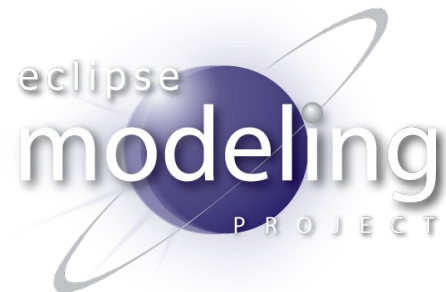


# OOPSLA'06 Tutorial T38


## Introduction to the Eclipse Modeling Framework

→ <http://www.eclipse.org/emf/docs/presentations/OOPSLA/>

Nick Boldt and Marcelo Paternostro  
IBM Rational Software  
Toronto, Canada  
EMF Project

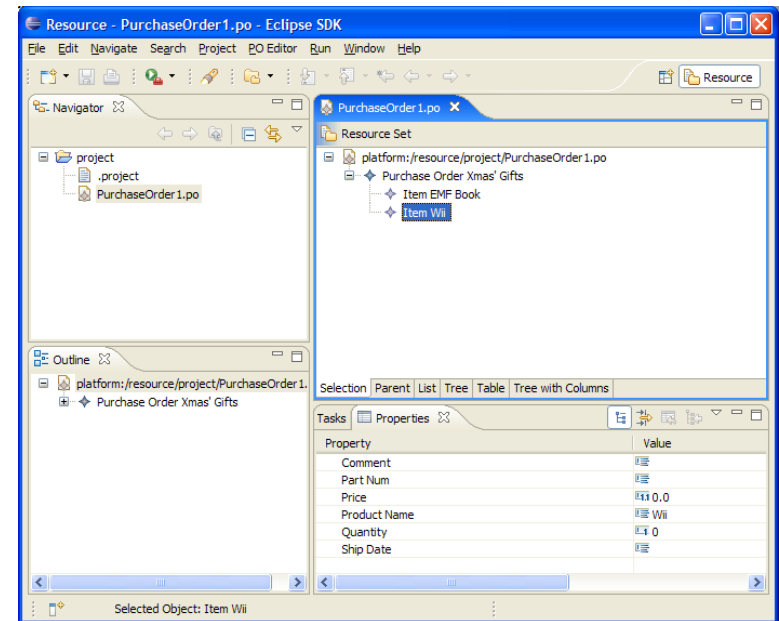
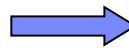
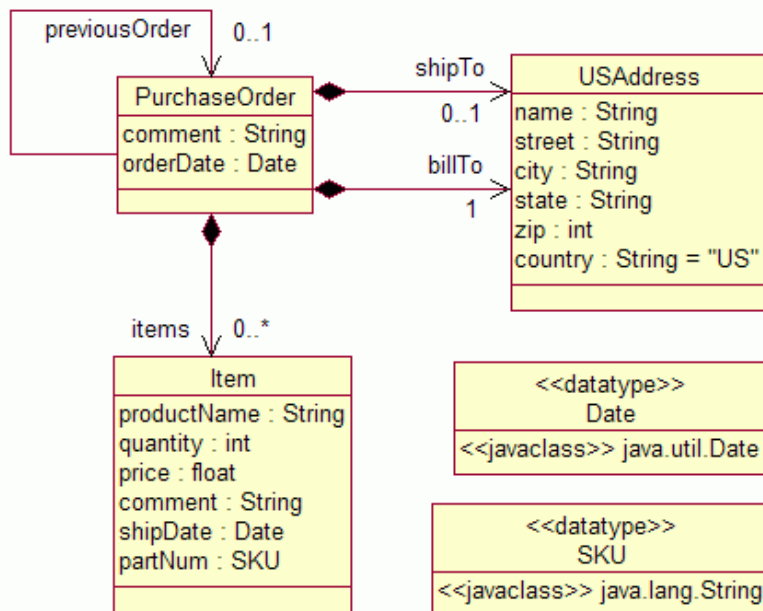


# Agenda


- **Demo**
  - Introduction
    - EMF in a Nutshell
    - EMF Components
    - The Ecore Metamodel
  - Exercise 1: Code Generation, Regeneration and Merge
  - Exercise 2: EMF Runtime
- 
- 
- Exercise 3: Recording Changes
  - Exercise 4: Validation
  - Exercise 5: Reflection, Dynamic EMF and XML Processor
- 
- What's New in EMF 2.2
  - Summary

## Demo

- Using EMF to quickly generate a working graphical editor to create and manipulate instances of a UML model



# Agenda

- Demo
  - ***Introduction***
    - EMF in a Nutshell
    - EMF Components
    - The Ecore Metamodel
  - Exercise 1: Code Generation, Regeneration and Merge
  - Exercise 2: EMF Runtime
- 
- 
- Exercise 3: Recording Changes
  - Exercise 4: Validation
  - Exercise 5: Reflection, Dynamic EMF and XML Processor
- 
- What's New in EMF 2.2
  - Summary

# What is EMF?

- A modeling & data integration framework
- Exploits the facilities offered in Eclipse to...
  - Generate code without losing user customizations (merge)
  - Automate important tasks (such as registering the runtime information)
  - Improve extensibility
  - Provide a UI layer
- What is an EMF “model”?
  - Specification of your application’s data
    - Object attributes
    - Relationships (associations) between objects
    - Operations available on each object
    - Simple constraints (eg. cardinality) on objects and relationships
  - Essentially it represents the class diagram of the application

## What does EMF Provide?

- From a model specification, EMF can generate efficient, correct, and easily customizable implementation code
- Out of the box, EMF provides support for
  - Java™ interfaces
  - UML
  - XML Schema
- EMF converts your models to Ecore (EMF metamodel)
- Tooling support within the Eclipse framework (UI, headless mode, Ant and standalone), including support for generating Eclipse-based and RCP editors
- Reflective API and dynamic model definition
- Persistence API with out of box support for XML/XMI (de)serialization of instances of a model
- And much more....

## Why EMF?

- EMF is middle ground in the modeling vs. programming worlds
  - Focus is on class diagram subset of UML modeling (object model)
  - Transforms models into Java code
  - Provides the infrastructure to use models effectively in your application
- Very low cost of entry
  - EMF is free and open source
  - Full scale graphical modeling tool not required
  - Reuses your knowledge of UML, XML Schema, or Java
- It's real, proven technology (since 2002)

# EMF History

- First version was released in June, 2002
- Originally based on MOF (Meta Object Facility)
  - From OMG (Object Management Group)
  - Abstract language and framework for specifying, constructing, and managing technology neutral metamodels
- EMF evolved based on experience supporting a large set of tools
  - Efficient Java implementation of a practical subset of the MOF API
- 2003: EMOF defined (Essential MOF)
  - Part of OMG's MOF 2 specification; UML2 based
  - EMF is approximately the same functionality
    - Significant contributor to the spec; adapting to it



# Who is Using EMF Today?

- Eclipse projects
  - UML2
  - Graphical Modeling Framework (GMF)
  - EMF Technologies (EMFT): OCL, Validation, Query, Transaction, EODM, and Database Persistence
  - Visual Editor (VE)
  - Data Tools Platform (DTP)
  - Web Tools Platform (WTP)
  - Test and Performance Tools Platform (TPTP)
  - Business Intelligence and Reporting Tools (BIRT)  
... to name but a few
- Commercial offerings
  - IBM, Borland, Oracle, Omondo, Versata, MetaMatrix, Bosch, Ensemble, ...
- Applied sciences
  - Darmstadt University of Technology, Mayo Clinic College of Medicine, European Space Agency, ...
- Large open source community
  - Over 770,000 download requests from January to July 2006
  - In first month of its release, over 100,000 download requests for EMF 2.2.0!

## EMF at IBM

- Pervasive usage across product lines
  - IBM® Rational® Software Architect
  - IBM Rational Application Developer for WebSphere Software
  - IBM WebSphere® Integration Developer
  - IBM WebSphere Application Server
  - IBM Lotus® Workplace
- Emerging technology projects: alphaWorks
  - Emfatic Language for EMF Development  
(<http://www.alphaworks.ibm.com/tech/emfatic>)
  - Model Transformation Framework  
(<http://www.alphaworks.ibm.com/tech/mtf>)
  - XML Forms Generator (<http://www.alphaworks.ibm.com/tech/xfg>)

# What Have People Said About EMF?

- EMF represents the **core subset** that's left when the non-essentials are eliminated. It represents a **rock solid foundation** upon which the more ambitious extensions of UML and MDA can be built.
  - Vlad Varnica, OMONDO Business Development Director, 2002
- EMF **provides the glue between the modeling and programming worlds**, offering an infrastructure to use models effectively in code by integrating UML, XML and Java. EMF thus fits well into [the] Model-Driven Development approach, and is **critically important for Model-Driven Architecture**, which underpins service-oriented architectures [SOA].
  - Jason Bloomberg, Senior analyst for XML & Web services, ZapThink, 2003
- EMF is capable of **creating sophisticated editors** from abstract business models. ... EMF creates feature complete implementations including persistence, **business model implementation**, editing framework and editors. ... At InferData, we have been using EMF [to] create **persistence implementation** for various in-house products, ... **standalone products** for the Eclipse platform, [and] quick prototypes to validate complex business models. Business models remain technology independent; code generation is performed for all that can be code generated and kept separate from the manually developed code.
  - Petter Graff, SYS-CON Media, 2004
- EMF was chosen because it (a) provides a **lightweight, pragmatic approach to modeling** with very **low entry cost** and is thus suitable for rapid prototyping, (b) unifies key technologies such as Java and XML, and (c) integrates well into Eclipse.
  - Bruch, Bockisch, Schäfer, Mezini, Darmstadt Univ. of Technology, Germany, 2005
- [As] a consultant with fiduciary responsibility to my customers, [...] given the **enormous traction** that Eclipse has gathered, we have to view the EMF metadata management framework as the **de facto standard**.
  - David Frankel, as seen in Business Process Trends, March 2005

