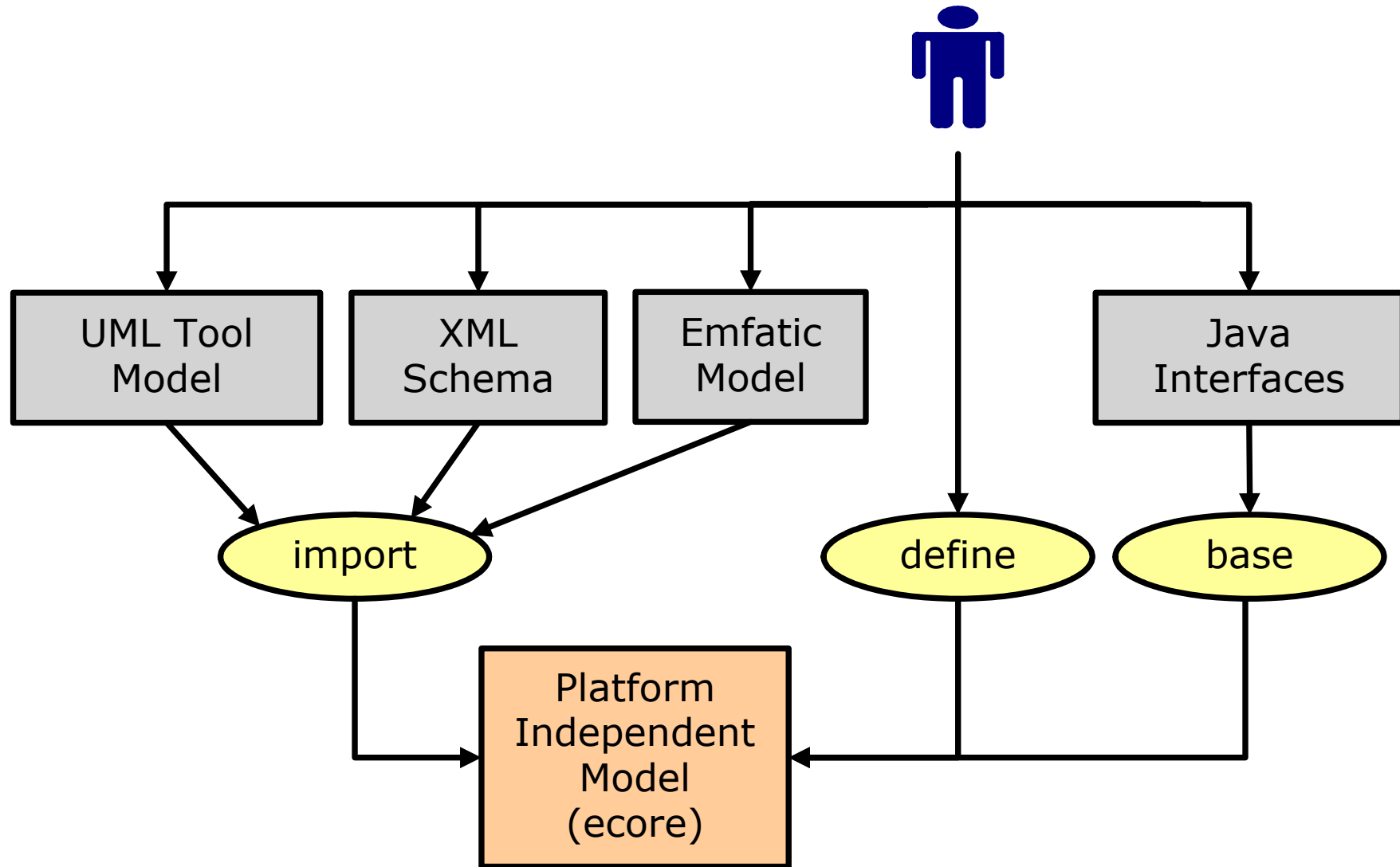




# Defining an *ecore* Model

- Introduction
- ***Defining an ecore model***
  - Options for creating *ecore* models
  - Introduction of a music library example
- Defining *ecore* in XML
- The *ecore* Editor
- Defining *ecore* using UML tools
- Defining *ecore* using Java
- Defining *ecore* using XML Schema
- Defining *ecore* using Emfatic

# Defining a PIM (ecore model)



- Many options for creating an *ecore* model

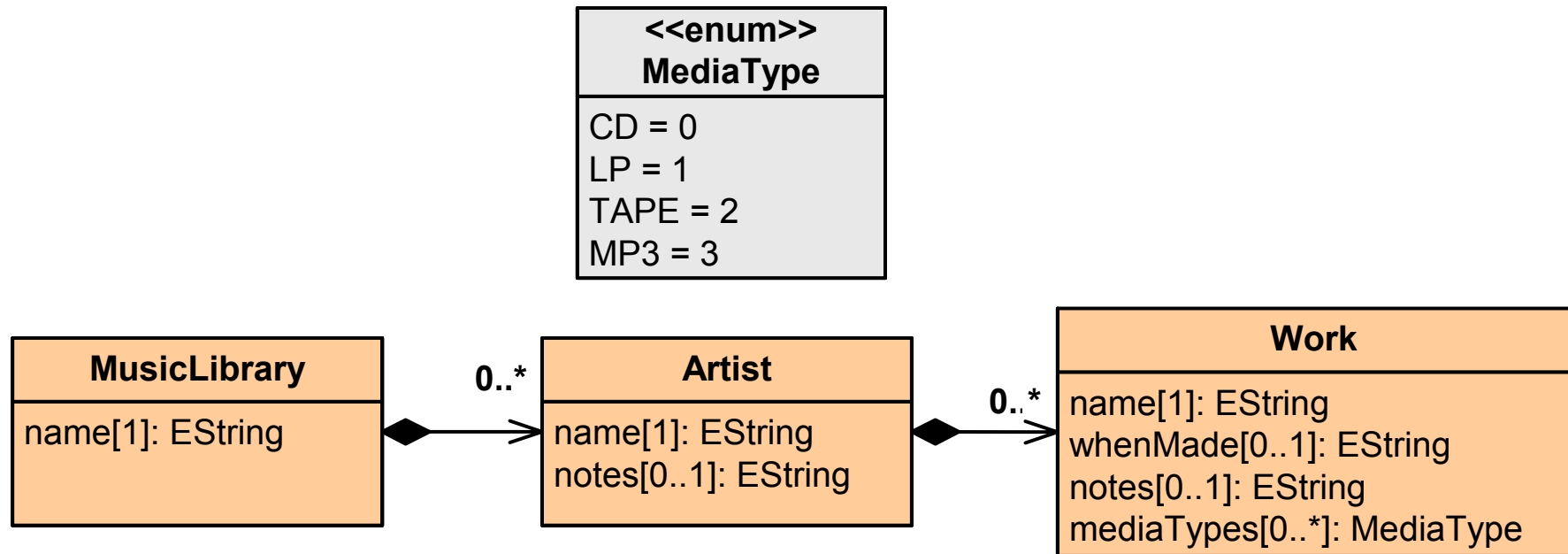
# Options for Defining an *ecore* Model

- ***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

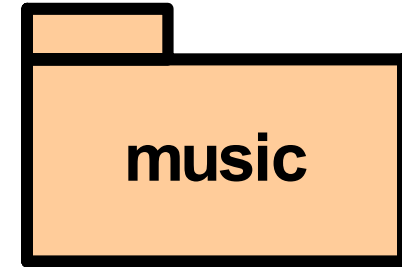


- We'll work through an example to illustrate the ecore model
- A music library tracking **Artists** and their **Work**

# Defining *ecore* in XML

- Introduction
- Defining an *ecore* model
- ***Defining ecore in XML***
  - Using XML editors to create *ecore* models
- The *ecore* Editor
- Defining *ecore* using UML tools
- Defining *ecore* using Java
- Defining *ecore* using XML Schema
- Defining *ecore* using Emfatic

# 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

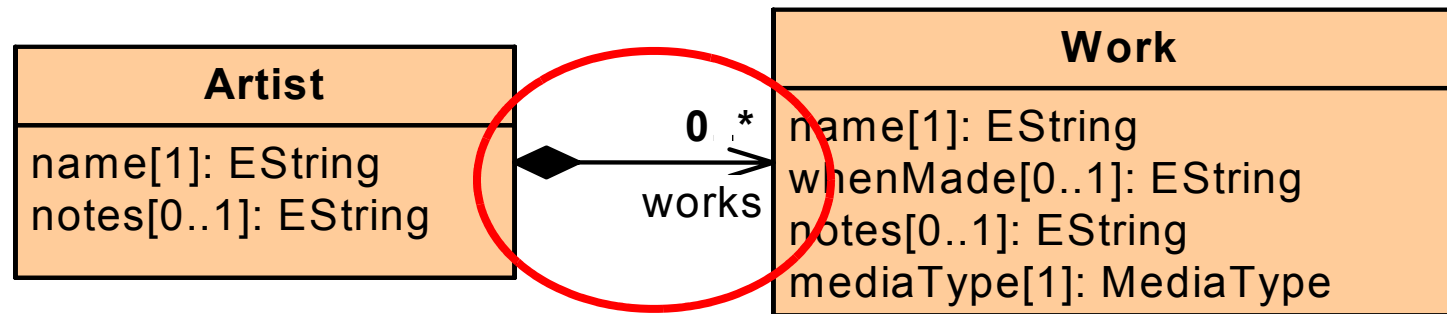
# Defining an Class

```
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>
```

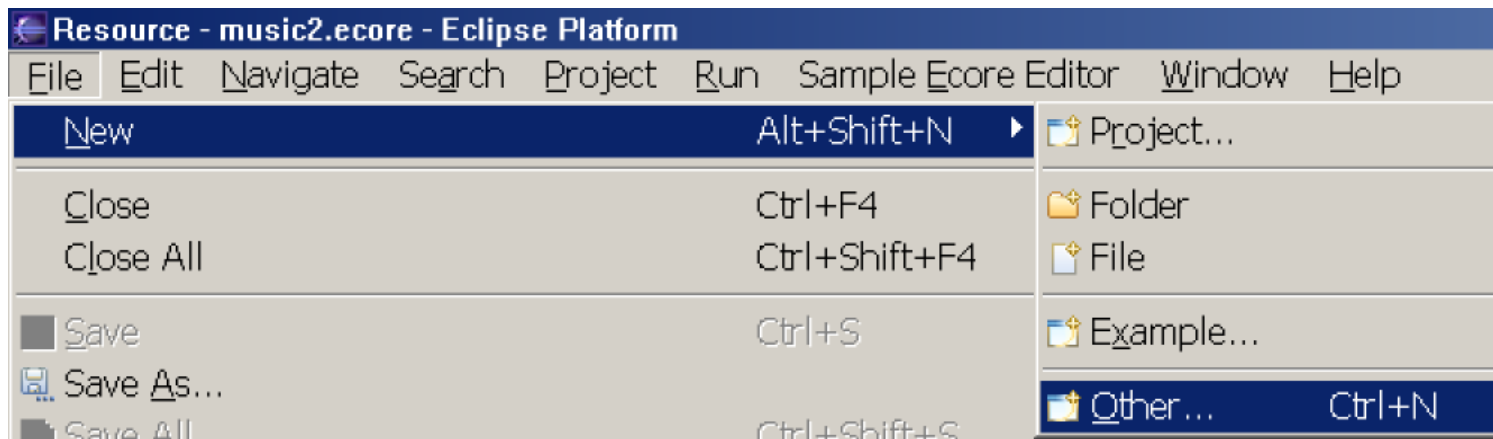
<b>&lt;&lt;enum&gt;&gt; MediaType</b>
<u>CD = 0</u>
<u>LP = 1</u>
<u>TAPE = 2</u>
<u>MP3 = 3</u>

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

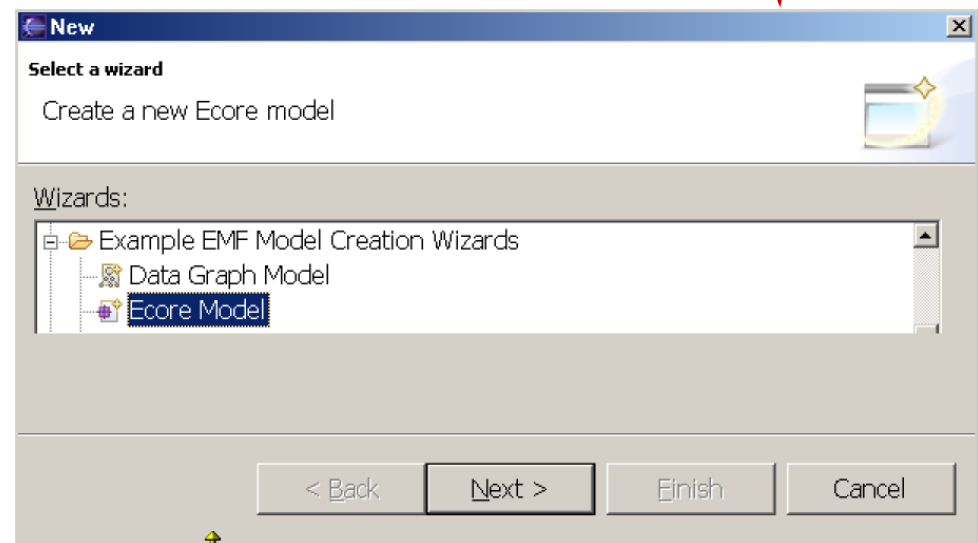
# The *ecore* Editor

- Introduction
- Defining an *ecore* model
- Defining *ecore* in XML
- ***The ecore Editor***
  - Introduction to the built-in *ecore* editor
- Defining *ecore* using UML tools
- Defining *ecore* using Java
- Defining *ecore* using XML Schema
- Defining *ecore* using Emfatic

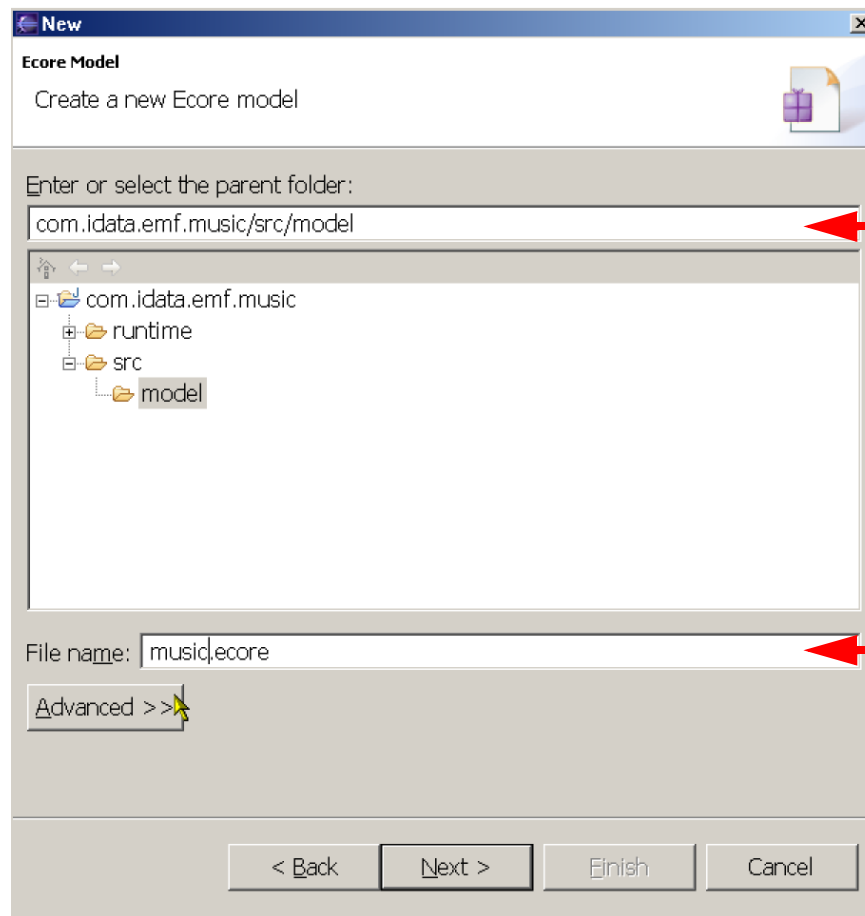
# Creating an *ecore* File



- EMF comes with a *ecore* editor
- The *ecore* editor was actually built using EMF
- It can be found in the category *Example EMF Model Creation Wizards*

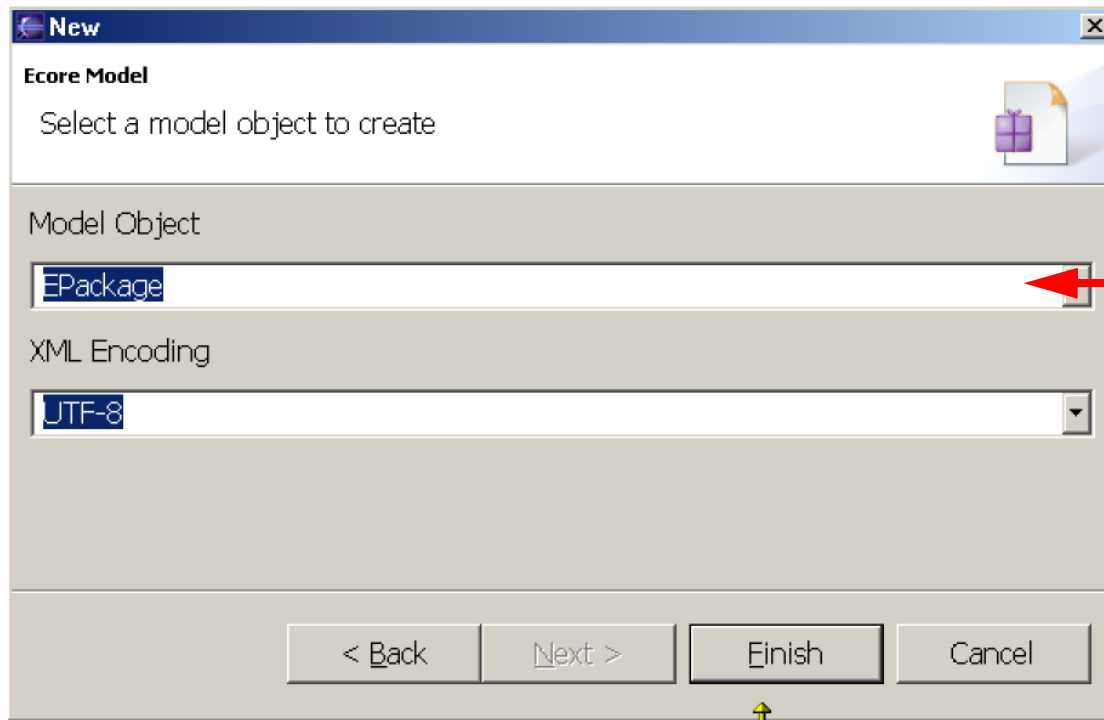


# ecore Wizard Page 2



- Next define the name and location of the model

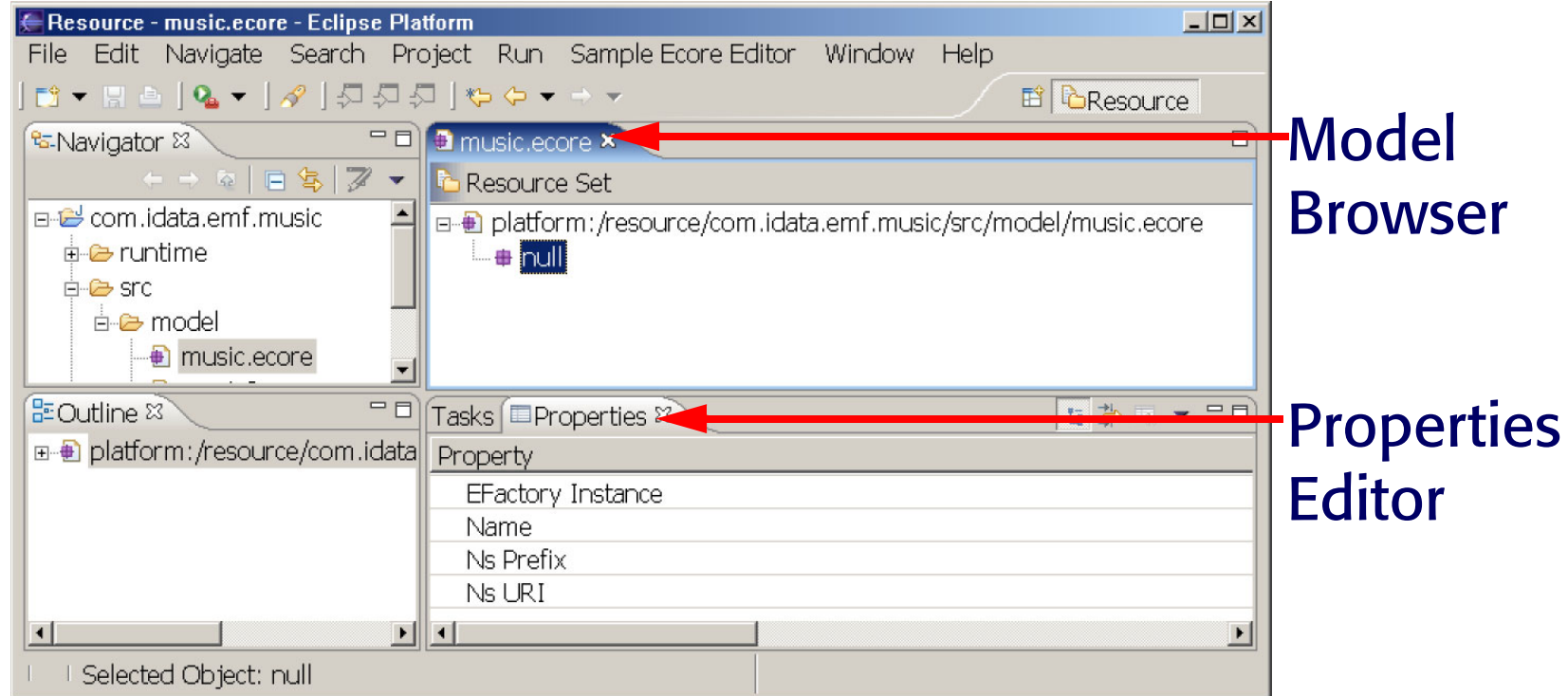
# Selecting the Root Element



Root element type

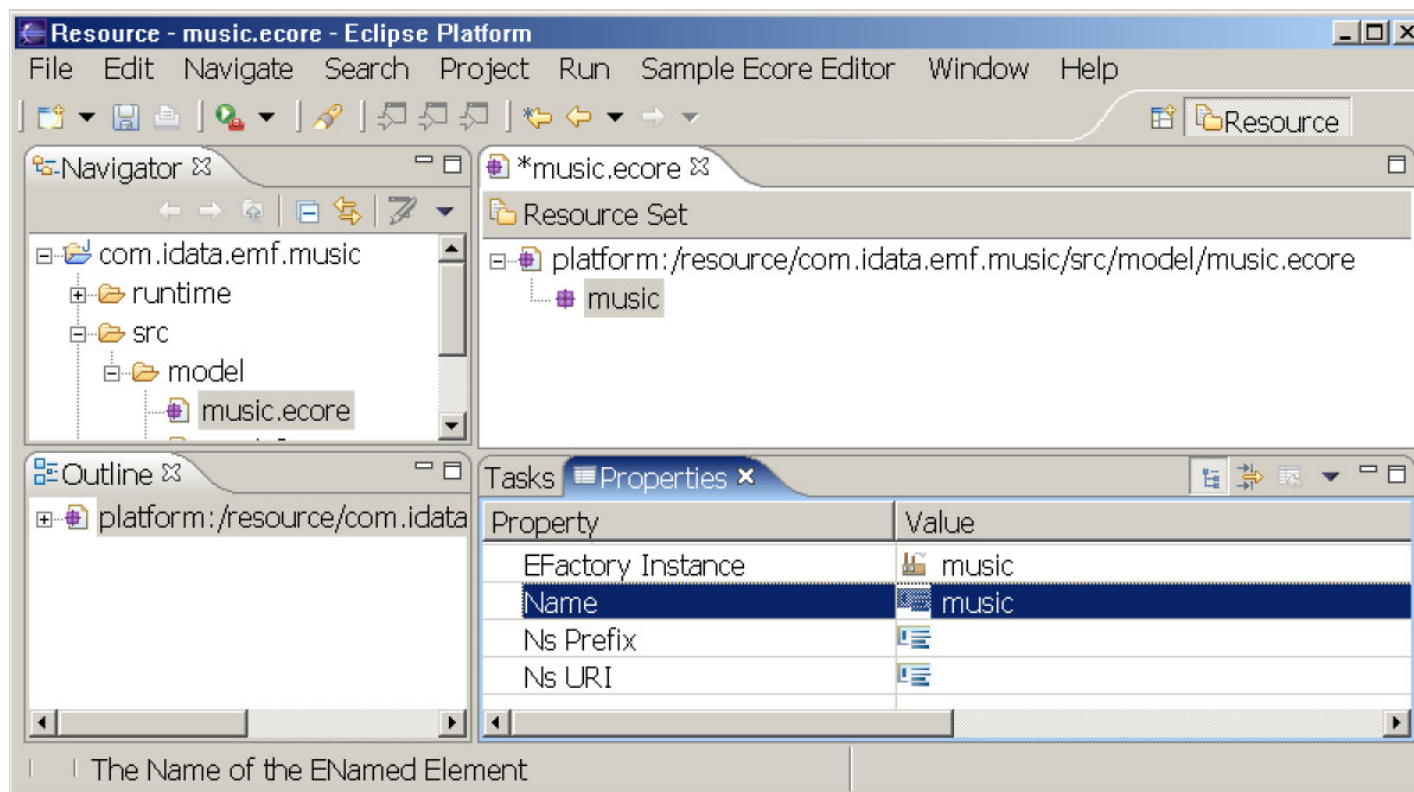
- The next step prompts for the root object type of the model
  - The root element should always be ***EPackage***
  - Why is EPackage not the default?
    - The *ecore* model was created using the ***default EMF generator*** which allows any element type to be the root

# The *ecore* Editor



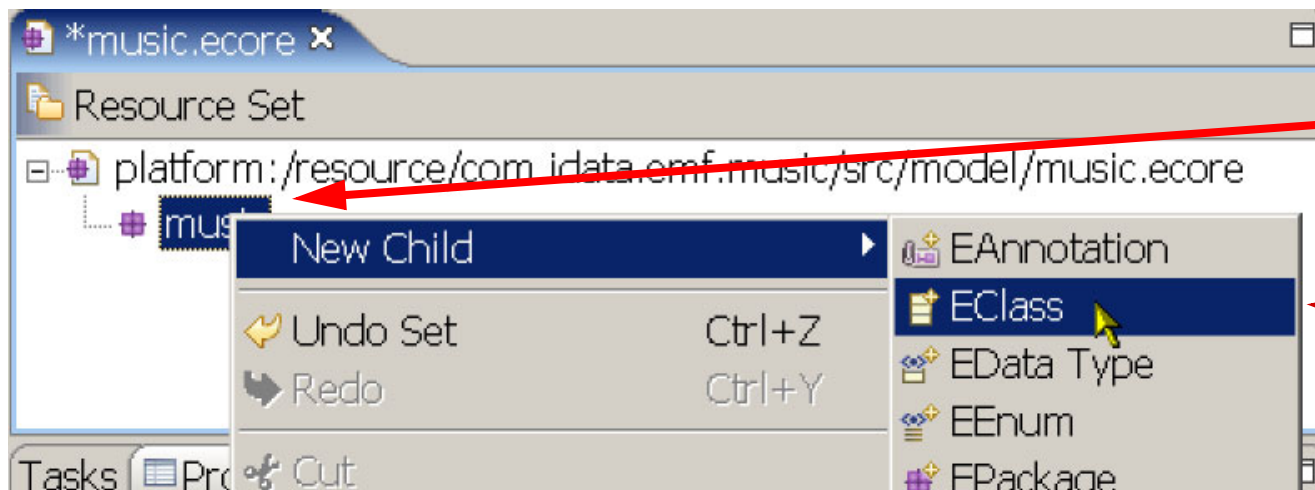
- Eclipse now opens a new *ecore* file with a custom *ecore* editor
  - The tree browser is used to browse and create new *ecore* elements
  - The property editor is used to modify *ecore* elements

# Defining the Package



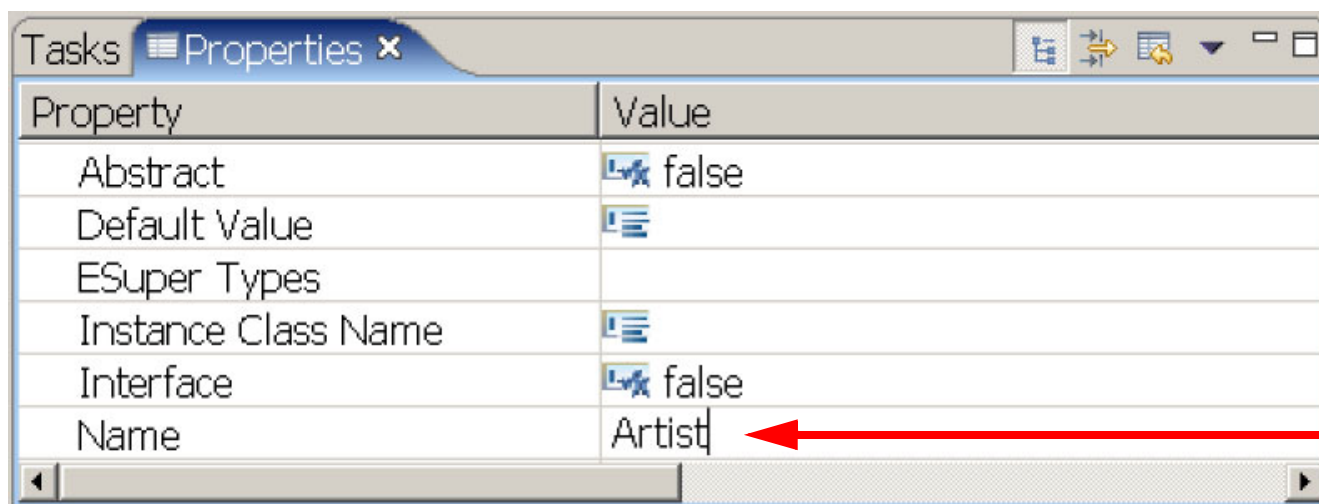
- First define the package name
  - Typically the name of the project
  - This will become the **root** name for most of the code generation

# Defining a New Class



Right-click on the package

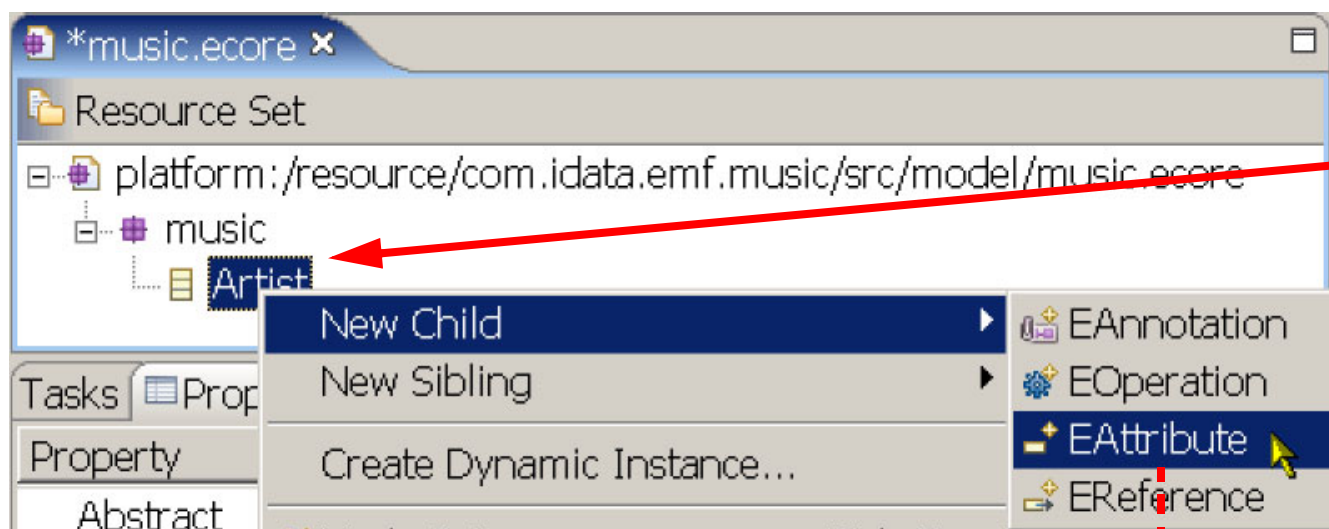
Select EClass



Enter the name

- Next we define the classes in our model

# Defining Attributes



Right-click  
on the class

Select  
EAttribute

Essential properties to define:

- ***EType***
- ***Name***
- ***Lower Bound***
- ***Upper Bound***

The screenshot shows the Eclipse Properties view for the 'Artist' class. The view is divided into two columns: 'Property' and 'Value'. The 'Name' property is selected and highlighted in blue. The 'Value' column shows the value for each property, including the EType, ID, Lower Bound, Many, Name, Ordered, Required, Transient, Unique, Unsettable, and Upper Bound.

Property	Value
EType	EString <java.lang.String>
ID	false
Lower Bound	1
Many	false
Name	name
Ordered	true
Required	true
Transient	false
Unique	true
Unsettable	false
Upper Bound	1

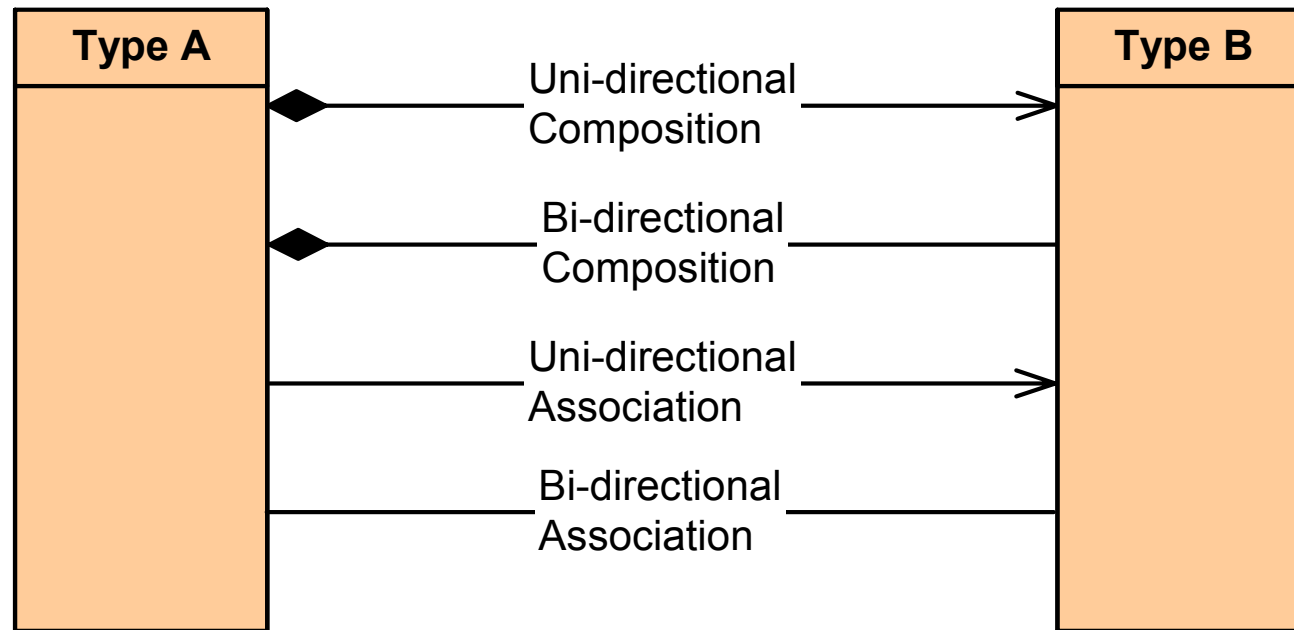
# Mapping UML to *ecore* Properties for Attributes

`attributeName [lowerBound..upperBound]: AttributeType`

Property	Value
EType	EString <java.lang.String>
ID	false
Lower Bound	1
Many	false
Name	name
Ordered	true
Required	true
Transient	false
Unique	true
Unsettable	false
Upper Bound	1

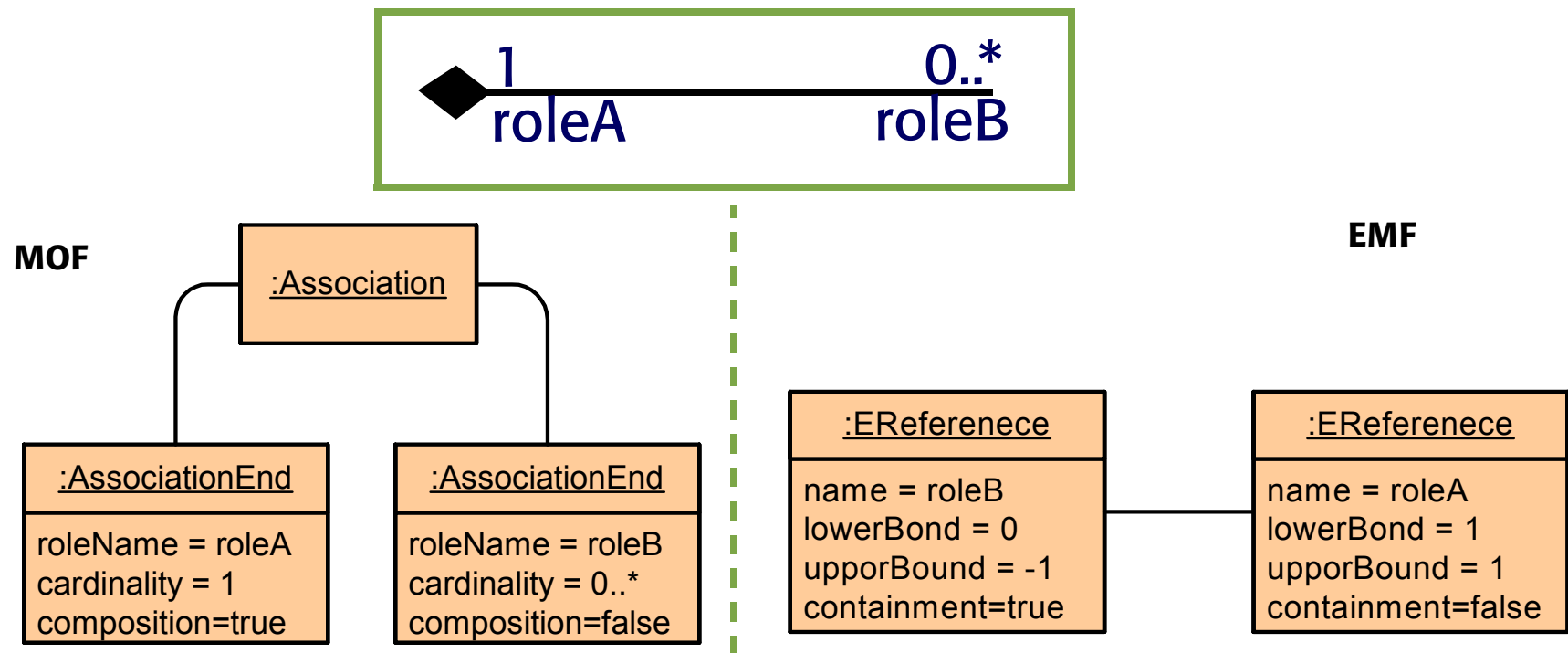
- There is an easy map from UML Attribute notation to the property editor
  - UML \*, mapped to **-1 (unbounded)** upper bounds