## Creating the Ecore Model

- Representing the modeled domain in Ecore is the first step in using EMF
- Ecore can be created
  - Directly using the EMF editors
  - Through a graphical UI provided by external contributions
  - By converting a model specification for which a Model Importer is available
- Model Importers available in EMF
  - Java Interfaces
  - UML models expressed in Rational Rose® files
  - XML Schema
- Choose the one matching your perspective or skills

# Model Importers Available in EMF

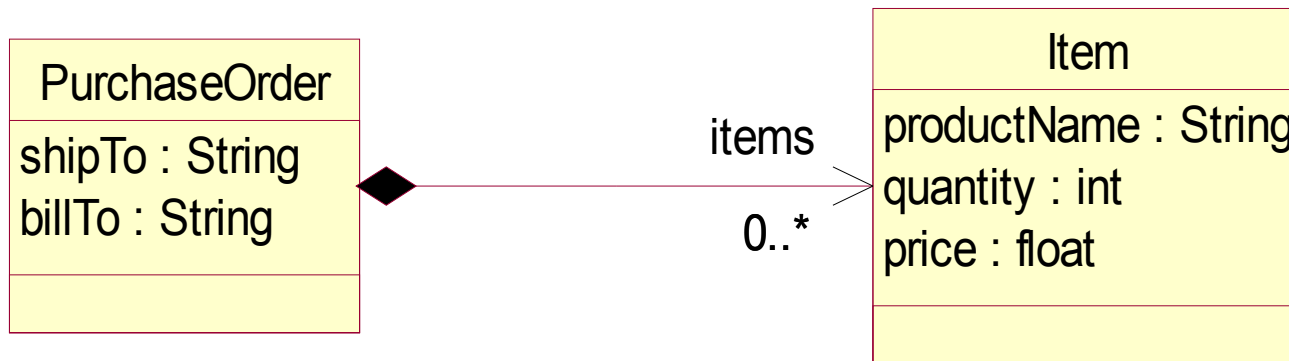
- Java Interfaces

```
public interface PurchaseOrder
{
    String getShipTo();
    void setShipTo(String value);
    String getBillTo();
    void setBillTo(String value);
    List getItems(); // List of Item
}
```

```
public interface Item
{
    String getProductName();
    void setProductName(String value);
    int getQuantity();
    void setQuantity(int value);
    float getPrice();
    void setPrice(float value);
}
```

## Model Importers Available in EMF

- UML Class Diagram



# Model Importers Available in EMF

## ■ XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.example.com/SimplePO"
            xmlns:PO="http://www.example.com/SimplePO">
  <xsd:complexType name="PurchaseOrder">
    <xsd:sequence>
      <xsd:element name="shipTo" type="xsd:string"/>
      <xsd:element name="billTo" type="xsd:string"/>
      <xsd:element name="items" type="PO:Item"
                    minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Item">
    <xsd:sequence>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="quantity" type="xsd:int"/>
      <xsd:element name="price" type="xsd:float"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

# Unifying Java, XML and UML Technologies


- The Model Importers available in EMF were carefully chosen to integrate today's most important technologies
- All three forms provide the same information
  - Different visualization/representation
  - The application's "model" of the structure
- From a model definition, EMF can generate
  - Java implementation code, including UI
  - XML Schemas
  - Eclipse projects and plug-in



## Typical EMF Usage Scenario

- Create an Ecore model that represents the domain you are working on
  - Import UML (e.g. Rose .mdl file)
  - Import XML Schema
  - Import annotated Java interfaces
  - Create Ecore model directly using EMF's Ecore editor or a graphical editor
- Generate Java code for model
- Prime the model with instance data using generated EMF model editor
- Iteratively refine model (and regenerate code) and develop Java application
  - You will use the EMF generated code to implement the use cases of your application
- Optionally, use EMF.Edit to build customized user interface

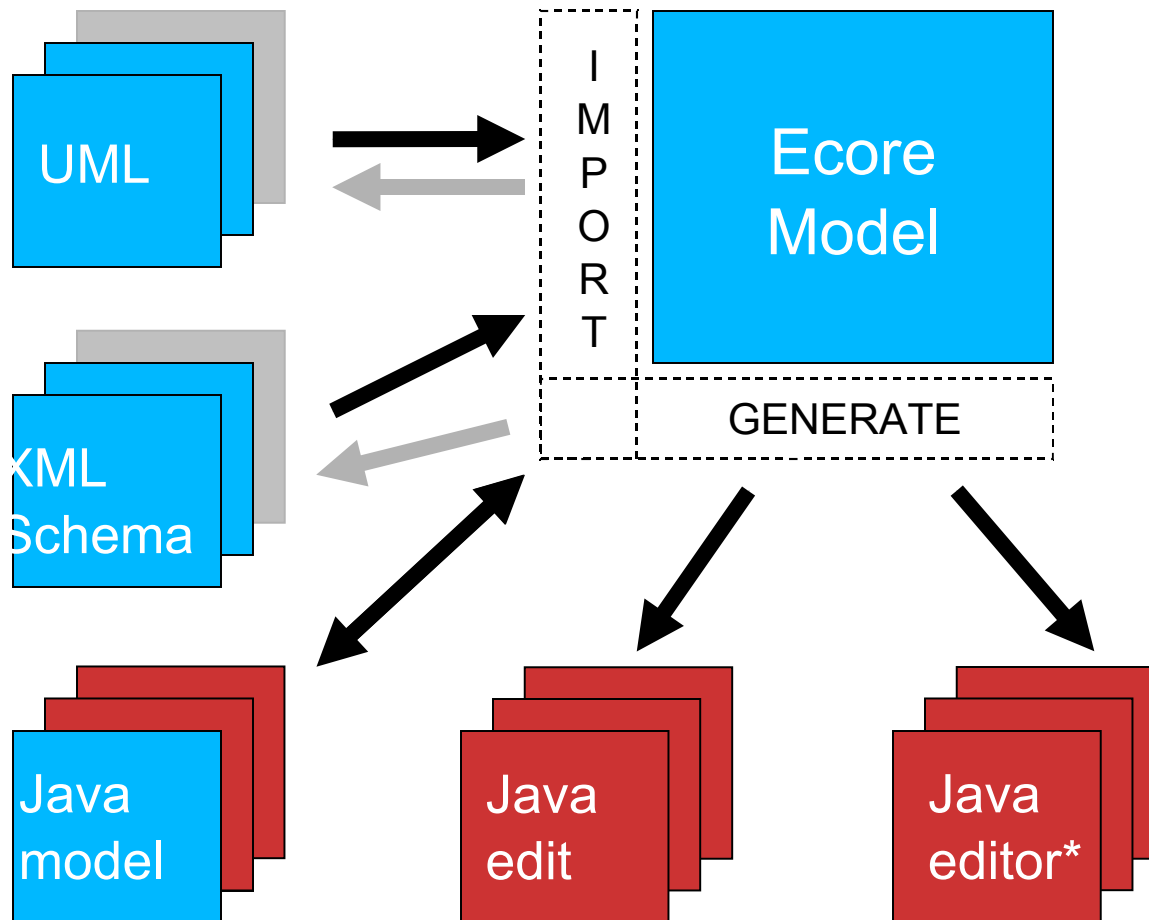
# Agenda

- Demo
  - Introduction
    - EMF in a Nutshell
    - ***EMF Components***
    - The Ecore Metamodel
  - Exercise 1: Code Generation, Regeneration and Merge
  - Exercise 2: EMF Runtime
- 
- 
- Exercise 3: Recording Changes
  - Exercise 4: Validation
  - Exercise 5: Reflection, Dynamic EMF and XML Processor
- 
- What's New in EMF 2.2
  - Summary

# EMF Components

- EMF Core
  - Ecore metamodel
  - Model change notification & validation
  - Persistence and serialization
  - Reflection API
  - Runtime support for generated models
- EMF Edit
  - Helps integrate models with a rich user interface
  - Used to build editors and viewers for your model
  - Includes default reflective model editor
- EMF Codegen
  - Code generator for core and edit based components
  - Extensible model importer framework

## EMF Tools: Model Import and Generation



### Generator Features:

- Customizable JSP-like templates (JET)
- JDT-integrated, command-line, or Ant
- Fully supports regeneration and merge

\* Eclipse IDE-integrated or RCP-based

# EMF Model Importers

- UML
  - Rational Rose .mdl file
  - Eclipse UML2 project provides importer for .uml2
- Annotated Java
  - Java interfaces representing modeled classes
  - Javadoc annotations using @model tags to express model properties not captured by method declarations
  - Lowest cost approach
- XML Schema
  - Describes the data of the modeled domain
  - Provides richer description of the data, which EMF exploits
- Ecore model (\*.ecore file)
  - Just creates the generator model (discussed later)
  - Also handles EMOF (\*.emof)

## Ecore Model Creation

- An Ecore model is created within an Eclipse project via a wizard
- Input: one of the model specifications from the previous slide
- Output:
  - `modelname.ecore`
    - Ecore model file in XMI format
    - Canonical form of the model
  - `modelname.genmodel`
    - A “generator model” for specifying generator options
    - Decorates `.ecore` file
    - EMF code generator is an EMF `.genmodel` editor
    - Automatically kept in synch with `.ecore` file

## Ecore Model Editor

- A generated (and customized) EMF editor for the Ecore model
- Create, delete, etc. model elements (EClass, EAttribute, EReference, etc.) using pop-up actions in the editor's tree
- Set names, etc. in the Properties view

The screenshot displays the Eclipse Ecore Model Editor interface. The top pane shows a 'Resource Set' with a tree view of the model structure. The bottom pane shows the 'Properties' view for the selected element.