# Defining Associations



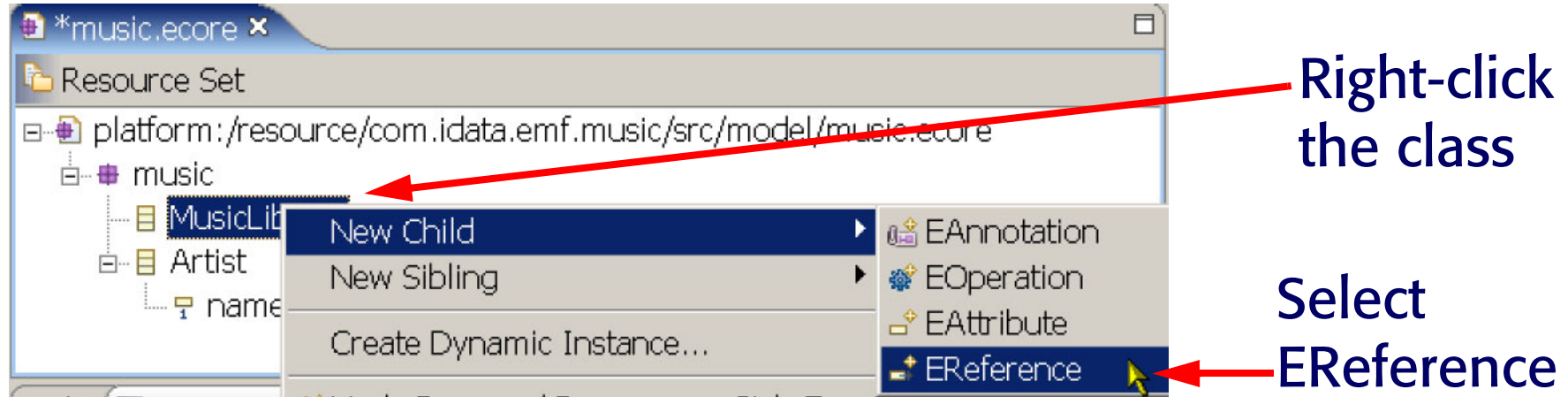
- Ecore supports the following main categories of associations
  - Directionality
    - Uni-directional
    - Bi-directional
  - Composition
    - Composition type
    - Association type

# 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 an Association



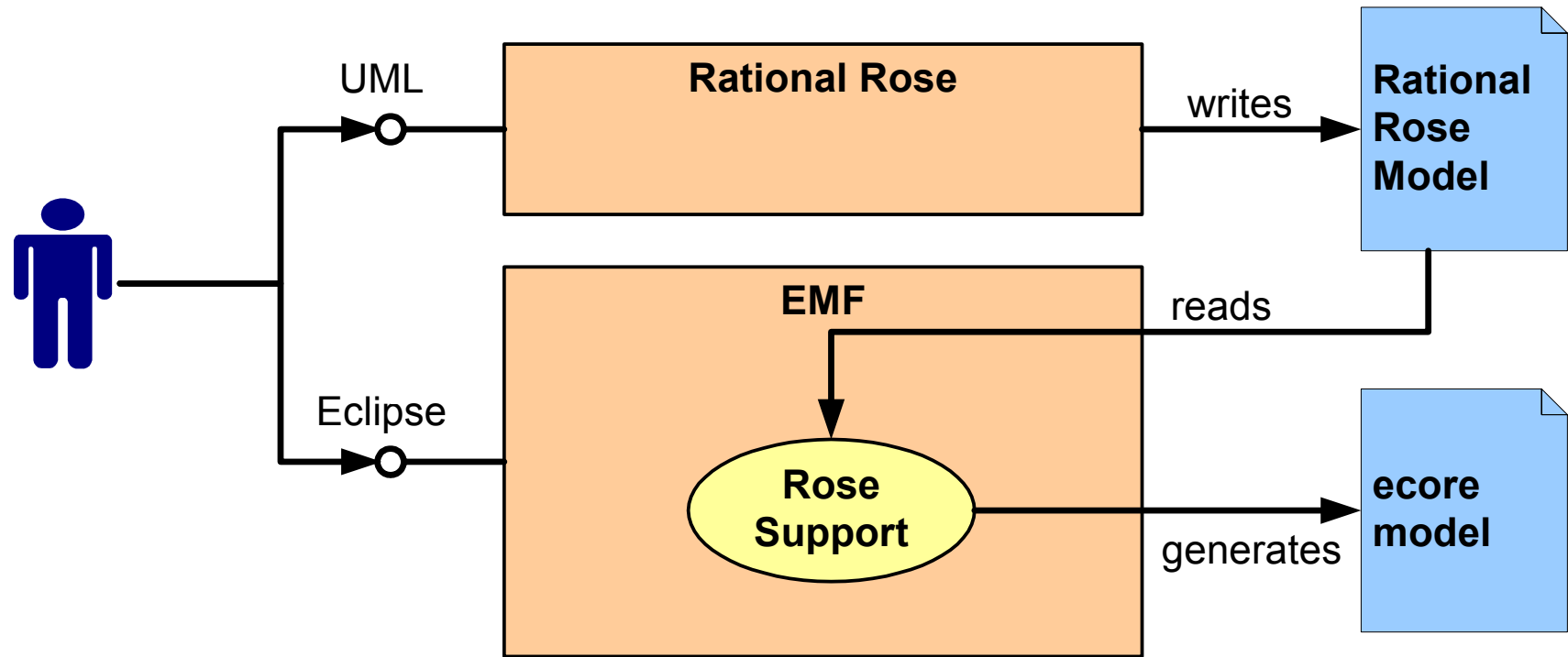
## ■ Key properties

- **Name.** The role name
- **Lower bound.** The lower bound cardinality
- **Upper bound.** The upper bound cardinality
- **EType.** The type referred to
- **Containment.** Is it a composition type?
- **EOpposite.** In a bi-directional association, what is the other end?

# Defining *ecore* using Rational Rose

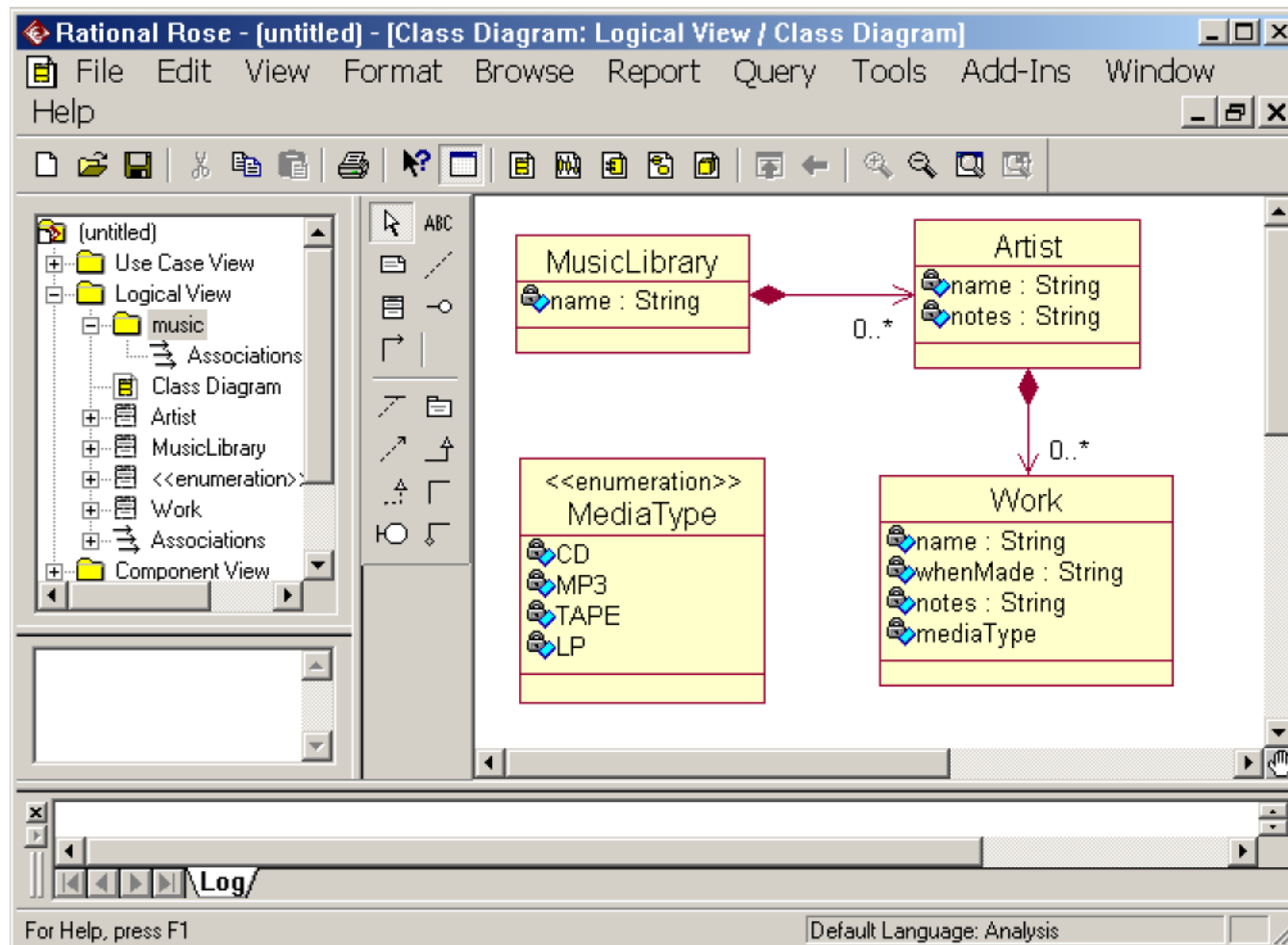
- Introduction
- Defining an *ecore* model
- Defining *ecore* in XML
- The *ecore* Editor
- ***Defining *ecore* using UML tools***
  - How to use Rational Rose and EclipseUML to build *ecore* models
- Defining *ecore* using Java
- Defining *ecore* using XML Schema
- Defining *ecore* using Emfatic

# Rational Rose



- It is possible to use Rational Rose to build *ecore* models
  - Create class diagrams in Rational Rose
  - Use ecore profile
  - Import the ecore model using the Rational Rose model file (\*.mdl)

# Create a Class Model Inside Rational Rose

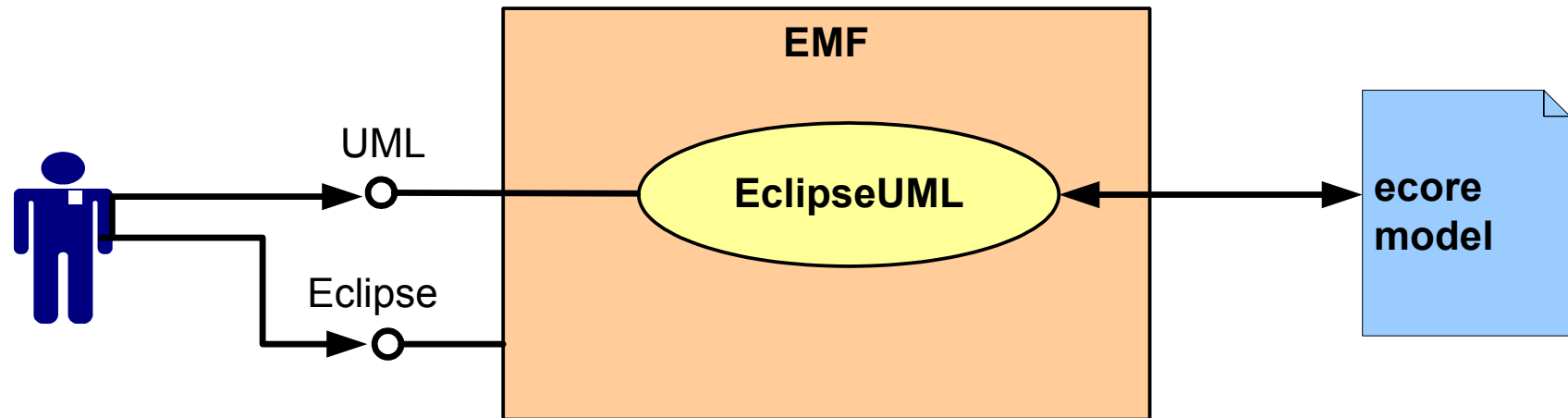


- Use either the outline view or the class diagram view

# Create an EMF Model from Rational Rose

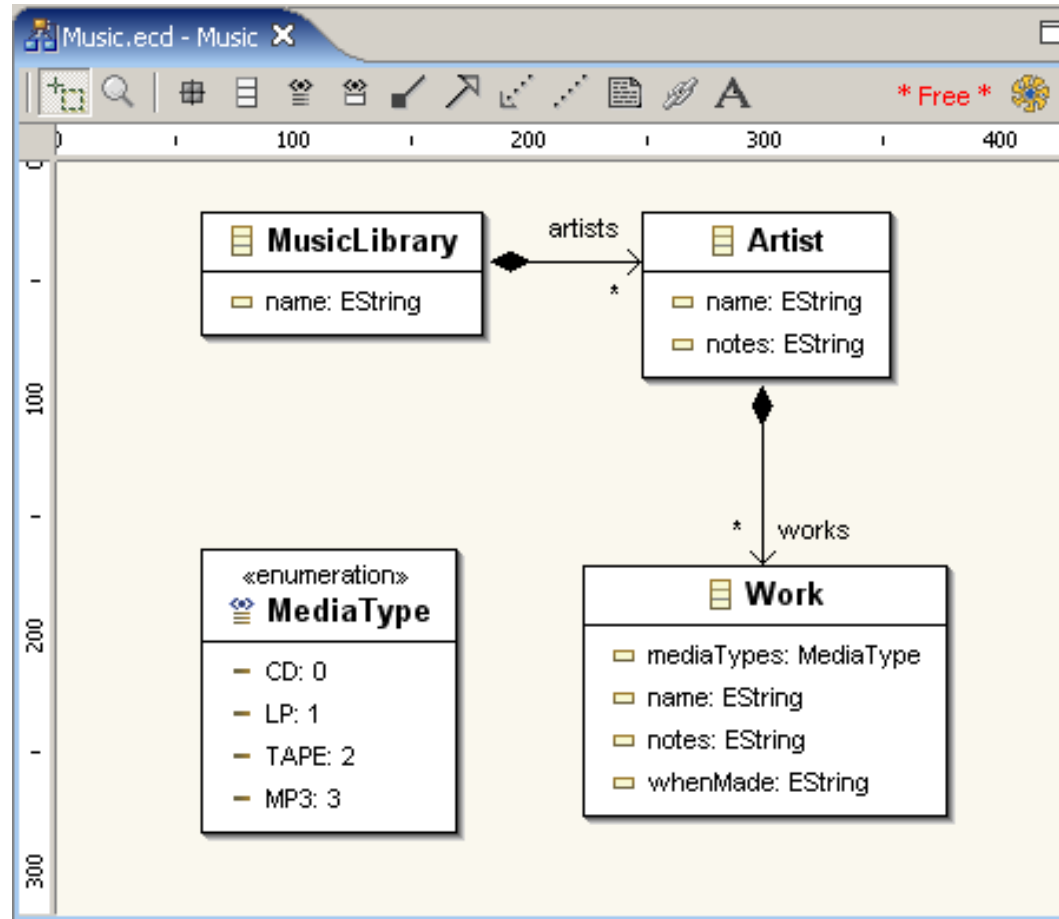
- After saving the Rational Rose model a **.mdl** is created
- The .mdl file can be used as the foundation for building an ecore model

# Defining *ecore* using EclipseUML



- Omondo EclipseUML: a UML tool that natively support EMF
  - EclipseUML is built on top of EMF
- EMF class diagram directly edits the ecore model

# Create a Class Model Inside EclipseUML



- If we change the ecore file, the diagram will be updated (and vice versa)

# Defining *ecore* using Java

- Introduction
- Defining an *ecore* model
- Defining *ecore* in XML
- The *ecore* Editor
- Defining *ecore* using UML tools
- ***Defining *ecore* using Java interfaces***
  - Annotated Java interfaces
  - Building *ecore* models from Java
- Defining *ecore* using XML Schema
- Defining *ecore* using Emfatic

# Java Interface 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 an 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

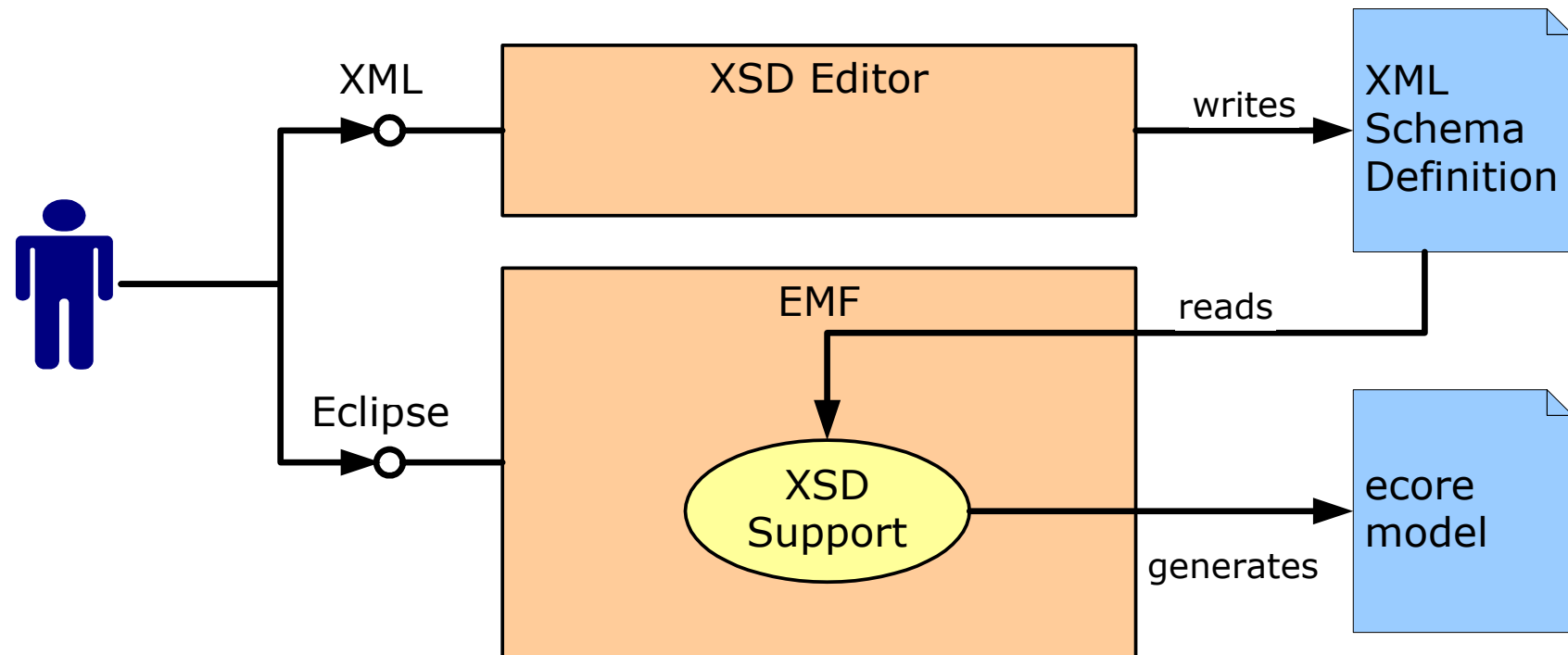
```
1 package com.idata.music;
2 /**
3  * @model
4  */
5 public final class MediaType {
6     /**
7      *@model
8      */
9     public final static int CD=0;
10    /**
11     *@model
12     */
13    public final static int MP3=1;
14    /**
15     *@model
16     */
17    public final static int TAPE=2;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

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

# Defining *ecore* using XML Schema

- Introduction
- Defining an *ecore* model
- Defining *ecore* in XML
- The *ecore* Editor
- Defining *ecore* using UML tools
- Defining *ecore* using Java interfaces
- ***Defining *ecore* using XML Schema***
  - Schema support
  - Schema/*ecore* mappings
- Defining *ecore* using Emfatic

# XSD Support



- EMF support generation of *ecore model* from ***XML Schema Definition***
- Popular in XML-focused environments
  - Quickly generate an editor for XML files

# Defining *ecore* using Emfatic

- Introduction
- Defining an *ecore* model
- Defining *ecore* in XML
- The *ecore* Editor
- Defining *ecore* using UML tools
- Defining *ecore* using Java interfaces
- Defining *ecore* using XML Schema
- ***Defining *ecore* using Emfatic***

# What is Emfatic?

- Emfatic is a Java-like language for representing ecore models
  - Developed at IBM, available at:  
*<http://alphaworks.ibm.com/tech/emfatic>*
- Emfatic code can be compiled to ecore models and vice-versa
- Emfatic is installed as a set of plugins

# Our Music Library in Emfatic

```
class MusicLibrary {  
    attr String[1] name;  
    val Artist[*] artists;  
}  
  
class Artist {  
    attr String[1] name;  
    attr String notes;  
    val work[*] works;  
}  
  
class work {  
    attr String[1] name;  
    attr String whenMade;  
    attr String notes;  
    attr MediaType[*] mediaTypes;  
}  
  
enum MediaType {  
    CD;  
    LP;  
    TAPE;  
    MP3;  
}
```

# Emfatic References

- Containment references — used in our example:

*va1* work[\*] works

- Normal references:

*ref* work[\*] works

- Bidirectional references:

*...* work[\*]#author works

- The opposite role is written after the "#"
  - In a model, there would be an association to zero or more "Works" with the role "works"; the opposite role of the association is "author"

# The Generator Model



# Introduction to the Genmodel

- ***Introduction to the genmodel***
  - The role of the genmodel
  - Relationship between ecore and genmodel
- Configuration of the genmodel

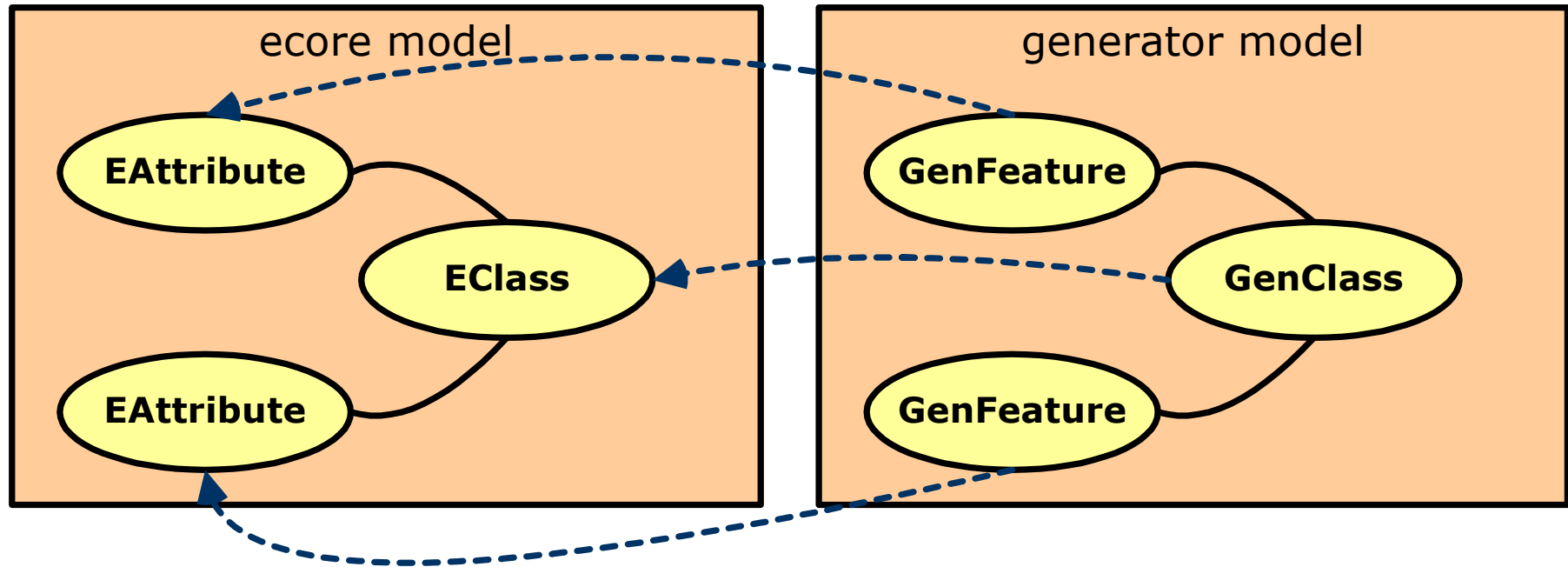
# The Role of the *genmodel*

- 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



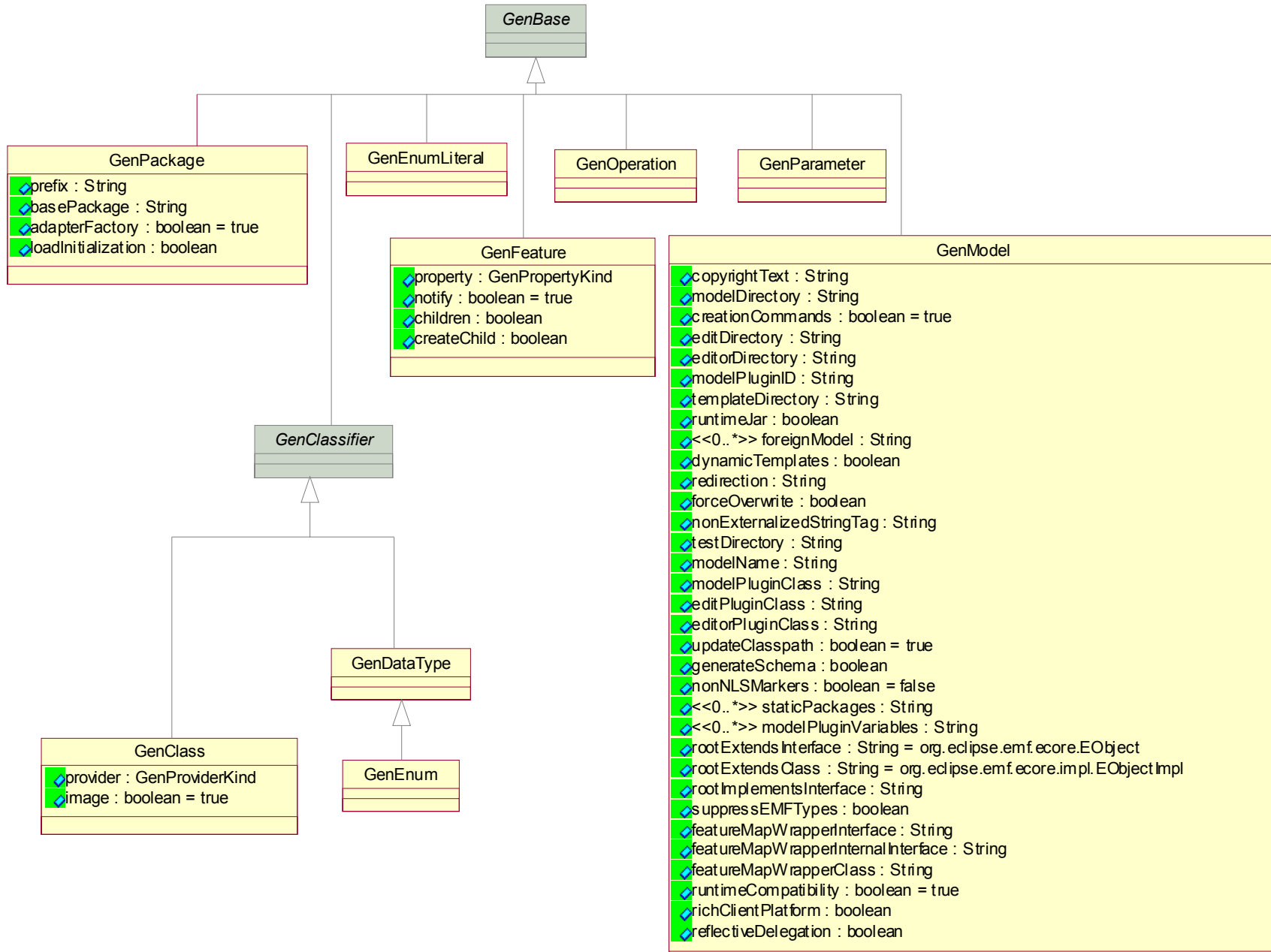
# Configuration of the *genmodel*

- Introduction to the genmodel
- ***Configuration of the genmodel***
  - What can be configured?
  - Metamodel for the genmodel
  - Configuration options and their effect (reference only)