**Resource Set Tree:**

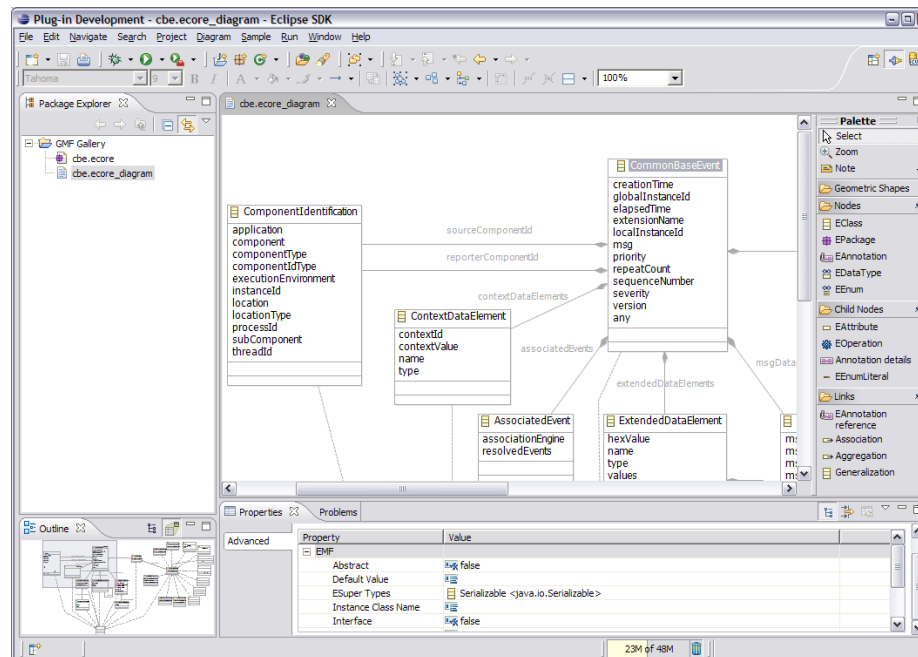
- platform:/resource/com.example.po/model/po.ecore
  - po
    - DocumentRoot
    - Item
    - PurchaseOrder
    - QuantityType <int>
    - QuantityTypeObject <java.lang.Integer>
    - SKU <java.lang.String>
    - USAddress
      - ExtendedMetaData
      - name : String
      - street : String
      - city : String
      - state : String
      - zip : Int
      - country : NMTOKEN

**Properties View:**

Property	Value
Changeable	<input checked="" type="checkbox"/> true
Default Value	<input type="text"/>
Default Value Literal	<input type="text"/>
Derived	<input checked="" type="checkbox"/> false
EAttribute Type	String <java.lang.String>
EContaining Class	USAddress
EType	String <java.lang.String>
ID	<input checked="" type="checkbox"/> false
Lower Bound	1
Many	<input checked="" type="checkbox"/> false
Name	state

# Ecore Model Editor

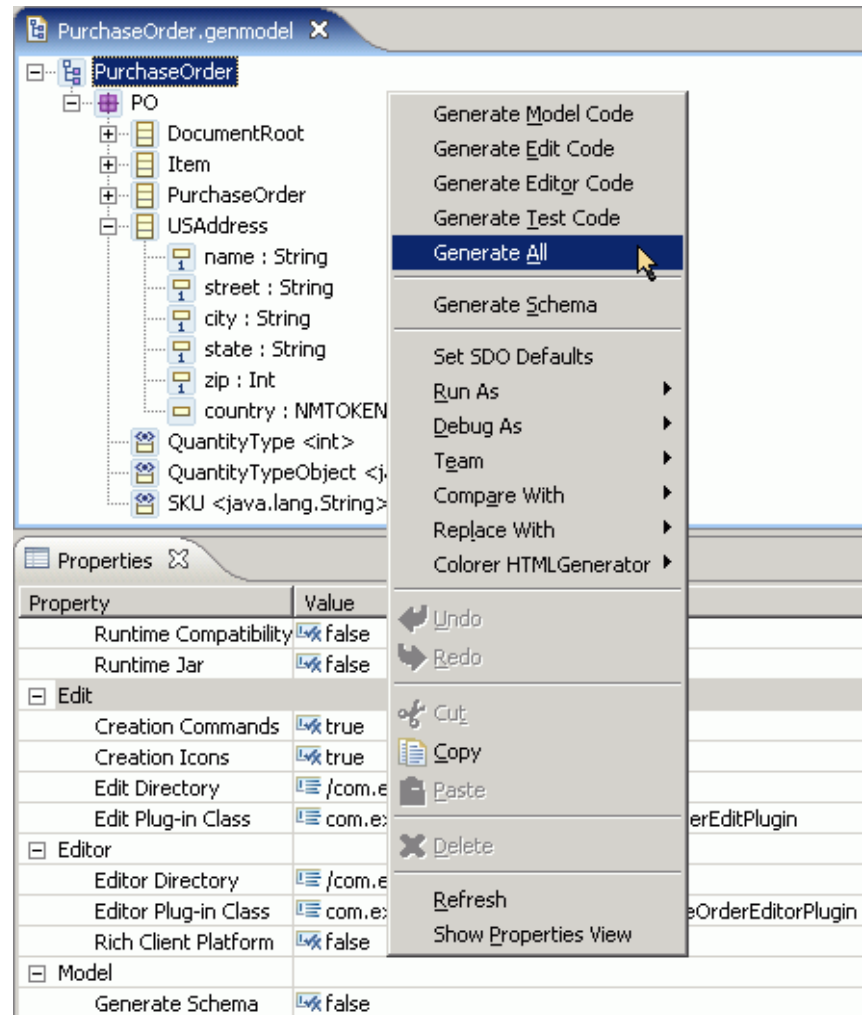
- A graphical editor is a better approach
  - GMF Ecore Diagram Example (<http://www.eclipse.org/gmf/>)
  - Omondo EclipseUML (<http://www.omondo.com/>)






## EMF Generator

- Similar layout to Ecore model editor
- Automatically keeps in synch with .ecore changes
- Generate code with pop-up menu actions
  - Generate Model Code
  - Generate Edit Code
  - Generate Editor Code
  - Generate Test Code
  - Generate All
- Code generation options in Properties view
- Generator > Reload to reload .genmodel and .ecore files from original model form