# 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

# Meta Model for Configuration



# How to Create a genmodel?

- Use wizard included in EMF to create a new EMF Model
- Define the PIM base
  - *ecore* model?
  - XML schema definition?
  - Java interfaces?
  - Rational Rose model?
- Define what package(s) to include
- Generate the model

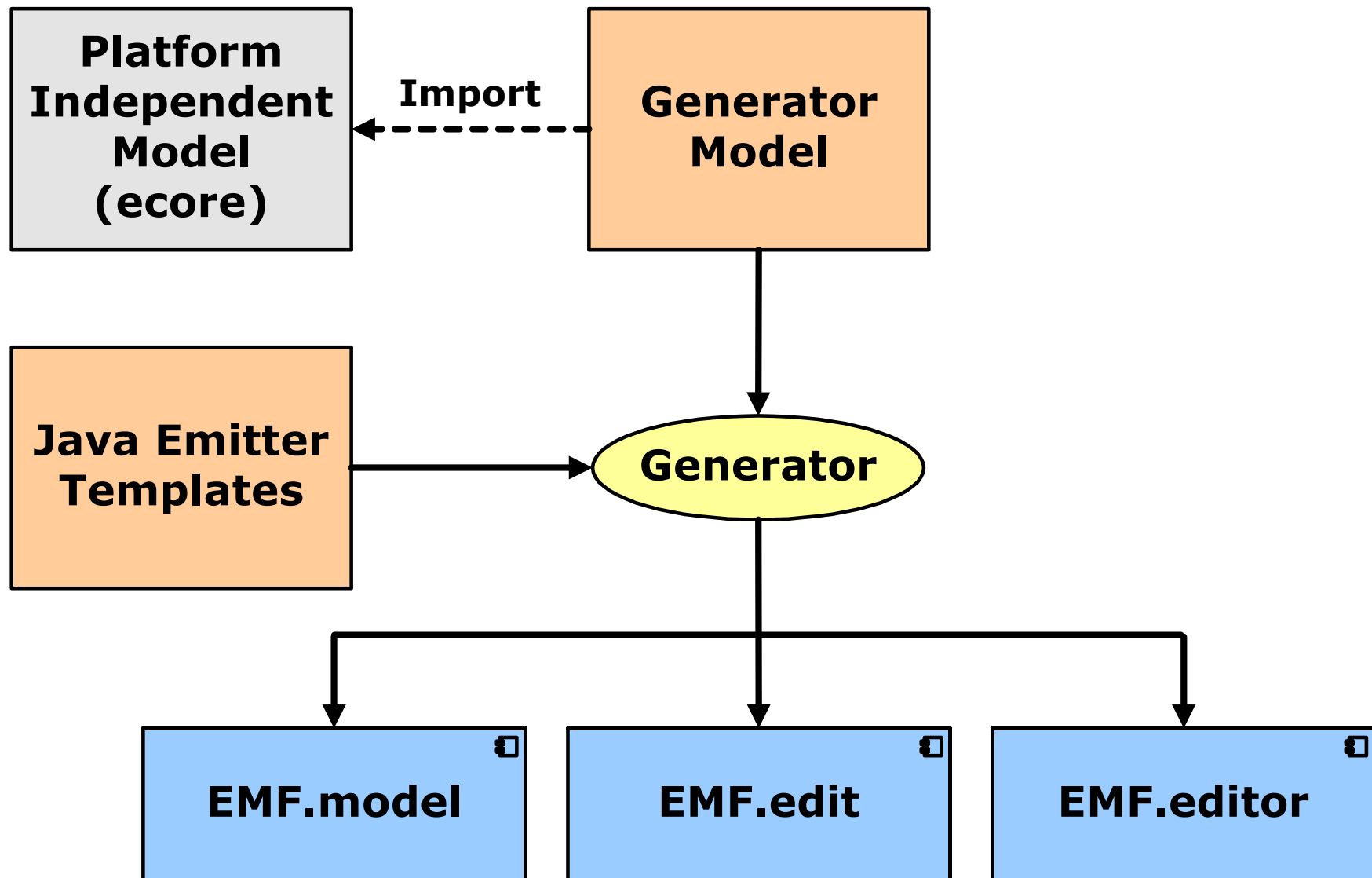
# Code Generation



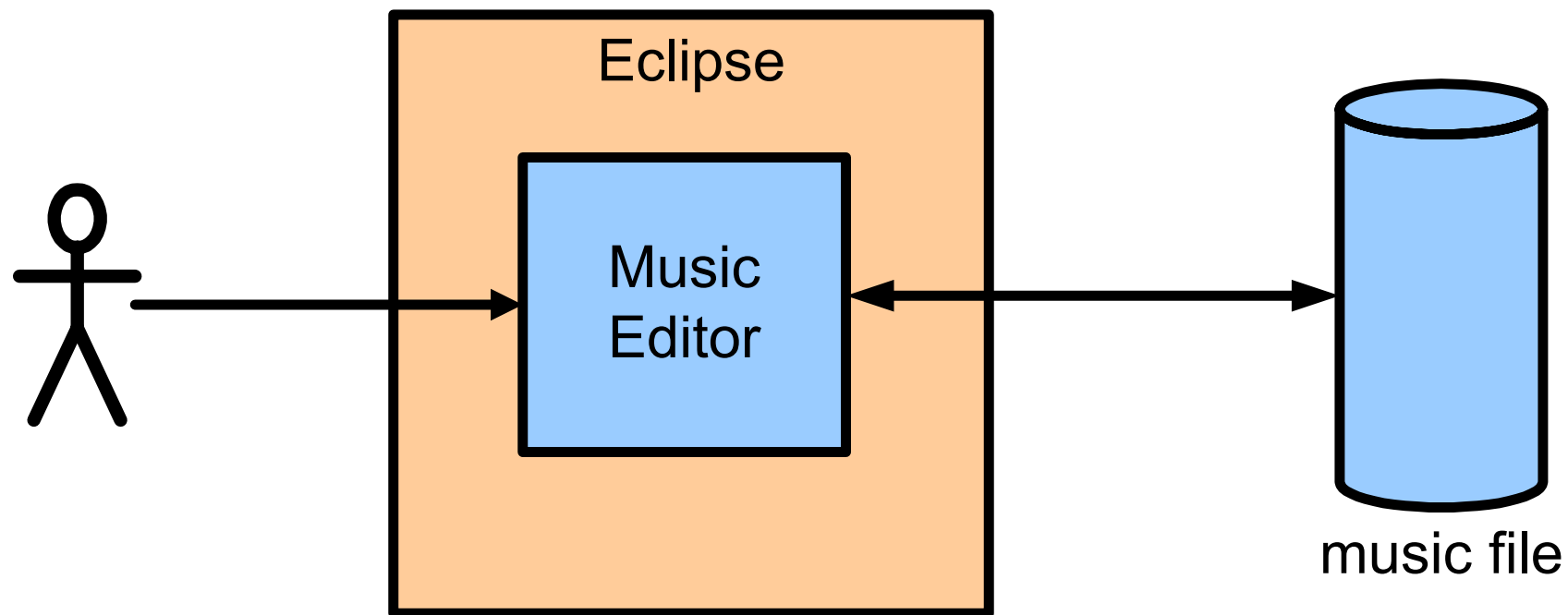
# Code Generation

- ***Code generation***
  - What is generated?
  - What is not generated?
  - Overriding generated code

# Code Generation Overview

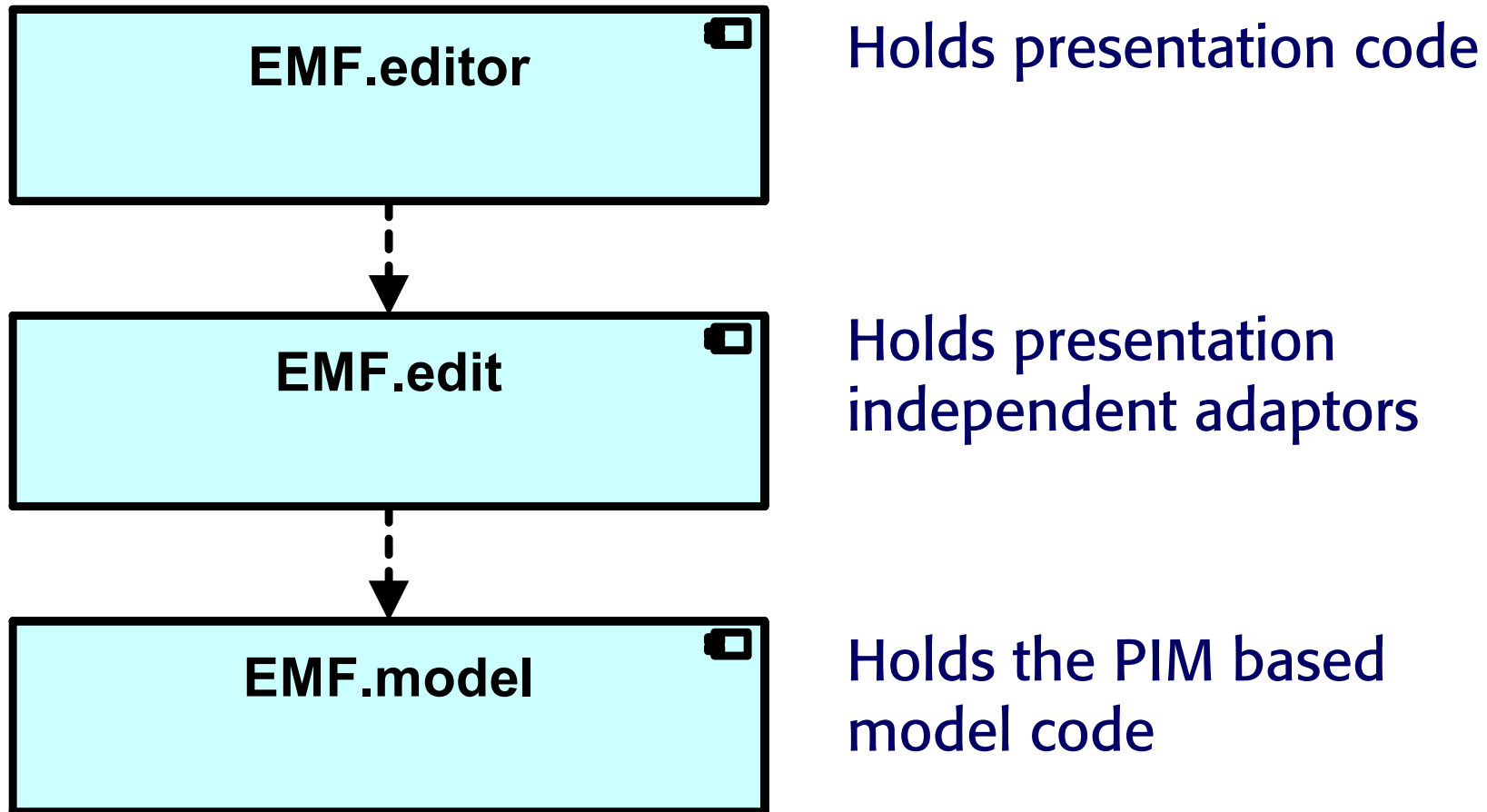


# About the Generated Implementation



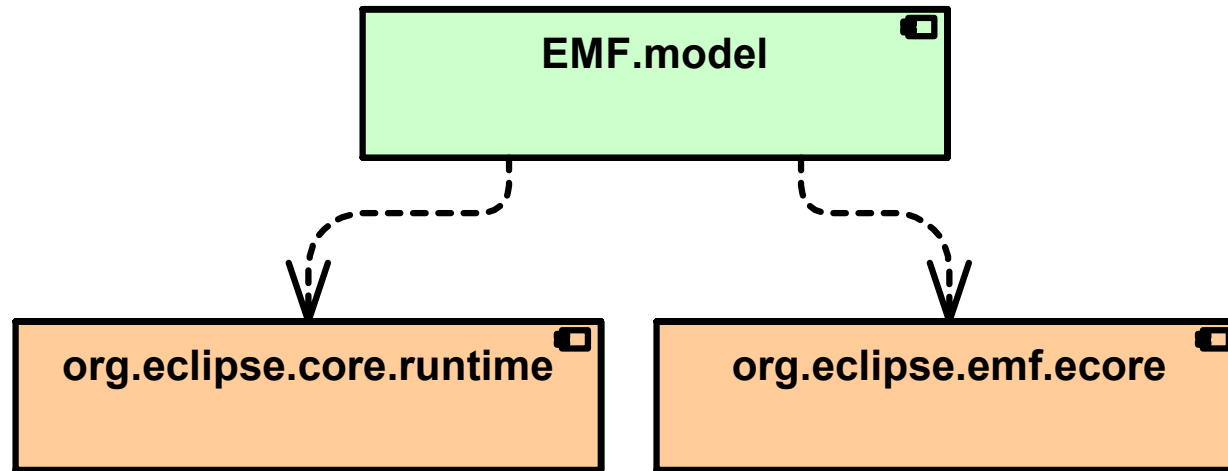
- The EMF generator creates an editor for content based on the PIM
  - Plug-in for eclipse
  - A default XML serialization

# Plug-ins Created by EMF

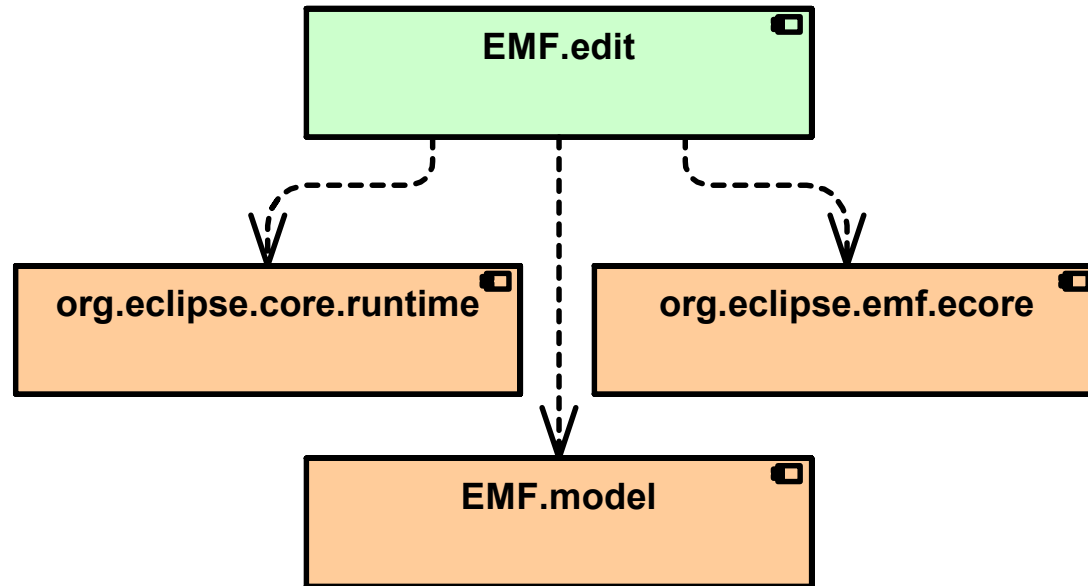


- EMF can create three different plug-ins

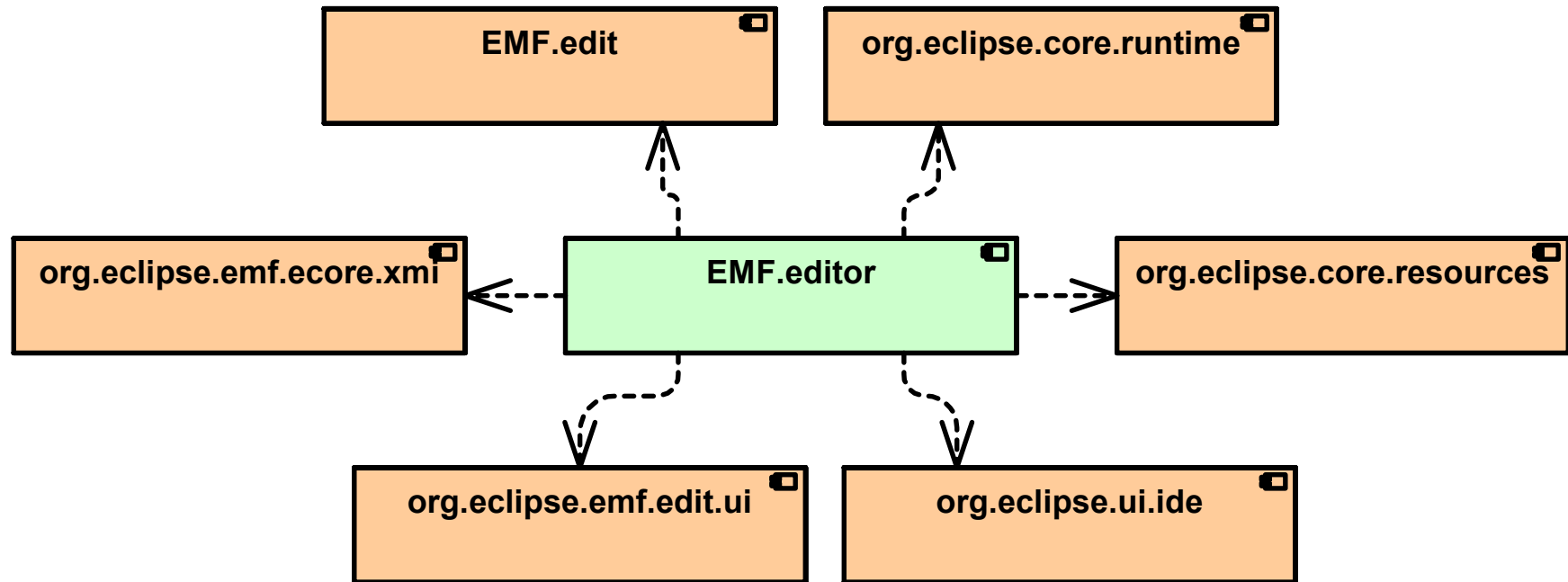
# The EMF.model



- 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



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



- 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

# What to Change?

- Your goal ought to be to leave the generated code alone
  - When you change the code, MDA is out
  - You now run the risk of increasing the *spurious complexity*
- EMF supports *mixing* of generated and manually implemented code
- The technique is the use of *javadoc markup*
- Generally, you can expect to
  - Leave the *EMF.model* plug-in *untouched*
  - *Modify* the *EMF.edit* plug-in at well defined places
  - Either *use or reimplement* the *EMF.editor* plug-in

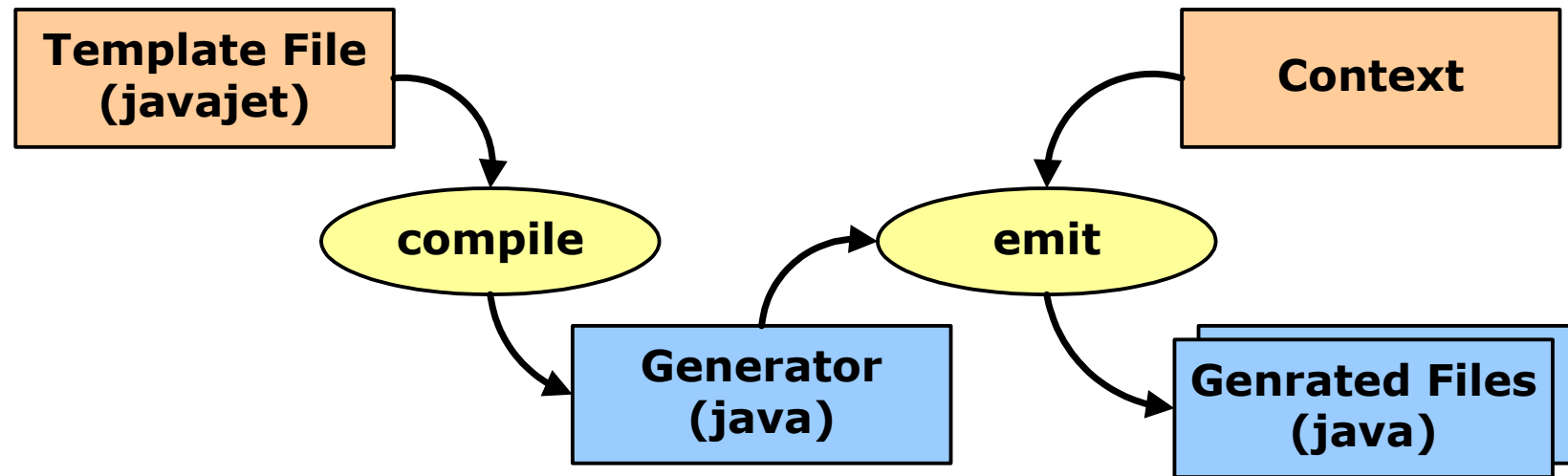
# Java Emitter Template (JET)

- Code generation
- ***Java Emitter Template (JET)***
  - What is JET?
  - How does JET work?
  - JET outside EMF
  - JET in EMF
  - Manipulating JET templates

# What is JET?

- JET is used in EMF for generating and merging java source code
- JET is a part of the EMF delivery
- Can be used in isolation
- Key terms in JET:
  - Template files
    - Contains scripts and structure that defines the mapping of EMF models to code
  - Generators
    - Code generated from templates that map the templates to strings

# How Does JET Work?



- Originally built from the source code of Tomcat's JSP compiler
- JET works similar to JSP
  - Templates are to JET what JSP's are to J2EE
  - Generators are to JET what JSP-generated Servlets are to J2EE
  - Generated source code is to JET what HTML output is to J2EE

# JET Syntax

- JET uses JSP syntax
  - Code in scriptlets
    - `<% ... %>`
  - Text outside scriptlets are generated as is
- Explicit context is different
  - JSP has implicit page, request, session, ...
  - JET implicit objects
    - ***stringBuffer*** (of type StringBuffer)
    - ***argument*** (of type Object)
- If you know JSP, you will immediately be effective in JET

# Simple Example of a javajet Template