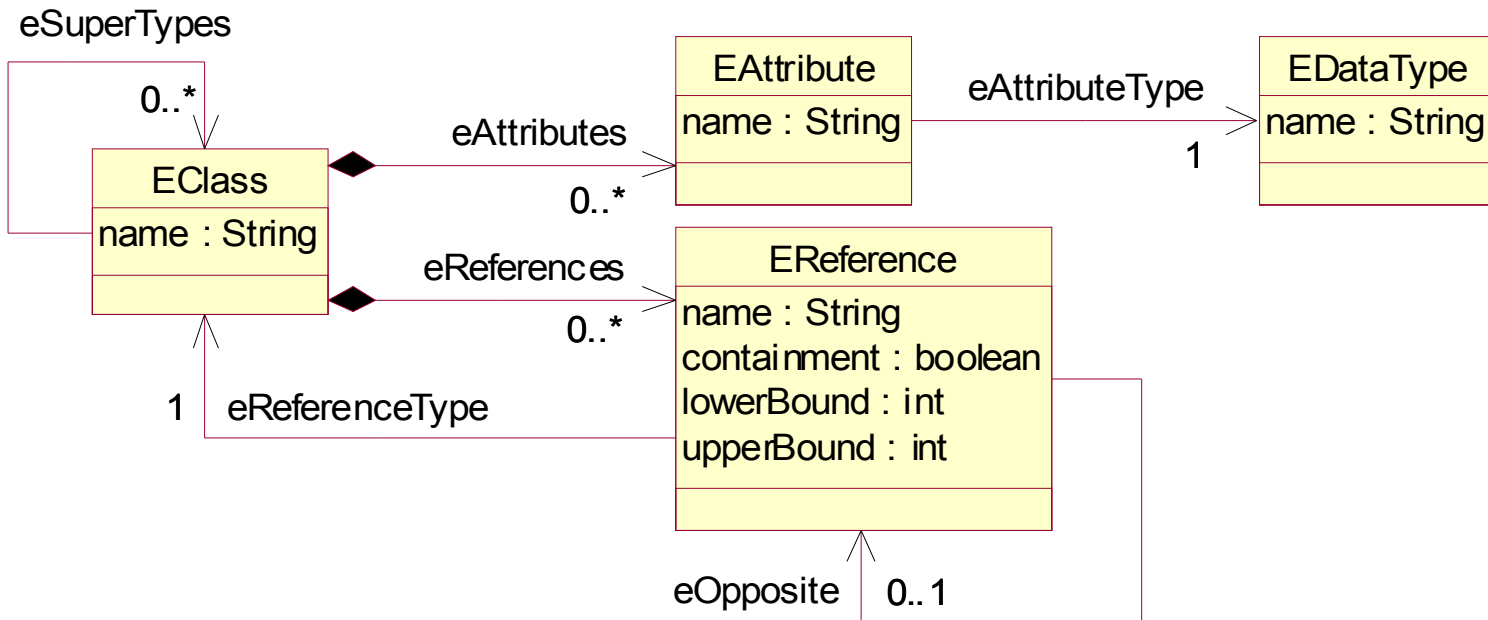


# Agenda

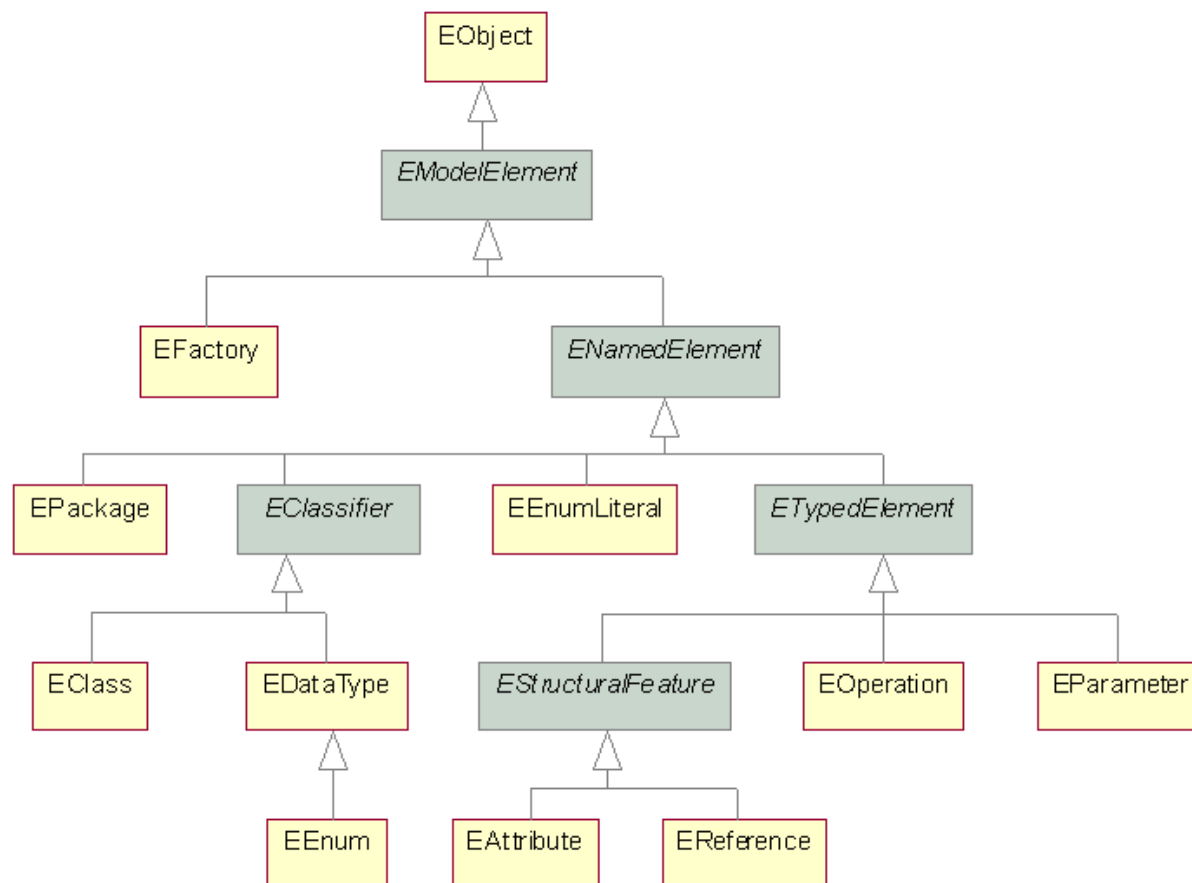
- Demo
  - Introduction
    - EMF in a Nutshell
    - EMF Components
    - ***The Ecore Metamodel***
  - Exercise 1: Code Generation, Regeneration and Merge
  - Exercise 2: EMF Runtime
- 
- Exercise 3: Recording Changes
  - Exercise 4: Validation
  - Exercise 5: Reflection, Dynamic EMF and XML Processor
- What's New in EMF 2.2
  - Summary

# The Ecore (Meta) Model

- Ecore is EMF's model of a model
  - Also called a “metamodel”
  - Persistent representation is XML



# The Ecore Metamodel



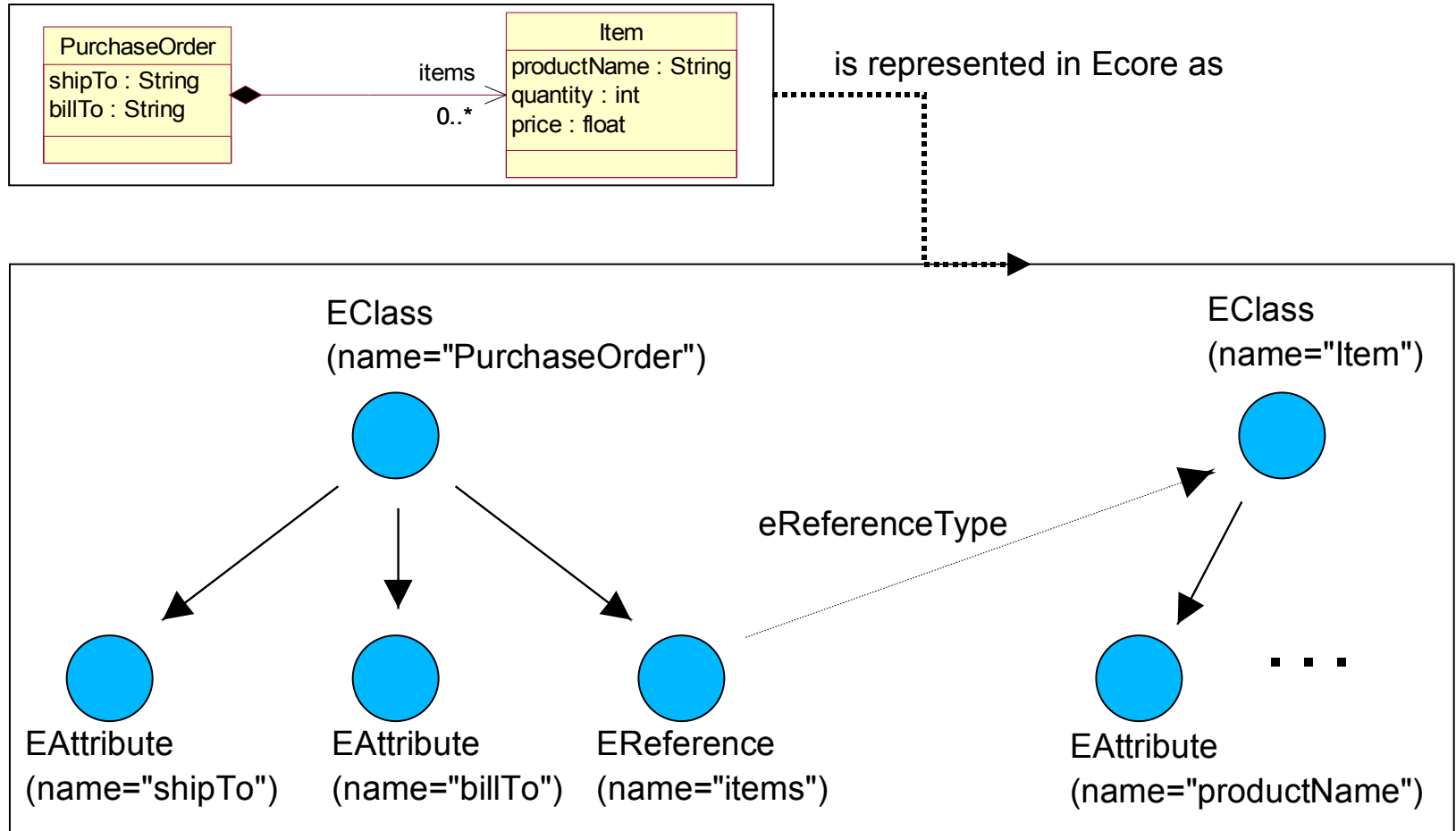
- EObject is the root of every model object – equivalent to `java.lang.Object`

## Partial List of Ecore Data Types

<b>Ecore Data Type</b>	<b>Java Primitive Type or Class</b>
EBoolean	boolean
EChar	char
EFloat	float
EString	java.lang.String
EByteArray	byte[ ]
EBooleanObject	java.lang.Boolean
EFloatObject	java.lang.Float
EJavaObject	java.lang.Object

- Ecore data types are serializable and custom data types are supported

# Ecore Model for Purchase Orders



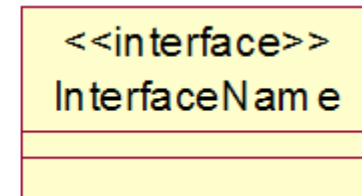
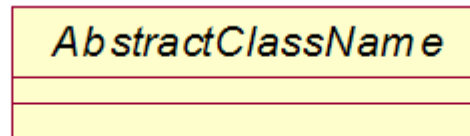
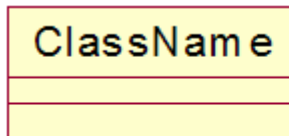
## Purchase Order Ecore XMI

```
<eClassifiers xsi:type="ecore:EClass"
  name="PurchaseOrder">
  <eReferences name="items" eType="#//Item"
    upperBound="-1" containment="true"/>
  <eAttributes name="shipTo"
    eType="ecore:EDatatype http:...Ecore#//EString"/>
  <eAttributes name="billTo"
    eType="ecore:EDatatype http:...Ecore#//EString"/>
</eClassifiers>
```

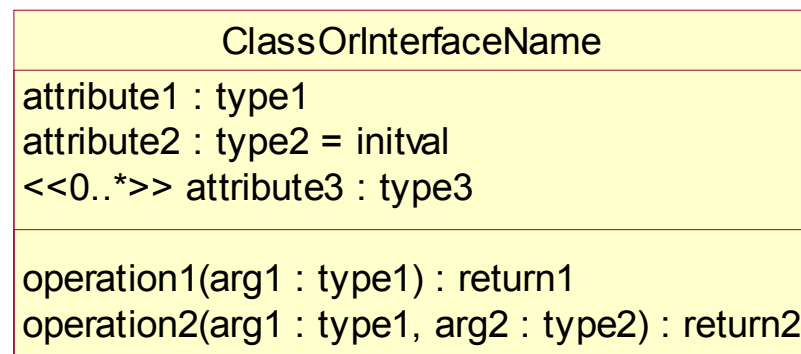
- Alternate serialization format is EMOF (Essential MOF) XMI
  - Part of OMG Meta Object Facility (MOF) 2.0 standard (<http://www.omg.org/docs/ptc/04-10-15.pdf>)