```
<%@ jet package="com.idata.hello" class="HelloWorldTemplate" %>

class Test {
    public void main(String args[]) {
        System.out.println("Hello world");
    }
}
```

- This is a very simple java template
- Create a java
  - Name = HelloWorldTemplate
  - Package = com.idata.hello
  - Capabilities
    - Creates a HelloWorld file in the default package

# Generator File

```
package com.idata.hello;
public class HelloWorldTemplate
{
    protected final String NL =
        System.getProperties().getProperty("line.separator");

    protected final String TEXT_1 = "class Test {" + NL +
        "\tstatic public void main(String args[]) {" + NL +
        "\t\tSystem.out.println(\"Hello World\");" + NL + "\t}" +
        NL + "}";

    public String generate(Object argument) {
        StringBuffer stringBuffer = new StringBuffer();
        stringBuffer.append(TEXT_1);
        return stringBuffer.toString();
    }
}
```

- The output is a java class that can generate the hello world example

# Use of JET

```
package com.idata.hello;

public class HelloWorldGenerator {

    static public void main(String[] args) {

        HelloWorldTemplate hwt = new HelloWorldTemplate();
        System.out.println(hwt.generate(null));

    }

}
```

- We can now create a simple test client that prints out the hello world class
- Example of use:
  - `java com.idata.hello.HelloWorldGenerator > HelloWorld.java`

# Use of JET in EMF

- EMF uses JET to generate code from the models
- The EMF javajet templates can be found at:
  - ...\`org.eclipse.emf.codegen.ecore_2.0.1\templates\model`
  - ...\`org.eclipse.emf.codegen.ecore_2.0.1\templates\edit`
  - ...\`org.eclipse.emf.codegen.ecore_2.0.1\templates\editor`
- The implicit ***argument*** value refers to a GenClass
  - Early in the javajet templates you'll find the cast of ***argument***  
  
`<% GenClass genClass = (GenClass)argument; ...%>`
  - From the instance of GenClass we can navigate to any.ecore or gen feature

# EMF.model



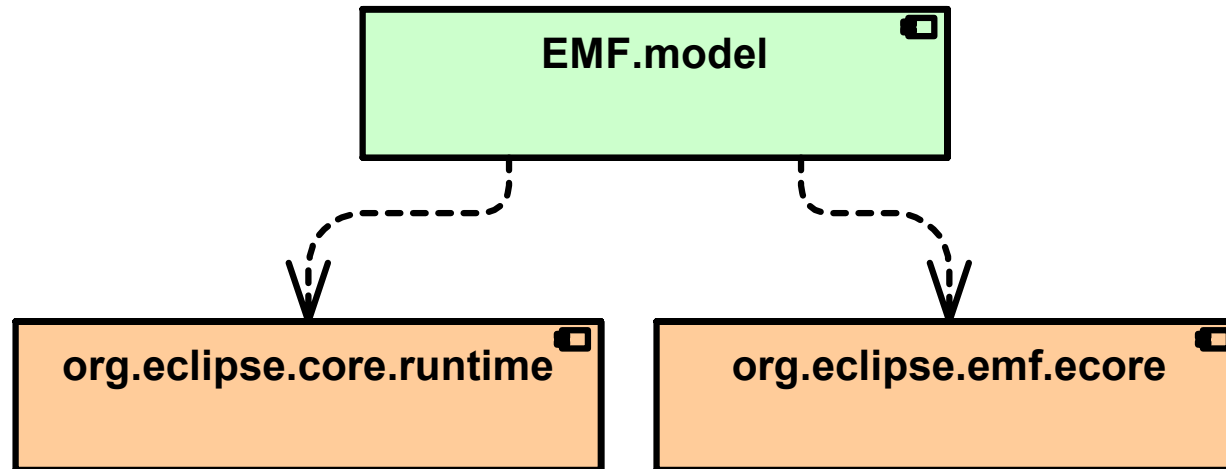
# Introduction to the EMF.model

- ***Introduction to the EMF.model***
  - What is the EMF.model?
  - EMF.model dependencies
- Anatomy of the EMF.model

# What is the 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

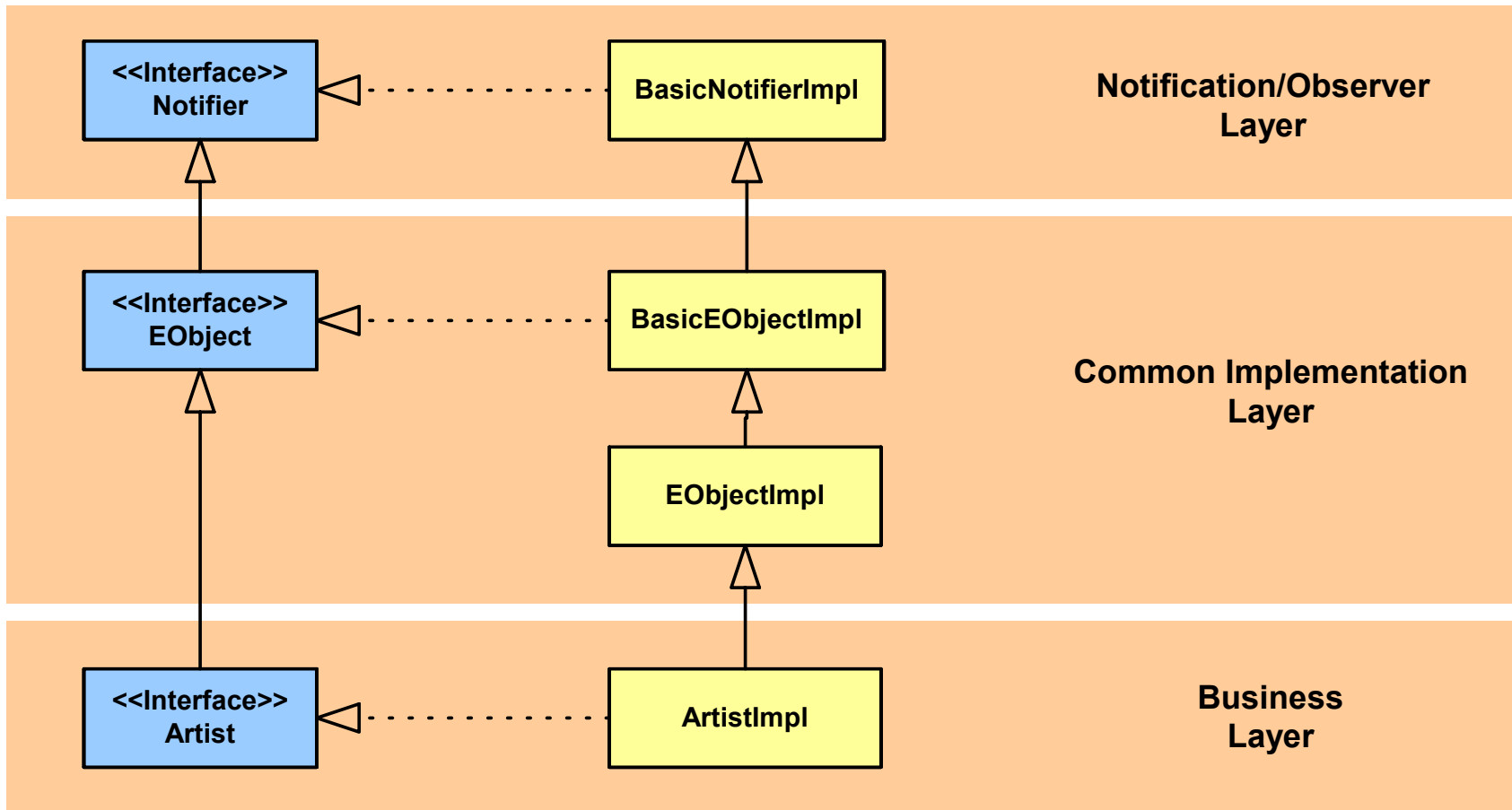


- The EMF.model depends on only two plug-ins
  - org.eclipse.core.runtime
  - org.eclipse.emf.ecore
- It is possible to setup the genmodel options such that the EMF.model can run outside Eclipse

# The Anatomy of the EMF.model

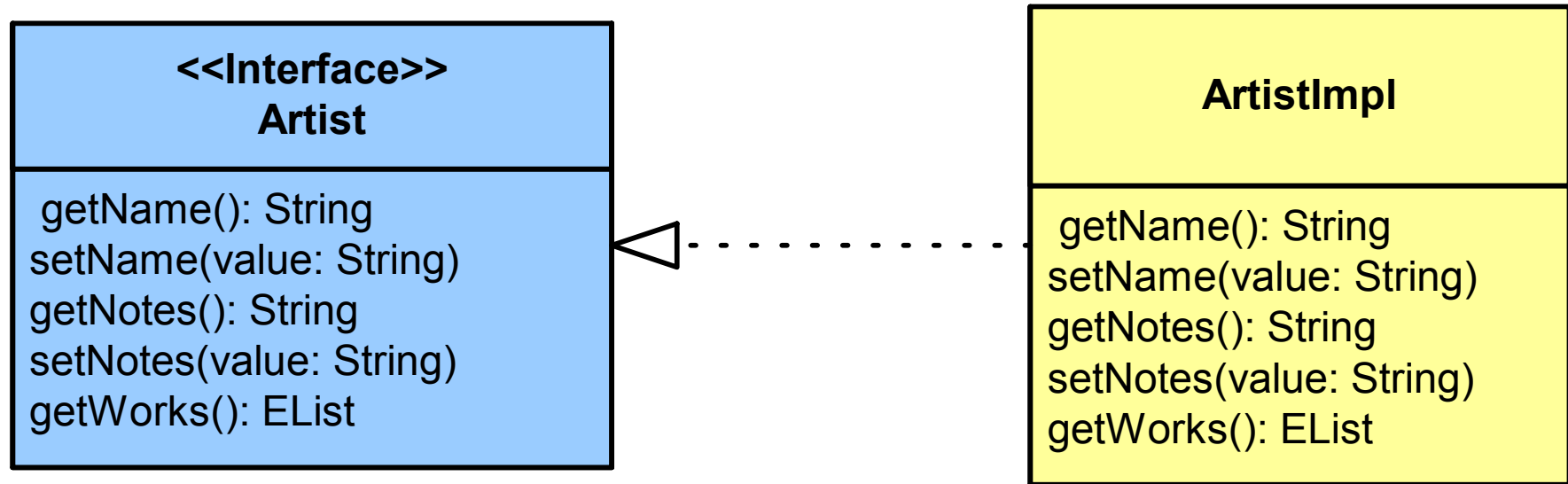
- Introduction to the EMF.model
- ***Anatomy of the EMF.model***
  - Implementation of EClass
  - Implementation of EAttribute
  - Implementation of EReference
  - Implementation of EOperation
  - Concept of notification

# 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

## ArtistImpl

```
# NAME_EDEFAULT : String = null
# name : String = NAME_EDEFAULT
# NOTES_EDEFAULT : String = null
# notes : String = NOTES_EDEFAULT

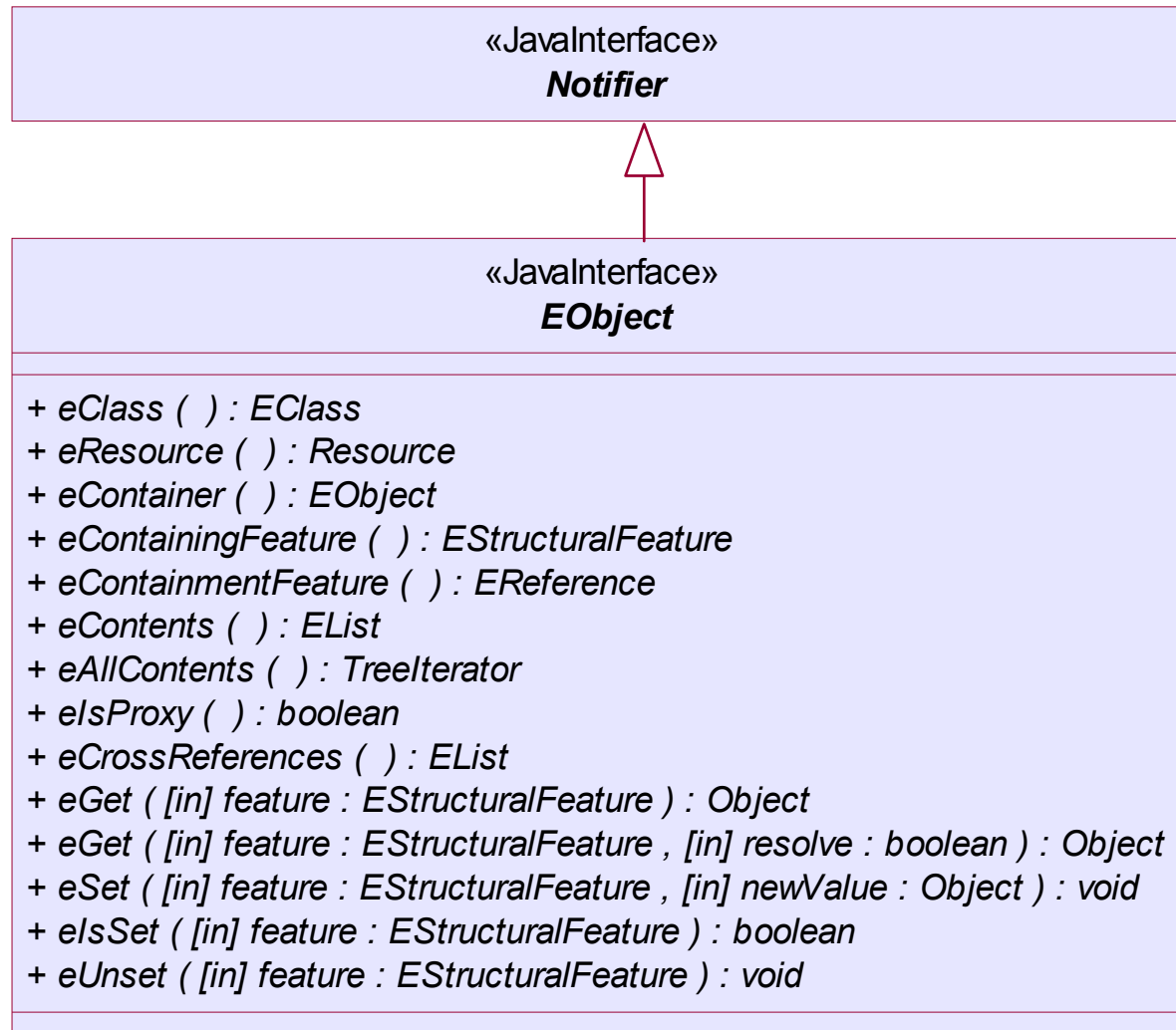
# ArtistImpl ( )
# eStaticClass ( ) : EClass
+ eInverseRemove ( [in] otherEnd : InternalEObject , [in] featureID : int , [in] baseClass : Class , [in] msgs : NotificationChain ) : NotificationChain
+ eGet ( [in] eFeature : EStructuralFeature , [in] resolve : boolean ) : Object
+ eSet ( [in] eFeature : EStructuralFeature , [in] newValue : Object ) : void
+ eUnset ( [in] eFeature : EStructuralFeature ) : void
+ elsSet ( [in] eFeature : EStructuralFeature ) : boolean
+ toString ( ) : String
```

- 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

# Understanding the ecore Framework

- The ecore runtime framework is huge
- We'll only scratch the surface in this course
- You typically only need to understand a small subset of the implementation
  - The ***EObject*** interface
  - The support ***reflection***
  - The ***notification*** feature
- The ***framework behavior*** is picked up by the generated business object extending ***EObjectImpl***

# EObject Interface



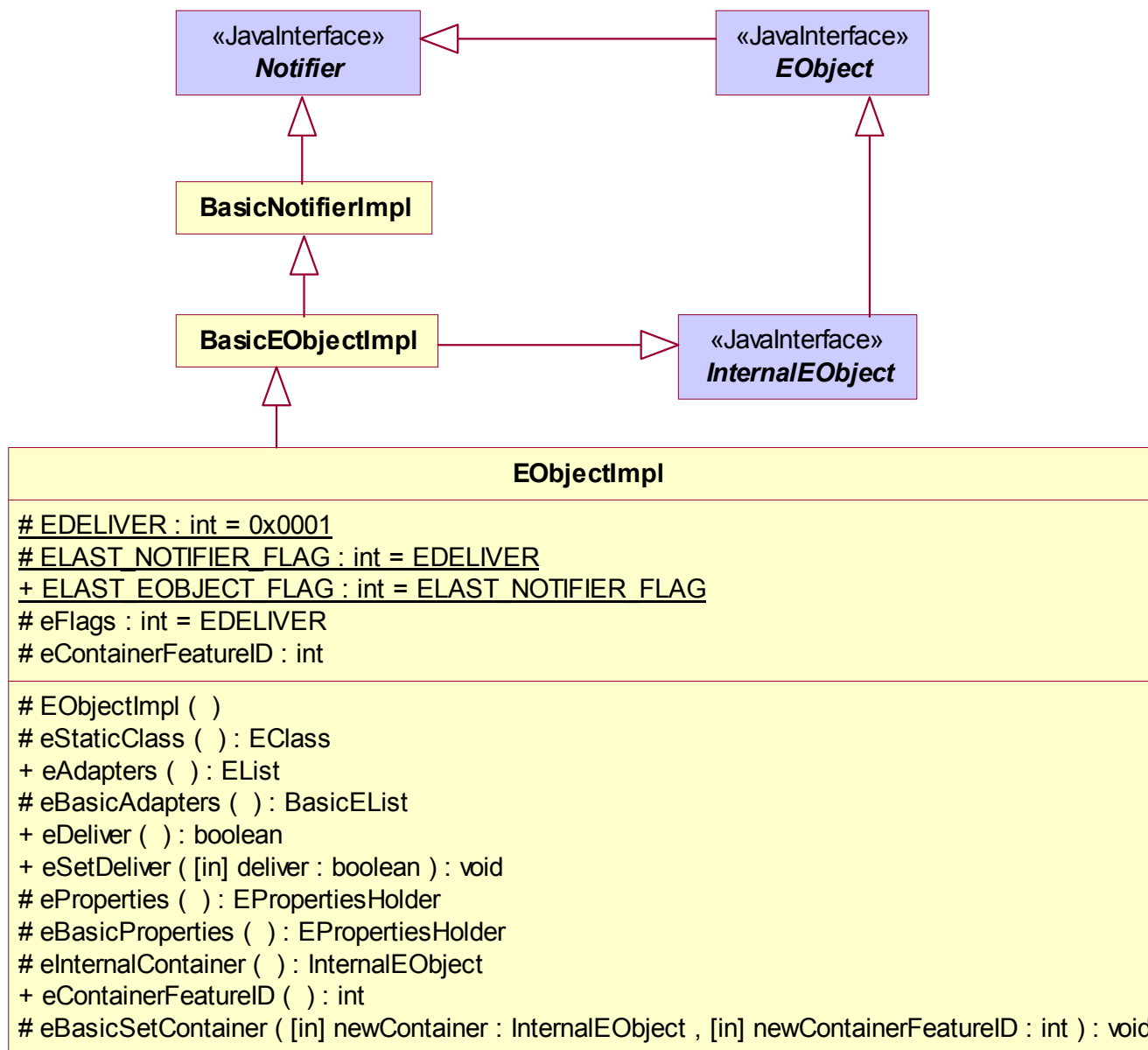
- All business object interfaces extends EObject

- A commonly used method is the ***EClass eClass();***
- Every business object can retrieve a representation of its original ecore class
- Similar to ***Java***
  - Every java object can retrieve its class representation ***Class getClass();***
  - Java reflection

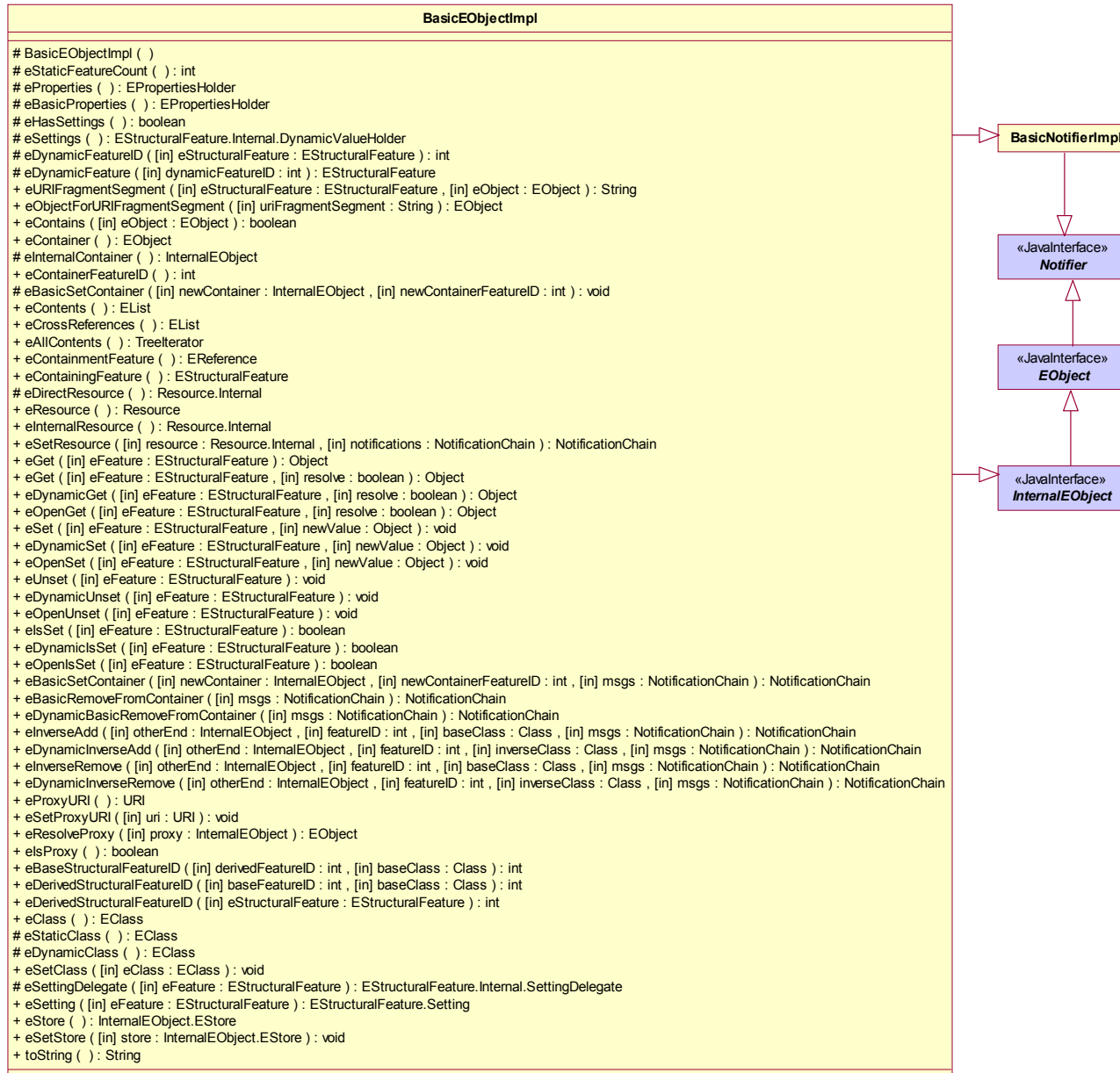
# EObject::eResource()

- A business object may be associated with a particular resource  
*Resource eResource();*
- The resource represents the persistent storage for the object

# EObjectImpl Class Diagram



# BasicEObjectImpl Class Diagram



# 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

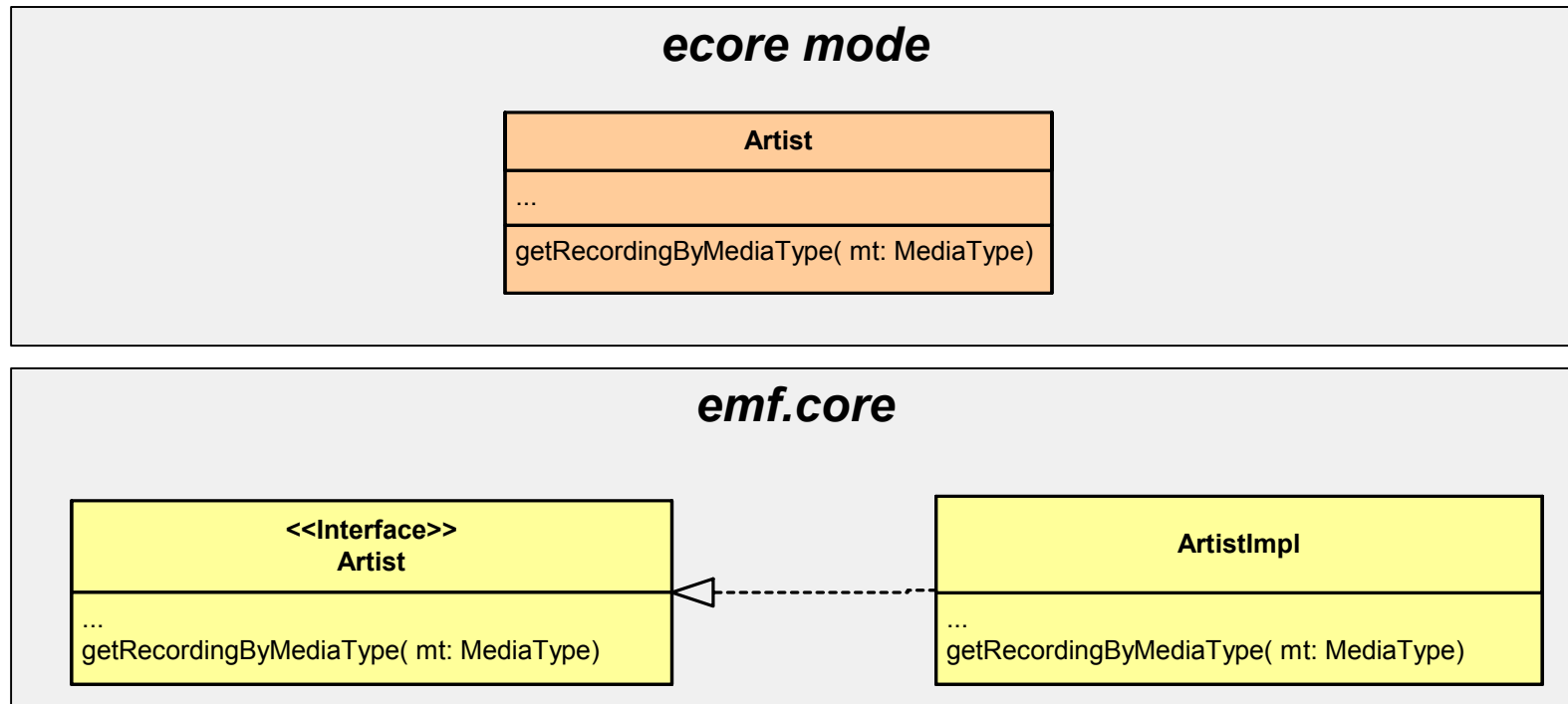
- It is possible to define operations in the ecore model
- There is no support for defining the semantic of the operations in ecore
- The idea is to:
  - Declare the business operations in the ecore model
    - Name of the operation
    - Return value type
    - Parameters
  - Implement the semantic 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

# Implementation of EOperation



- If we define an operation in our ecore model
  - The operation is added to the business interface
  - A dummy implementation of the method is generated in the business implementation

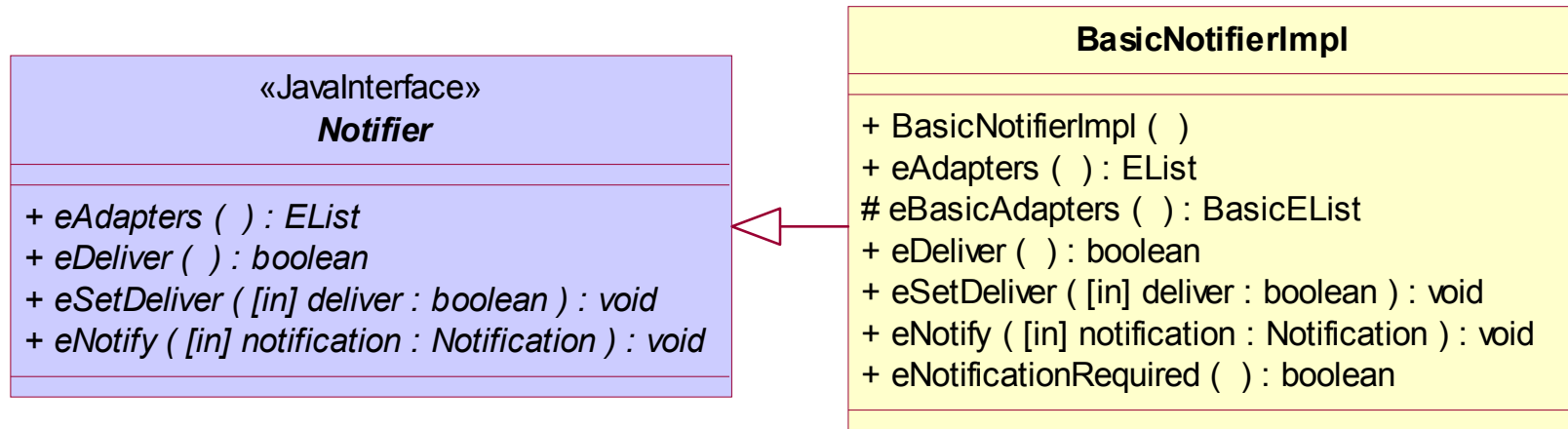
# Defining an Implementation for EOperations

```
/**
 * @generated
 */
EList getWorkByMediatype( MediaType mt ) {
    throw UnimplementedException();
}
```

- We can now take over the generated implementation and do something useful

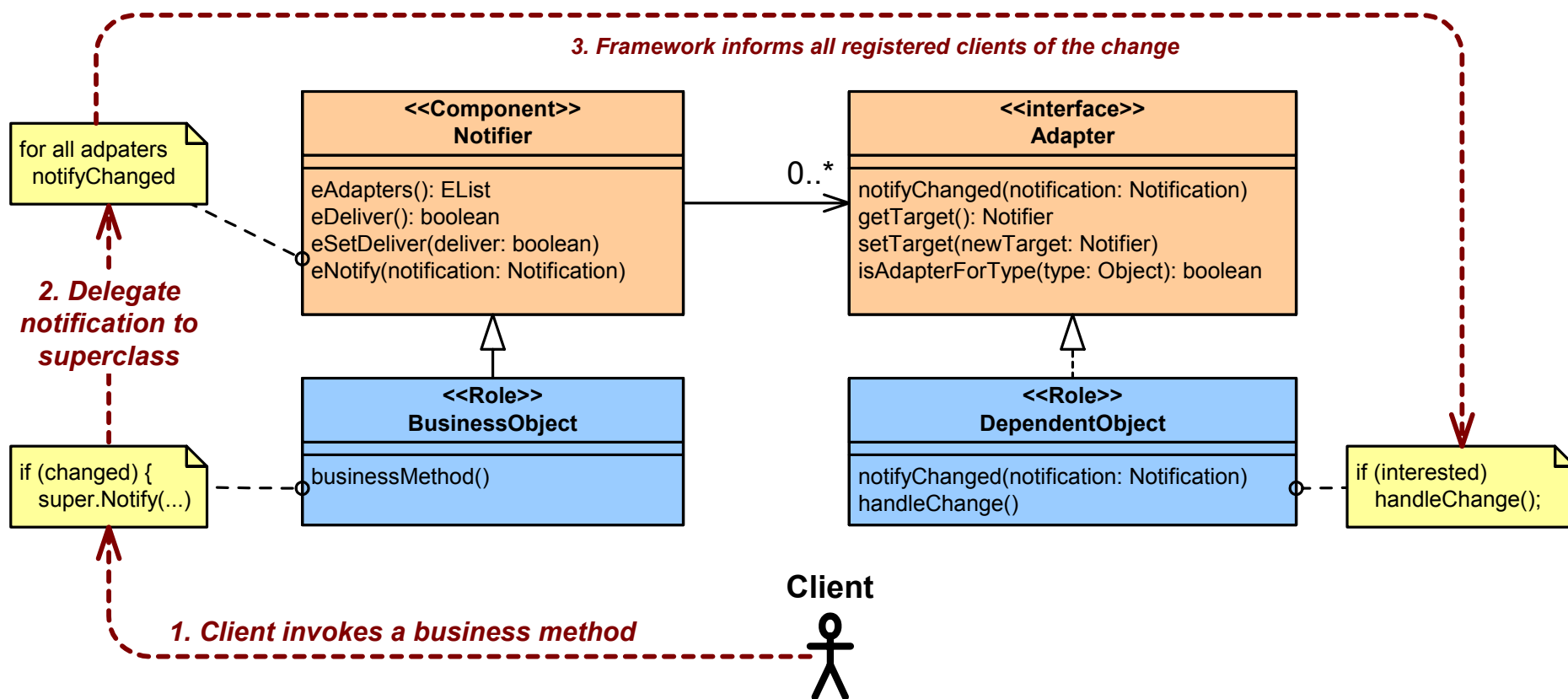
```
/**
 * @generated NOT
 */
EList getWorkByMediatype( MediaType mt ) {
    throw UnimplementedException();
    EList collection = ...;
    Iterator i = this.getworks().iterator();
    while (i.hasNext() ) {
        ...
    }
    return collection;
}
```

# Notification Implementation



- Each model object support event propagation on change
  - Using the **Observer Pattern**
    - GOF Observable ~ **Notifier**
    - GOF Observer ~ **Adapter**
  - Use of event objects during notification
    - Implementation of **Notification**
  - The **genmodel** allows you to configure what change causes notification

# Logical Class Diagram for Notification



- The details of the implementation is not important
  - You typically never change the implementation of the notification behavior

# EMF.edit



# Introduction to EMF.edit

- ***Introduction to EMF.edit***
  - The role of EMF.edit
  - What is generated?
- EMF.edit and design patterns
- Modifying presentation behavior
- Modifying commands

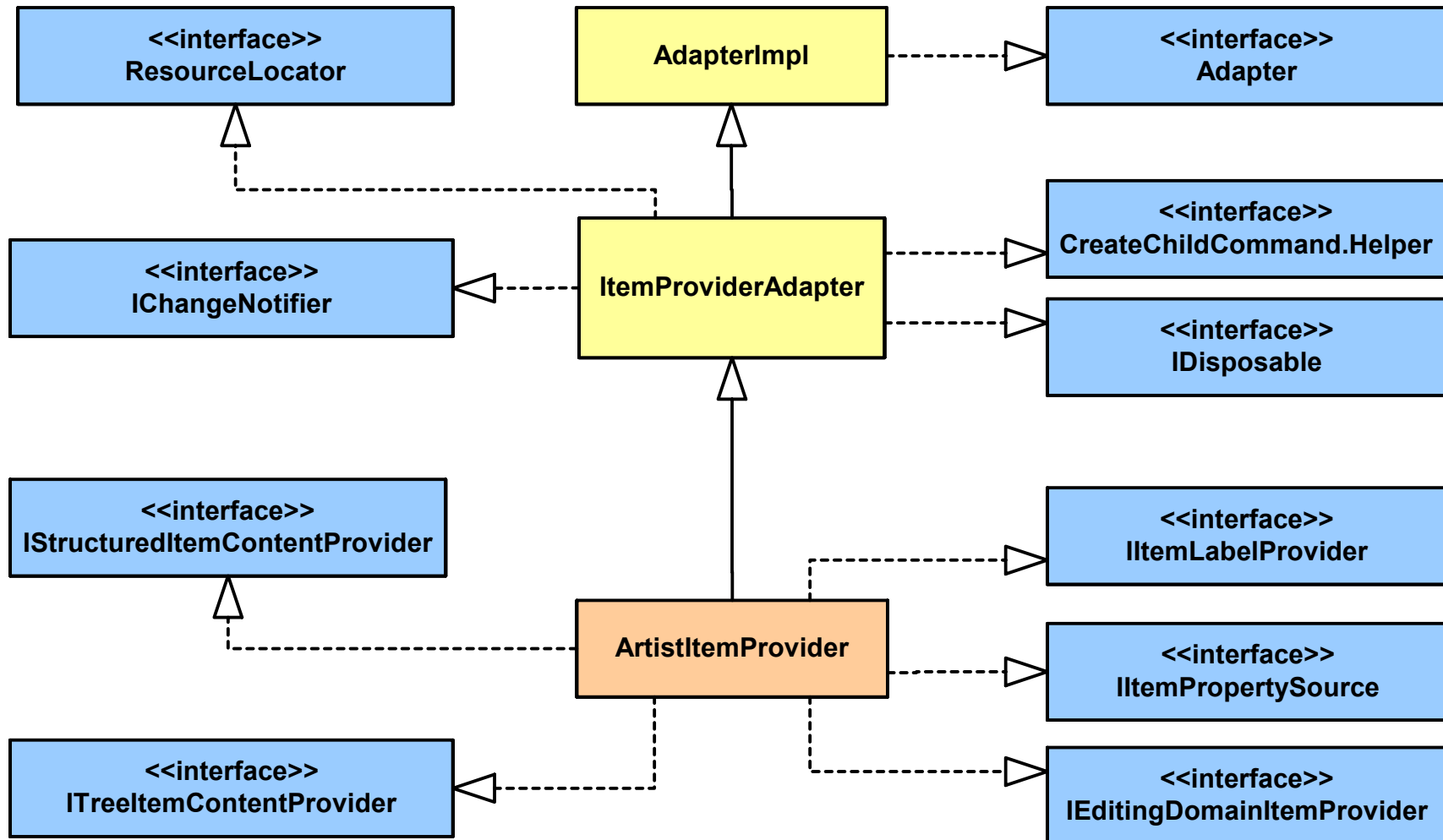
# Role of EMF.edit

- 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

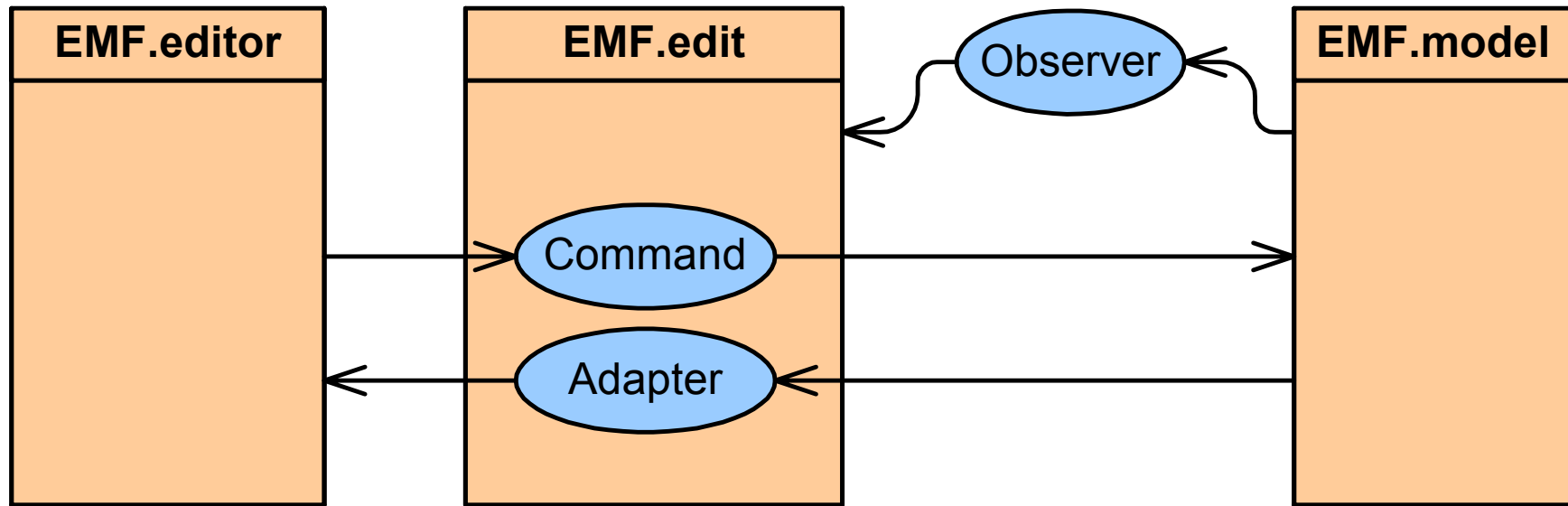
# Framework Generator Structure



# EMF.edit and Design Patterns

- Introduction to EMF.edit
- ***EMF.edit and design patterns***
  - Use of observer pattern
  - Use of adapter pattern
  - Use of command pattern
- Modifying presentation behavior
- Modifying commands

# 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

# Modifying Presentation Behavior

- Introduction to EMF.edit
- EMF.edit and design patterns
- ***Modifying presentation behavior***
  - Change to the item provider text
  - Change to the item provider icon
- Modifying commands

# 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");  
}
```

# Modifying Commands

- Introduction to EMF.edit
- EMF.edit and design patterns
- Modifying presentation behavior
- ***Modifying commands***
  - Use of command in EMF
  - Overriding commands

# Use of Command in EMF

- All modification in EMF are done through commands
  - Menu action
  - Property changes
  - Drag-n-drop
- The framework uses a combination of framework and generated code
  - The ***Common Command Framework (CCF)***
  - The ***EMF.edit Generated Commands***
- Three roles
  - ***Commands***
  - ***Command stack***
  - ***Command factory***

# Commands in EMF.edit

- The commands in EMF.edit are handled using the design pattern *Template Method*
- The ItemProviderAdapter implements a createCommand(...) method
  - Checks what the user wants to do
  - Dispatches to protected member functions based on requested user actions, e.g.:
    - createAddCommand(...)
    - createRemoveCommand(...)
    - ...
- Overriding the commands in EMF.edit usually involves overriding one of the protected dispatch functions

# Example: Overriding SetCommand for Artist Name

```
public class SetArtistNameCommand extends SetCommand {
    public SetQuantityCommand(EditingDomain domain, EObject owner,
        EStructuralFeature feature, Object value ) {
        super( domain, owner, feature, value);
    }
    public void doExecute() {
        Artist work = (Artist)this.owner;
        Logger.log("Name of artist changed from " +
            work.getName() +
            " to " +
            this.value.toString());
        super.doExecute();
    }
}
```

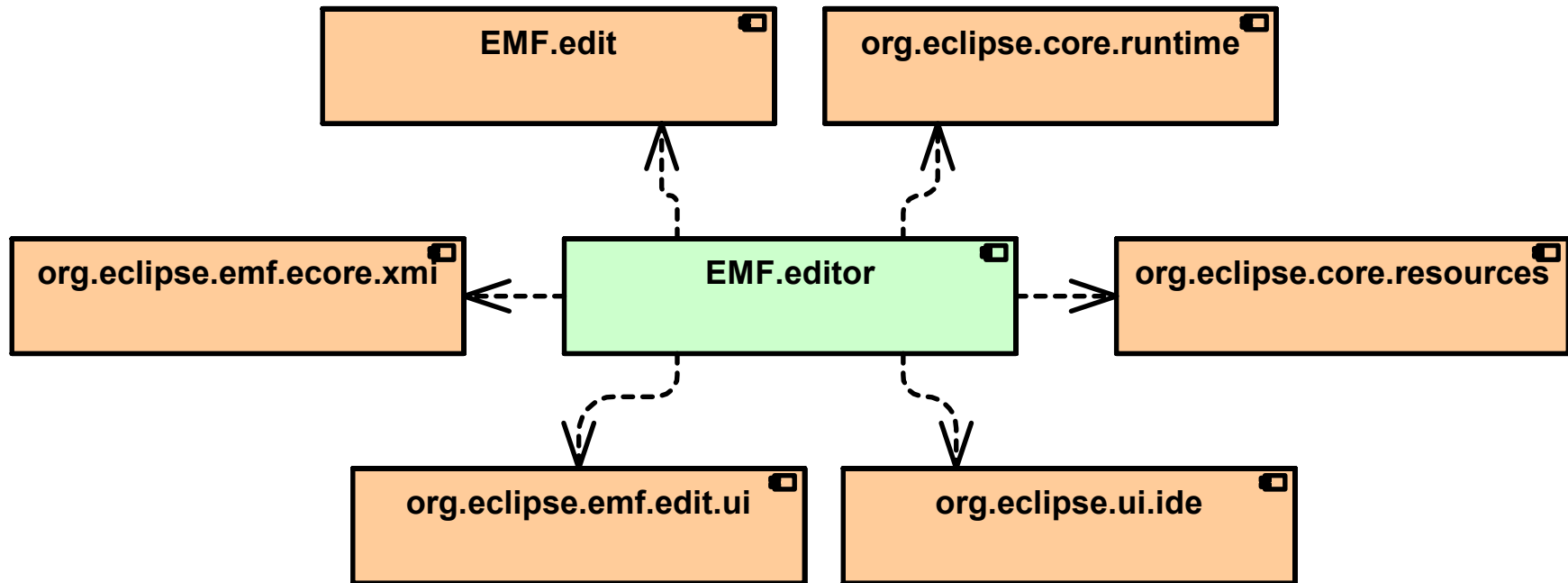
- Simple example adding logging
- More sophisticated examples may require you to extend the compound command

# EMF.editor



# Overview of the EMF.editor

- ***Overview of the EMF.editor***
  - Role of the EMF.editor
  - What is generated?