# UML Constructs Available in Ecore

- Classes, Abstract Classes, and Interfaces



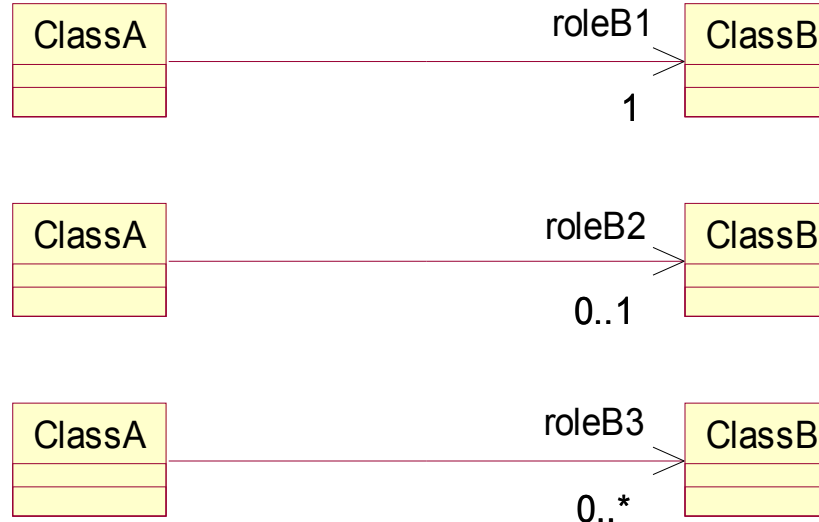
- Attributes and Operations





## UML Constructs Available in Ecore

- References (Associations)
  - One-way



# UML Constructs Available in Ecore

- References (Associations)
  - Bidirectional

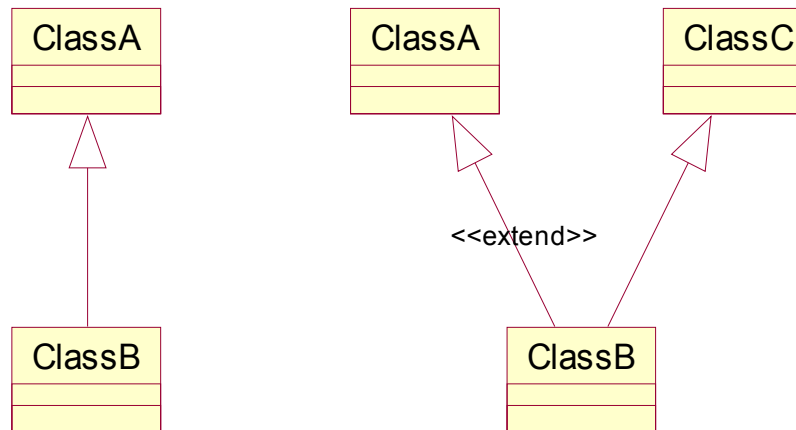


- Containment

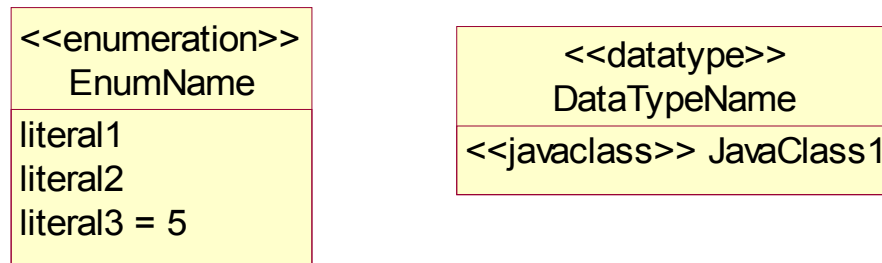


# UML Constructs Available in Ecore


## ■ Class Inheritance



## ■ Enumerations and Data Types



# Agenda

- Demo
- Introduction
  - EMF in a Nutshell
  - EMF Components
  - The Ecore Metamodel
- ***Exercise 1: Code Generation, Regeneration and Merge***
- Exercise 2: EMF Runtime
- 
  - Exercise 3: Recording Changes
  - Exercise 4: Validation
  - Exercise 5: Reflection, Dynamic EMF and XML Processor
- What's New in EMF 2.2
- Summary

## Code Generation

- EMF framework is lightweight
  - Generated code is clean, simple, efficient
- EMF can generate
  - Model implementation
  - UI-independent edit support
  - Editor and views for Eclipse IDE-integrated or RCP application
  - JUnit test skeletons
  - Manifests, plug-in classes, properties, icons, etc.

## Generated Model Code

- Interface and implementation for each modeled class
  - Includes get/set accessors for attributes and references

```
public interface PurchaseOrder extends EObject
{
    String getShipTo();
    void setShipTo(String value);
    String getBillTo();
    void setBillTo(String value);
    EList getItems();
}
```

- Usage example

```
order.getItems().add(item);
```

## Generated Model Code

- Factory to create instances of model objects

```
POFactory factory = POFactory.eINSTANCE;  
PurchaseOrder order = factory.createPurchaseOrder();
```

- Package class provides access to metadata

```
POPackage poPackage = POPackage.eINSTANCE;  
EClass itemClass = poPackage.getItem();  
  
EAttribute priceAttr = poPackage.getItem_Price();  
//or itemClass.getEStructuralFeature(POPackage.ITEM__PRICE)
```

- Also generated: switch utility, adapter factory base, validator, custom resource, XML processor

## Generated Edit/Editor Code

- Viewing/editing code divided into two parts
  - UI-independent code
    - Item providers (adapters)
    - Item provider adapter factory
  - UI-dependent code
    - Model creation wizard
    - Editor
    - Action bar contributor
    - Advisor (RCP)
- By default each part is placed in a separate Eclipse plug-in



# Summary of Generated Artifacts

- Model
  - Interfaces and classes
  - Type-safe enumerations
  - Package (metadata)
  - Factory
  - Switch utility
  - Adapter factory base
  - Validator
  - Custom resource
  - XML Processor
- Editor
  - Model Wizard
  - Editor
  - Action bar contributor
  - Advisor (RCP)
- Tests
  - Test cases
  - Test suite
  - Stand-alone example
- Manifests, plug-in classes, properties, icons...
- Edit (UI independent)
  - Item providers
  - Item provider adapter factory

## Regeneration and Merge

- Hand-written code can be added to generated code and preserved during regeneration
  - This merge capability has an Eclipse dependency, so is not available standalone
- All generated classes, interfaces, methods and fields include `@generated` marker in their Javadoc
- To replace generated code:
  - Remove `@generated` marker
  - Or include additional text, e.g.  
`@generated NOT`
- Methods without `@generated` marker are left alone during regeneration

## Regeneration and Merge

- Extend (vs. replace) generated method through redirection
  - Append “Gen” suffix to the generated method's name

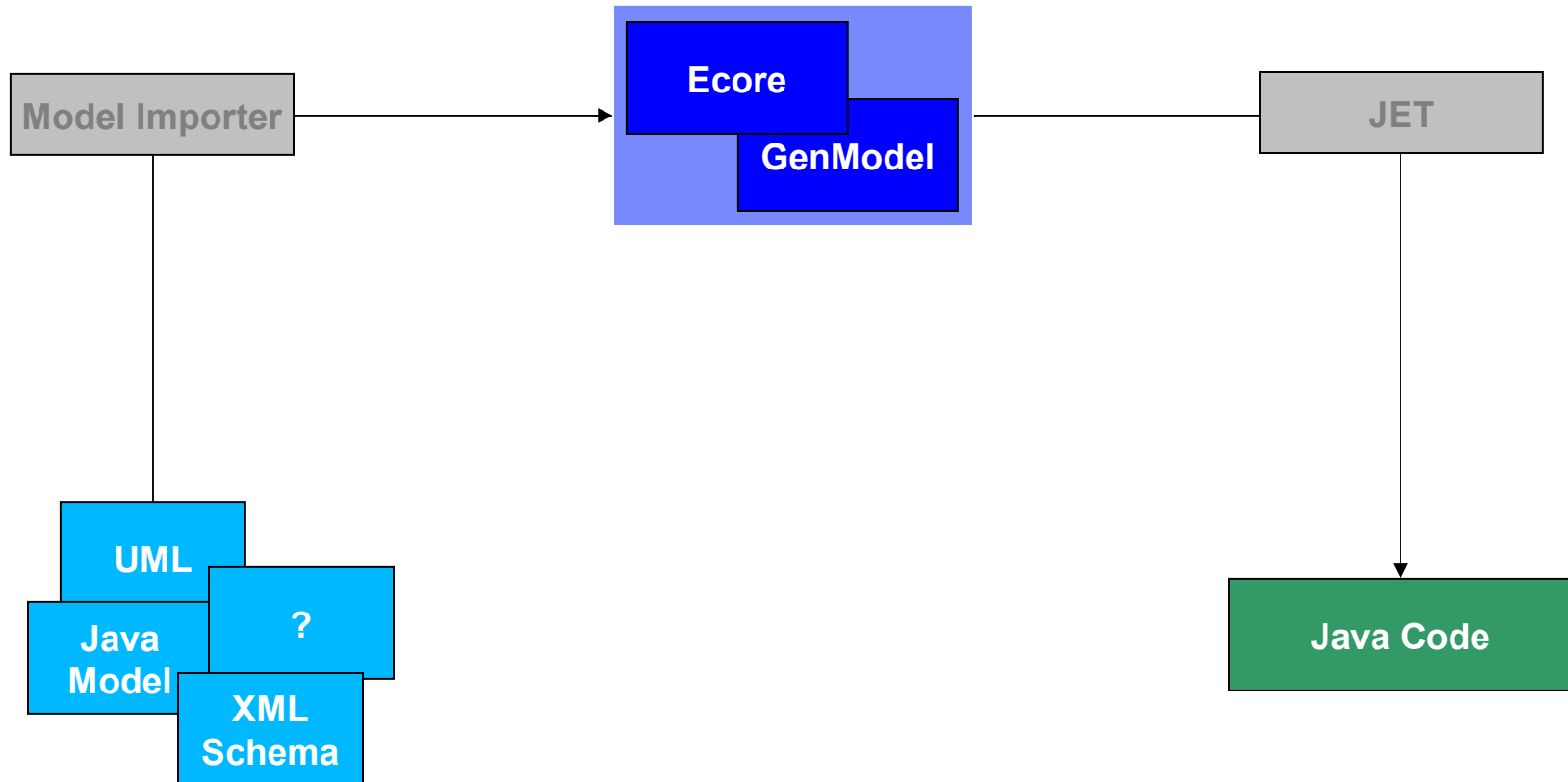
```
/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public String getName()
{
    return name;
}
```



```
/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public String getNameGen()
{
    return name;
}

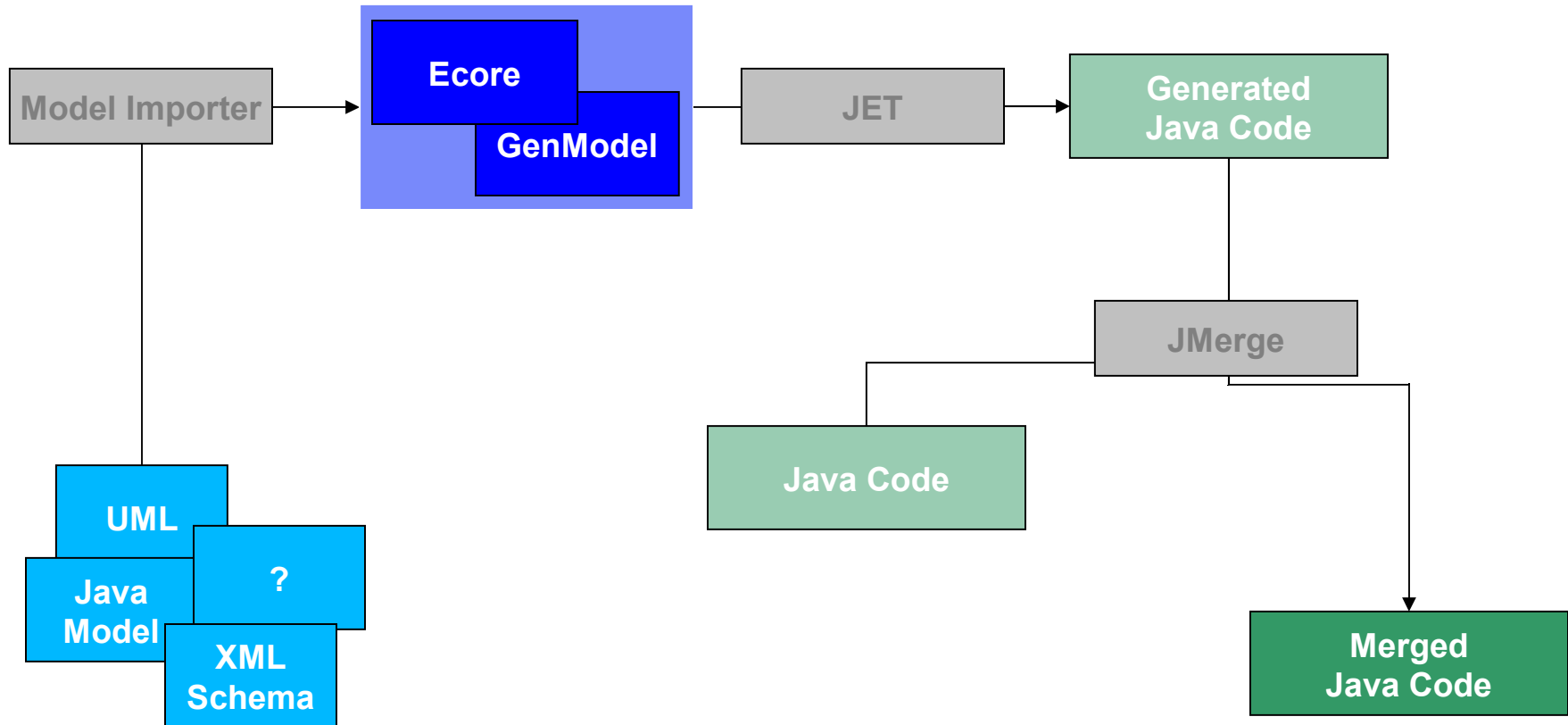
public String getName()
{
    return format(getNameGen());
}
```

# Summarizing the Code Generation Process



Simplified version

# Summarizing the Code Generation Process




Full version

# Exercise 1:

## Code Generation, Regeneration and Merge

# Agenda

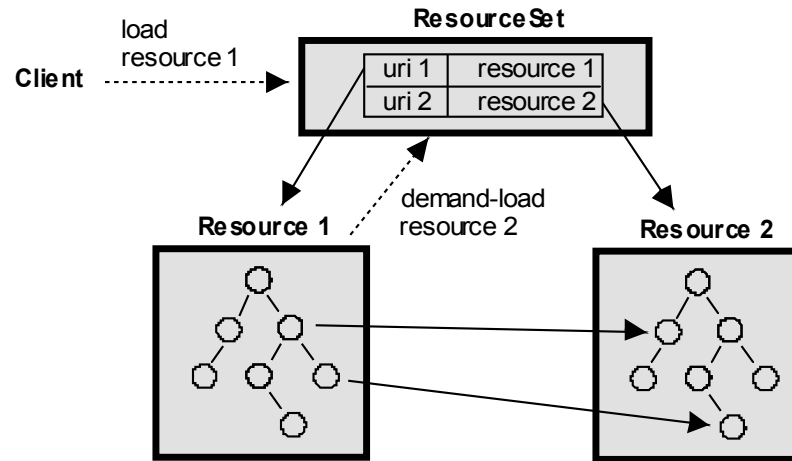
- Demo
  - Introduction
    - EMF in a Nutshell
    - EMF Components
    - The Ecore Metamodel
  - Exercise 1: Code Generation, Regeneration and Merge
  - ***Exercise 2: EMF Runtime***
- 
- Exercise 3: Recording Changes
  - Exercise 4: Validation
  - Exercise 5: Reflection, Dynamic EMF and XML Processor
- What's New in EMF 2.2
  - Summary

# EMF Runtime

- Persistence and serialization of model data
  - Proxy resolution and demand load
- Automatic notification of model changes
- Bi-directional reference handshaking
- Dynamic object access through a reflective API
- Runtime environments
  - Eclipse
    - Full IDE
    - RCP
  - Standalone Java



# Persistence and Serialization



- Serialized data is referred to as a resource
- Data can be spread out among a number of resources in a resource set
- One resource is loaded at a time, even if it has references to objects in other resources in the resource set
  - Proxies exist for objects in other resources
  - Lazy or demand loading of other resources as needed
  - A resource can be unloaded

## Resource Set

- Context for multiple resources that may have references among them
- Usually just an instance of `ResourceSetImpl`, or a customized subclass
- Provides factory method for creating new resources in the set:

```
ResourceSet rs = new ResourceSetImpl();  
URI uri = URI.createFileURI("C:/data/po.xml");  
Resource resource = rs.createResource(uri);
```

- Also provides access to the registries, URI converter, and default load options for the set

# Resource Factory Registry

- Returns a resource factory for a given type of resource
  - Based on the URI scheme or filename extension
  - Determines the type of resource, hence format for save/load

```
Resource.Factory.Registry reg = rs.getResourceFactoryRegistry();  
reg.getExtensionToFactoryMap().put("xml", new  
XMLResourceFactoryImpl());
```

- For models created from XML Schema, the generated custom resource factory implementation should be registered to ensure schema-conformant serialization
  - When running as a plug-in under Eclipse, EMF provides an extension point for registering resource factories
  - Generated plugin.xml registers generated resource factory against a package specific extension (e.g. "po")
- Global registry: Resource.Factory.Registry.INSTANCE
  - Consulted if no registered resource factory found locally

# Package Registry

- Returns the package identified by a given namespace URI
  - Used during loading to access the factory for creating instances

```
EPackage.Registry registry = rs.getPackageRegistry();  
registry.put(POPackage.eNS_URI, POPackage.eINSTANCE);
```

- Global registry: EPackage.Registry.INSTANCE
  - Consulted if no registered package found locally
- Running in Eclipse, EMF provides an extension point for globally registering generated packages
- Even standalone, a package automatically registers itself when accessed:

```
POPackage poPackage = POPackage.eINSTANCE;
```

## Resource

- Container for objects that are to be persisted together
  - Convert to and from persistent form via `save()` and `load()`
  - Access contents of resource via `getContents()`

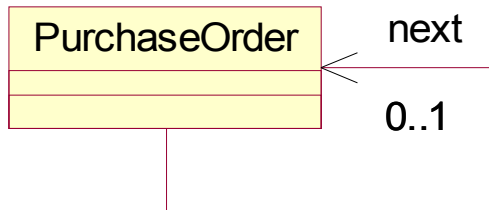
```
URI uri = URI.createFileURI("C:/data/po.xml");  
Resource resource = rs.createResource(uri);  
resource.getContents().add(p1);  
resource.save(null);
```

- EMF provides XMLResource implementation

```
<PurchaseOrder>  
  <shipTo>John Doe</shipTo>  
  <next>p2.xml#p2</next>  
</PurchaseOrder>
```

- Other, customized XML resource implementations, provided, too (e.g. XMI, Ecore, EMOF)

# Proxy Resolution and Demand Load



p1.xml

```

<PurchaseOrder>
  <shipTo>John Doe</shipTo>
  <next>p2.xml#p2</next>
</PurchaseOrder>
  
```

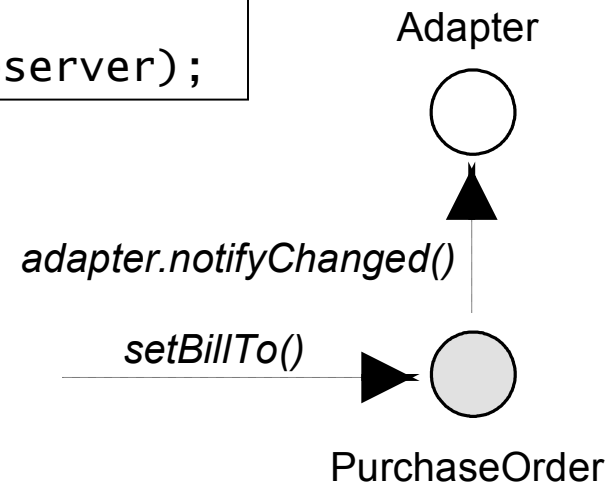


```
PurchaseOrder p2 = p1.getNext();
```

# Model Change Notification

- Every EMF object is also a Notifier
  - Send notification whenever an attribute or reference is changed
  - EMF objects can be “observed” in order to update views and dependent objects

```
Adapter poobserver = ...  
purchaseOrder.eAdapters().add(poobserver);
```



# Model Change Notification

- Observers or listeners in EMF are called adapters
  - An adapter can also extend class behavior without subclassing
  - For this reason they are typically added using an AdapterFactory

```
PurchaseOrder purchaseOrder = ...
AdapterFactory somePOAdapterFactory = ...
Object poExtensionType = ...

if (somePOAdapterFactory.isFactoryForType(poExtensionType))
{
    Adapter poAdapter = somePOAdapterFactory.adapt(purchaseOrder,
                                                    poExtensionType);
    ...
}
```