- 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 be "EMF Example New Wizards"
- More than one view
  - The multi-page editor only activates the tree editor properly
  - Change to custom dialog likely

# What is 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

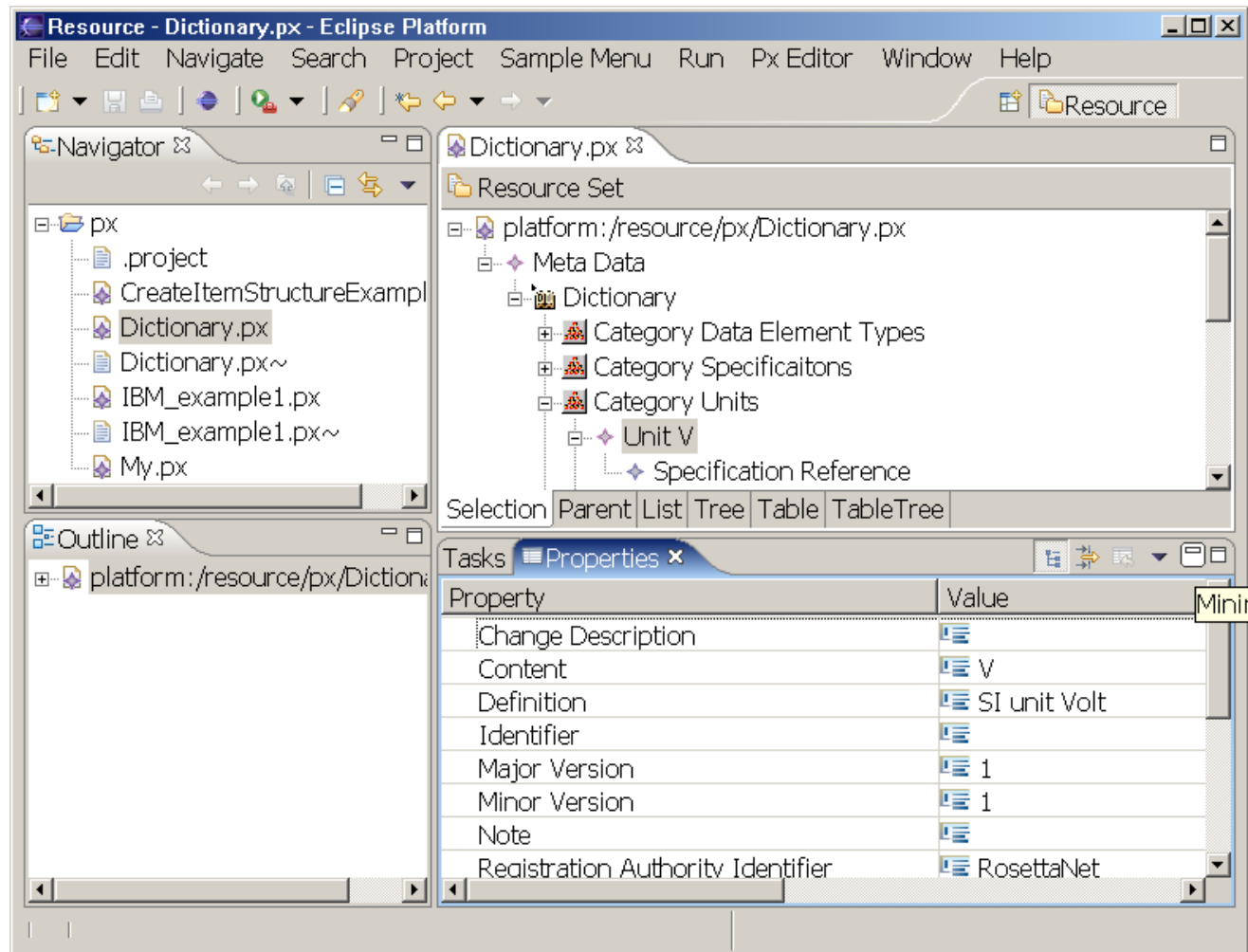
# Summary and Conclusions



# Some Experiences with EMF

- Positive experiences
  - It is a quick and effective way of testing requirements
    - Few people can read UML models
    - Most people can interact with an application
    - EMF has been used very effectively by InferData to test and prove models
  - An excellent start for building editor plug-ins
    - Used for many of the editors in WSAD
- Some negative experiences
  - Inflexible with respect to change to the JET templates
    - Fixes expected in upcoming releases
  - Error messages are quite difficult to interpret

# Case Study A: RosettaNet Dictionary Architecture

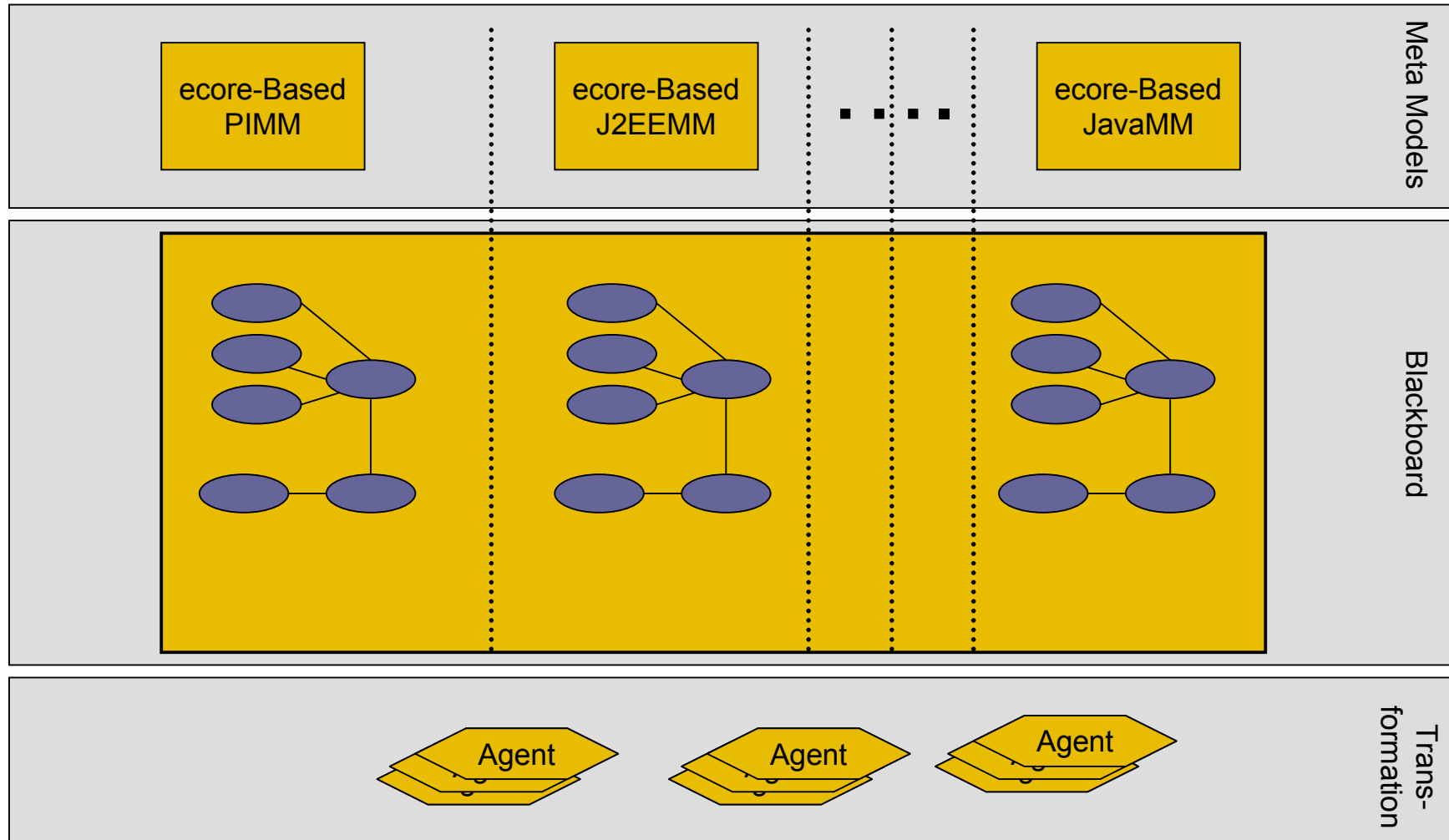


- Used EMF to create a powerful editor used to generate, store and lookup dictionary content

# Experiences from RosettaNet

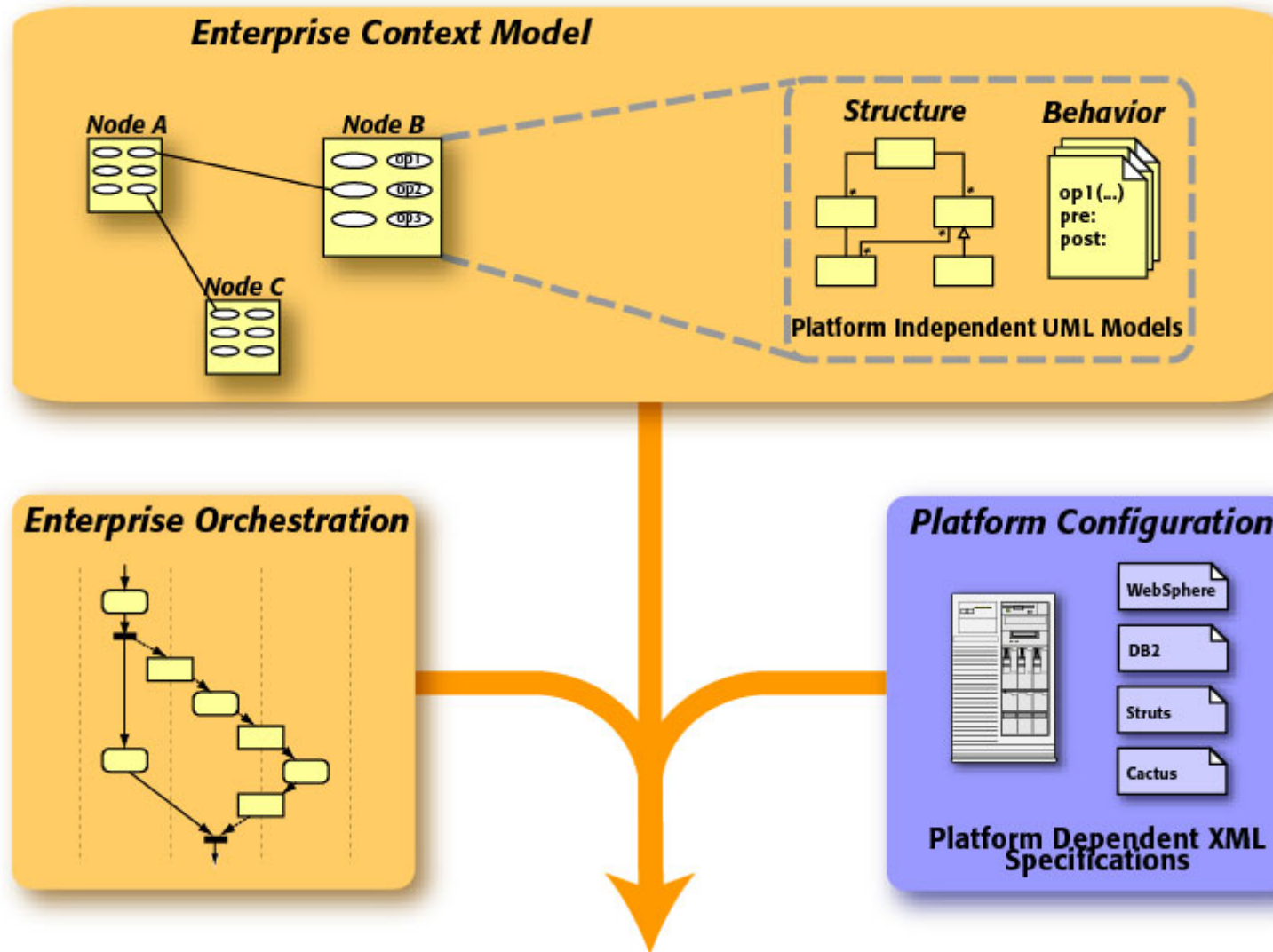
- Prior to the EMF implementation UML models provided the foundation for discussion
- Discussions became much more productive and the quality of the feedback increased after the EMF model was created
- Implementation size and efforts
  - 97 classes, 140 associations
  - ~25.000 lines of code
  - Model input + tailoring ~ 5 days effort

# Case Study B: MCC – An Agent-Based MDA Tool

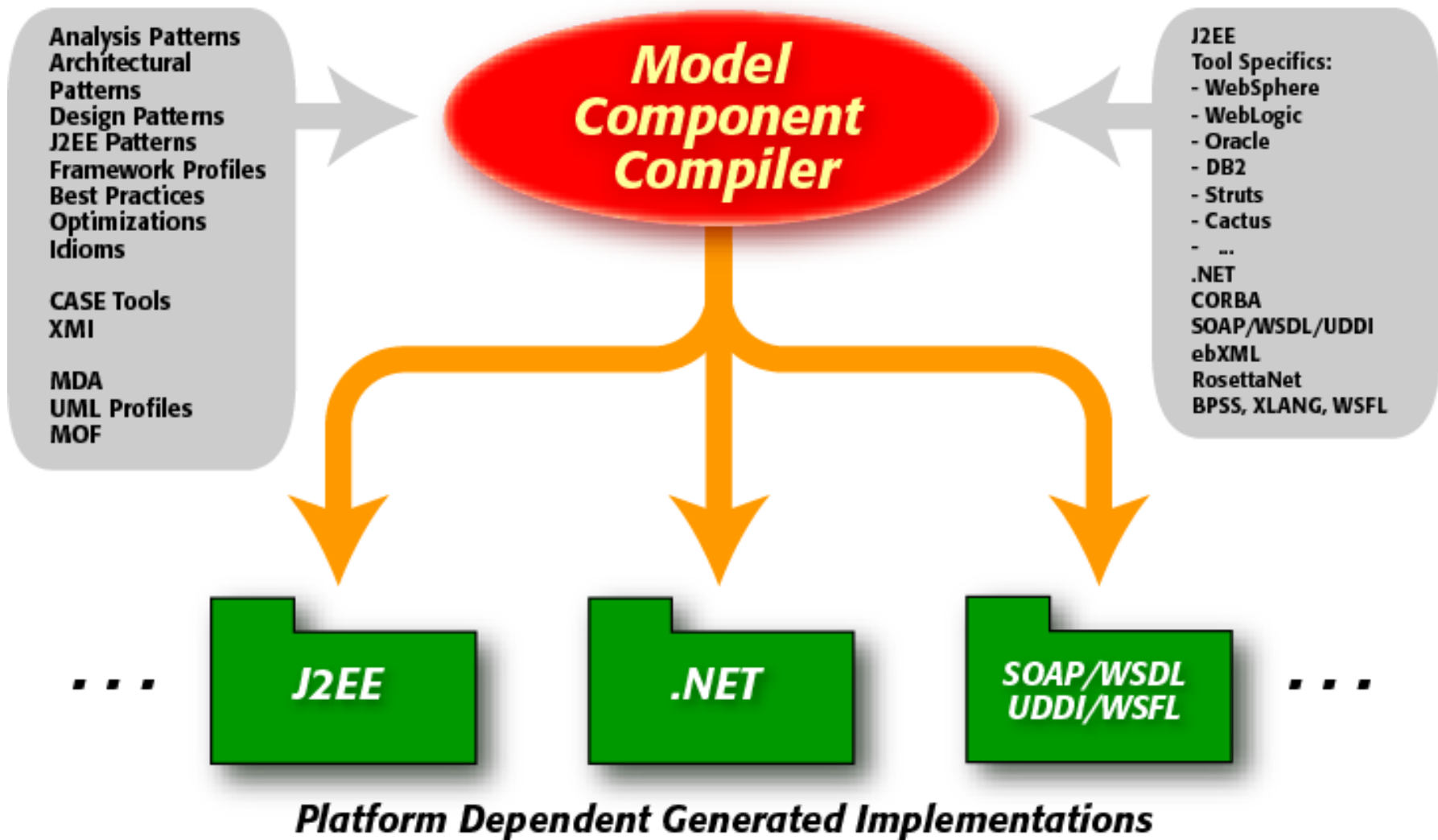


- Model repository and editor for an MCC

# MCC Input Models



# MCC Transformation and Output



# Experiences from MCC

- EMF allowed rapid development and metamodels
- Highly efficient and easy to read model code
- Easy to create domain specific languages
- Some of the domain specific languages created in the tool:
  - Enterprise PIM
  - Action semantic
  - Declarative Metamodel for .NET
  - Declarative Metamodel for J2EE
  - Declarative Metamodel for Agent-Agent + Agent-Blackboard Dependency
  - ... and many more

# Summary

- MDA
  - Too much hype, but...
    - There is substance behind the concepts
    - Some useful standards and some interesting tools emerging
  
- EMF
  - A powerful tool for generating eclipse plug-ins
  - Next generation tool may become even more powerful and span more code generation areas
  
- Conclusion
  - MDA is here to stay
  - EMF is here to stay
  - ***It is important, don't get left behind!***