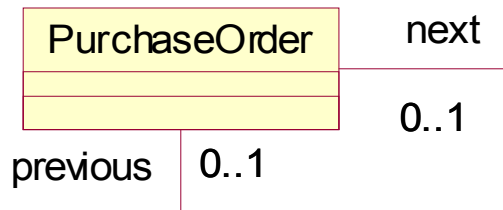
# Model Change Notification

- Efficient notification in “set” methods
  - Checks for listeners before creating and sending notification

```
public String getShipTo()
{
    return shipTo;
}

public void setShipTo(String newShipTo)
{
    String oldShipTo = shipTo;
    shipTo = newShipTo;
    if (eNotificationRequired())
        eNotify(new ENotificationImpl(this, ... ));
}
```

# Bidirectional Reference Handshaking

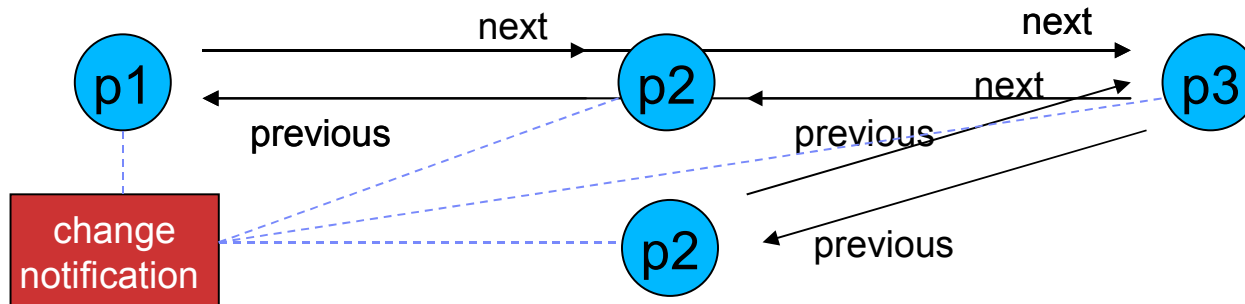


Invariant imposed by the bidirectional reference:

`po.getNext().getPrevious() == po`

```
public interface PurchaseOrder
{
    PurchaseOrder getNext();
    void setNext(PurchaseOrder value);
    PurchaseOrder getPrevious();
    void setPrevious(PurchaseOrder value);
}
```

## Bidirectional Reference Handshaking



```
p1.setNext(p3);
```

# Reflection

- All EMF classes implement interface EObject
- Provides an efficient API for manipulating objects reflectively
  - Used by the framework (e.g., serialization/deserialization, copy utility, generic editing commands, etc.)
  - Also key to integrating tools and applications built using EMF

```
public interface EObject
{
    EClass eClass();
    Object eGet(EStructuralFeature sf);
    void eSet(EStructuralFeature sf, Object val);
    ...
}
```

## Reflection Example

- Setting an attribute using generated API:

```
PurchaseOrder po = ...  
po.setBillTo("123 Elm St.");
```

- Using reflective API:

```
EObject po = ...  
EClass poClass = po.eClass();  
po.eSet(poClass.getEStructuralFeature("billTo"),  
        "123 Elm St.");
```

# Reflective Performance

- Efficient generated switch-based implementation of reflective methods

```
public Object eGet(int featureID, ...)  
{  
    switch (featureID)  
    {  
        case POPackage.PURCHASE_ORDER__SHIP_TO:  
            return getShipTo();  
        case POPackage.PURCHASE_ORDER__BILL_TO:  
            return getBillTo();  
        ...  
    }  
}
```

## Reflection Benefits

- Reflection allows generic access to any EMF model
  - Similar to Java's introspection capability
  - Every EObject (that is, every EMF object) implements the reflection API
- An integrator need only know your model!
- A generic EMF model editor uses the reflection API
  - Can be used to edit any EMF model

## Dynamic EMF

- Ecore models can be defined dynamically in memory
  - No generated code required
  - Dynamic implementation of reflective EObject API provides same runtime behavior as generated code
  - Also supports dynamic subclasses of generated classes
- All EMF model instances, whether generated or dynamic, are treated the same by the framework
- A dynamic Ecore model can be defined by
  - Instantiating model elements with the Ecore API
  - Loading from a .ecore file



## Dynamic EMF Example

- Model definition using the Ecore API

```
EPackage poPackage = EcoreFactory.eINSTANCE.createEPackage();
poPackage.setName("po");
poPackage.setNsURI("http://www.example.com/PurchaseOrder");


EClass poClass = EcoreFactory.eINSTANCE.createEClass();
poClass.setName("PurchaseOrder");
poPackage.getEClassifiers().add(poClass);

EAttribute billTo = EcoreFactory.eINSTANCE.createEAttribute();
billTo.setName("billTo");
billTo.setEType(EcorePackage.eINSTANCE.getEString());
poClass.getEStructuralFeatures().add(billTo);
...

EObject po = EcoreUtil.create(poClass);
po.eSet(billTo, "123 Elm St.");
```

## Exercise 2: EMF Runtime and Static Model APIs

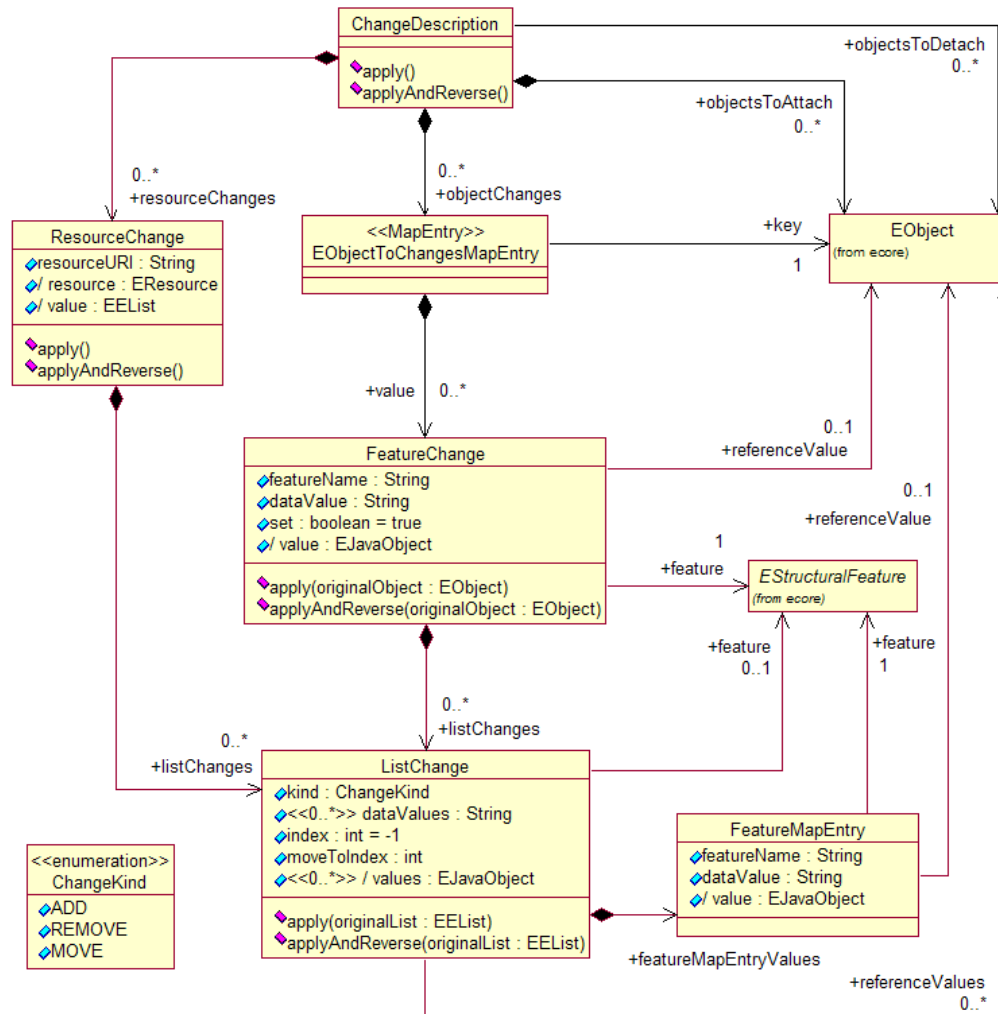
# Agenda

- Demo
- Introduction
  - EMF in a Nutshell
  - EMF Components
  - The Ecore Metamodel
- Exercise 1: Code Generation, Regeneration and Merge
- Exercise 2: EMF Runtime
- 
- ***Exercise 3: Recording Changes***
- Exercise 4: Validation
- Exercise 5: Reflection, Dynamic EMF and XML Processor
- What's New in EMF 2.2
- Summary

# Recording Changes

- EMF provides facilities for recording the changes made to instances of an Ecore model
- Change Model
  - An EMF model for representing changes to objects
  - Directly references affected objects
  - Includes “apply changes” capability
- Change Recorder
  - EMF adapter
  - Monitors objects to produce a change description (an instance of the change model)

# Change Model



## Change Recorder

- Can be attached to EObjects, Resources, and ResourceSets
  - Monitors changes to the objects and their contents trees
- Produces a description of the changes needed to return to the original state (a reverse delta)

```
PurchaseOrder order = ...  
order.setBillTo("123 Elm St.");  
  
ChangeRecorder recorder = new ChangeRecorder();  
recorder.beginRecording(Collections.singleton(order));  
order.setBillTo("456 Cherry St.");  
ChangeDescription change = recorder.endRecording();
```

- Result: a change description with one change, setting billTo to "123 Elm St."

# Applying Changes

- Given a change description, the change can be applied:
  - `ChangeDescription.apply()`
    - consumes the changes, leaving the description empty
  - `ChangeDescription.applyAndReverse()`
    - reverses the changes, leaving a description of the changes originally made (the forward delta)
- The model is always left in an appropriate state for applying the resulting change description

## Example: Transaction Capability


- If any part of the transaction fails, undo the changes

```
ChangeRecorder changeRecorder =  
    new ChangeRecorder(resourceSet);  
  
try  
{  
    // modifications within resource set  
}  
catch (Exception e)  
{  
    changeRecorder.endRecording().apply();  
}
```



## Exercise 3: Recording Changes

# Agenda

- Demo
- Introduction
  - EMF in a Nutshell
  - EMF Components
  - The Ecore Metamodel
- Exercise 1: Code Generation, Regeneration and Merge
- Exercise 2: EMF Runtime
- if time permits
- Exercise 3: Recording Changes
- **Exercise 4: Validation**
- Exercise 5: Reflection, Dynamic EMF and XML Processor
- What's New in EMF 2.2
- Summary

# Validation Framework

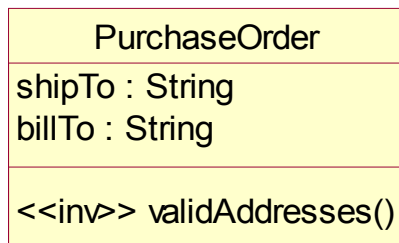
- Model objects validated by external EValidator

```
public interface EValidator
{
    boolean validate(EObject eObject,
                    DiagnosticChain diagnostics, Map Context);
    boolean validate(EClass eClass, EObject eObject,
                    DiagnosticChain, diagnostics, Map context);
    boolean validate(EDatatype eDataType, Object value,
                    DiagnosticChain diagnostics, Map context);
    ...
}
```

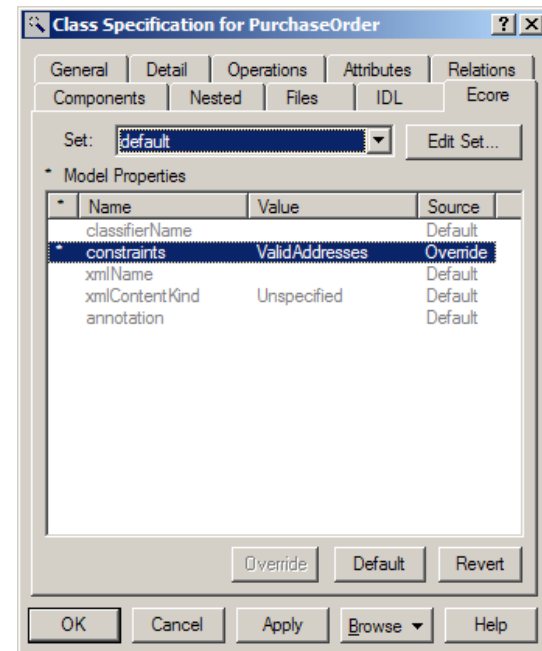
- Detailed results accumulated as Diagnostics
  - Essentially a non-Eclipse equivalent to IStatus
  - Records severity, source plug-in ID, status code, message, other arbitrary data, and nested children

# Invariants and Constraints

- Invariant
  - Defined directly on the class, as an operation with <<inv>> stereotype
  - Stronger statement about validity than a constraint



- Constraint
  - Externally defined for the class via a method on the validator



## Generated EValidator Implementations

- Generated for each package that defines invariants or constraints
- Dispatches validation to type-specific methods
- For classes, a validate method is called for each invariant and constraint
  - Method body must be hand coded for invariants and named constraints

# Schema-Based Constraints

- In XML Schema, named constraints are defined via annotations:

```
<xsd:annotation>  
  <xsd:appinfo source="http://www.eclipse.org/emf/2002/Ecore"  
    ecore:key="constraints">VolumeDiscount</xsd:appinfo>  
</xsd:annotation>
```

- Also, constraints can be defined as facets on simple types, and no additional coding is required
  - Constraint method implementation generated

```
<xsd:simpleType name="SKU">  
  <xsd:restriction base="xsd:string">  
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>  
  </xsd:restriction>  
</xsd:simpleType>
```

# Framework EValidator Implementations

- EObjectValidator validates basic EObject constraints:
  - Multiplicities are respected
  - Proxies resolve
  - All referenced objects are contained in a resource
  - Data type values are valid
- Used as base of generated validators and directly for packages without additional constraints defined

## Framework EValidator Implementations

- Diagnostician walks a containment tree of model objects, dispatching to package-specific validators
  - Diagnostician.validate() is the usual entry point
  - Obtains validators from its EValidator.Registry

```
Diagnostician validator = Diagnostician.INSTANCE;
Diagnostic diagnostic = validator.validate(order);


if (diagnostic.getSeverity() == Diagnostic.ERROR)
{
    // handle error
}

for (Iterator i = diagnostic.getChildren().iterator(); i.hasNext();)
{
    Diagnostic child = (Diagnostic)i.next();
    // handle child diagnostic
}
```



## Exercise 4: Validation

# Agenda

- Demo
- Introduction
  - EMF in a Nutshell
  - EMF Components
  - The Ecore Metamodel
- Exercise 1: Code Generation, Regeneration and Merge
- Exercise 2: EMF Runtime
- if time permits
- Exercise 3: Recording Changes
- Exercise 4: Validation
- ***Exercise 5: Reflection, Dynamic EMF and XML Processor***
- What's New in EMF 2.2
- Summary


## XML Processor

- Simplified API for loading and saving XML
  - Handles resource set, registries, etc. under the covers
- Can automatically create a dynamic Ecore representation of a schema
  - Load/save instance documents without generating code
  - Manipulate the objects using reflective EObject API

```
URI schemaURI = ...  
String instanceFileName = ...  
  
XMLProcessor processor = new XMLProcessor(schemaURI);  
Resource resource = processor.load(instanceFileName);  
  
EObject documentRoot = (EObject)resource.getContents().get(0);
```

## Exercise 5: Reflection, Dynamic EMF and XML Processor


# Agenda

- Demo
- Introduction
  - EMF in a Nutshell
  - EMF Components
  - The Ecore Metamodel
- Exercise 1: Code Generation, Regeneration and Merge
- Exercise 2: EMF Runtime
- if time permits
- Exercise 3: Recording Changes
- Exercise 4: Validation
- Exercise 5: Reflection, Dynamic EMF and XML Processor
- ***What's New in EMF 2.2***
- Summary

## Important Changes in EMF 2.2

- Content adapter for managing reverse of 1-way references
- Cross-resource containment
- XMI 2.1 support
- Model exporter
- Improve XML Schema generation
- Improve code generation error reporting and handling
  
- Performance optimizations
- For more, see:
  - <http://www.eclipse.org/emf/docs.php#plandocs>
  - <http://www.eclipse.org/emf/news/>

# Agenda

- Demo
  - Introduction
    - EMF in a Nutshell
    - EMF Components
    - The Ecore Metamodel
  - Exercise 1: Code Generation, Regeneration and Merge
  - Exercise 2: EMF Runtime
- 
- Exercise 3: Recording Changes
  - Exercise 4: Validation
  - Exercise 5: Reflection, Dynamic EMF and XML Processor
- What's New in EMF 2.2
  - ***Summary***

# Summary

- EMF is low-cost modeling for the Java mainstream
- Boosts productivity and facilitates integration
- Mixes modeling with programming to maximize the effectiveness of both



# Summary

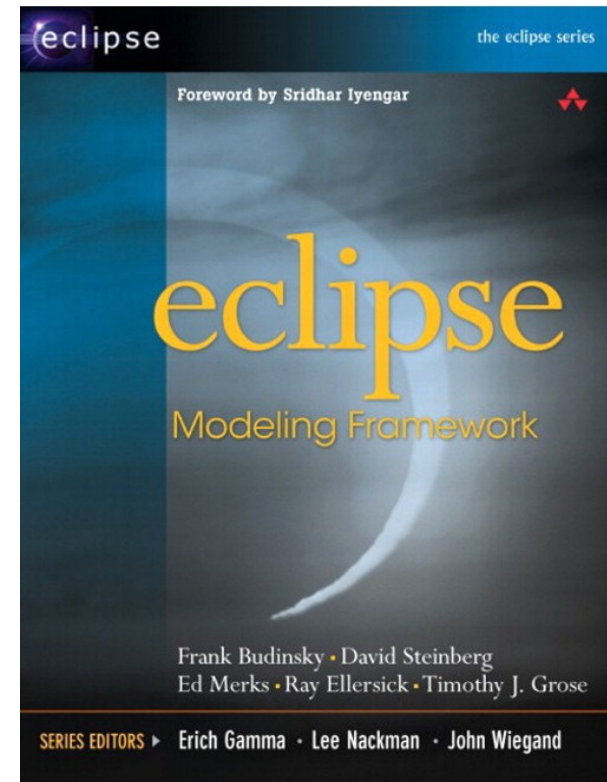
- EMF provides...
  - A metamodel (Ecore) with which your domain model can be specified
    - Your model can be created from UML, XML Schema or annotated Java interfaces
  - Generated Java code
    - Efficient and straightforward
    - Code customization preserved
  - Persistence and Serialization
    - Resource-based serialization
    - Proxy resolution and demand loading
    - Default resource implementation is XMI (XML metadata interchange), but can be overridden

# Summary

- EMF provides...
  - Model change notification is built in
    - Just add adapters (observers) where needed
  - Reflection and dynamic EMF
    - Full introspection capability
  - Simple change recording and roll-back
  - Extensible validation framework
  - Standalone runtime support
  - A UI-independent layer for viewing and editing modeled data (EMF.Edit)

## Resources

- **This presentation and related workshop materials**
    - <http://www.eclipse.org/emf/docs/presentations/OOPSLA/>
- 
- EMF documentation in Eclipse Help
    - Overviews, tutorials, API reference (javadoc)
  - EMF Project Web Site
    - <http://www.eclipse.org/emf/>
    - Overviews, tutorials, newsgroup, Bugzilla
  - Eclipse Modeling Framework by Frank Budinsky et al.
    - Addison-Wesley; 1st edition (August 13, 2003)
    - ISBN: 0131425420.



## Legal Notices

IBM, Rational, WebSphere, Lotus, and Rational Rose are registered trademarks of International Business Machines Corp. in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.