eclipseCON™ 2008

Fundamentals of the Eclipse Modeling Framework

Dave Steinberg
IBM Rational Software
Toronto, Canada
EMF Committer

Goal

Learn about modeling and how the Eclipse Modeling Framework can help you write your application in significantly less time, simply by leveraging the data model you've probably already defined (although you might not know it yet)

Agenda

- **What is Modeling?**
- Defining a Model with EMF
- EMF Architecture
- Code Generation
- Programming with EMF
- Summary

Model Driven Architecture (MDA)

- A software architecture proposed by the OMG (Object Management Group)
- Application specified in high-level, Platform Independent Model (PIM)
- Transformation technologies used to convert PIM to Platform Specific Model (PSM), implementation code
- Includes several open modeling standards:
  - ◆ UML™ (Unified Modeling Language)
  - ◆ MOF (Meta-Object Facility)
  - ◆ XMI (XML Metadata Interchange)
  - ◆ CWM (Common Warehouse Model)

MDA – Are We There Yet?

- Doubts exist about the ability of MDA to deliver on its promises:
    - ◆ Ambitiousness of vision
    - ◆ Model expressiveness vs. complexity
    - ◆ Availability of implementations ("vaporware")

Enter EMF!

- EMF is a simple, pragmatic approach to modeling:
  - ◆ Allows us to generate some of the code that we write over and over, paving the way for more complex systems (including more ambitious MDA tools)
  - ◆ Models are simple, but meant to be mixed with hand-written code
  - ◆ It's real, proven technology (since 2002)

Model Driven Development with EMF

- Contrary to the belief of many programmers, modeling is useful for more than just documentation
- Almost every program we write manipulates some data model
  - Defined using UML, XML Schema, some other definition language, or implicitly in Java™
- EMF aims to extract this intrinsic "model" and generate some of the implementation code
  - Can be a tremendous productivity gain

EMF at Eclipse.org

- Foundation for the Eclipse Modeling Project
  - ◆ EMF project incorporates core and additional mature components: Query, Transaction, Validation
  - ◆ EMF Technology project incubates complementary components: CDO, Teneo, Compare, Search, Temporality, Ecore Tools…
  - ◆ Other projects build on EMF: Graphical Modeling Framework (GMF), Model Development Tools (MDT), Model to Model Transformation (M2M), Model to Text Transformation (M2T)…

- Other uses: Web Tools Platform (WTP), Data Tools Platform (DTP), Business Intelligence and Reporting Tools (BIRT), SOA Tools Platform (STP)…
- Large open source user community

Agenda

- What is Modeling?
- Defining a Model with EMF
- EMF Architecture
- Code Generation
- Programming with EMF
- Summary

What is an EMF "Model"?

- Specification of an application's data
  - Object attributes
  - Relationships (associations) between objects
  - Operations available on each object
  - Simple constraints (e.g. multiplicity) on objects and relationships
- Essentially, the Class Diagram subset of UML

Model Sources

- EMF models can be defined in (at least) three ways:
    1. Java Interfaces
    2. UML Class Diagram
    3. XML Schema
- Choose the one matching your perspective or skills and EMF can create the others, as well as the implementation code
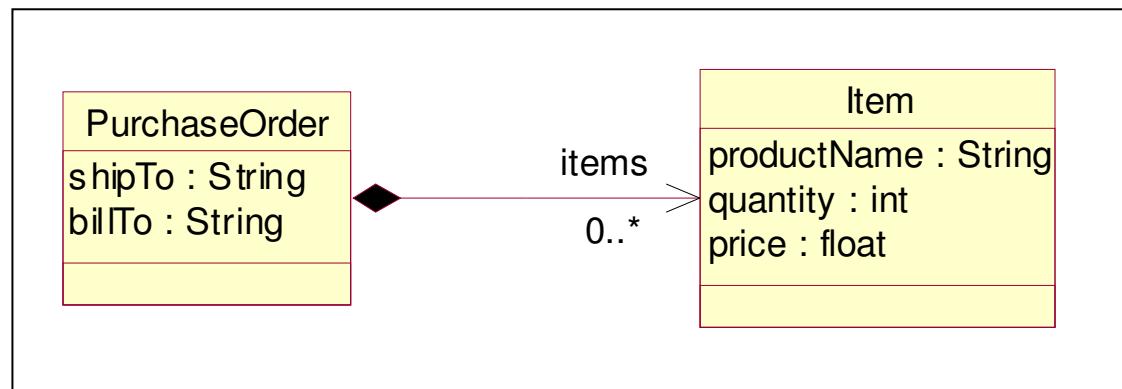
Java Interfaces

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

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

• Classes can be defined completely by a subset of members, supplemented by annotations

UML Class Diagram

- Built-in support for Rational Rose®
- UML2 support available with UML2 (from MDT)

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>
```
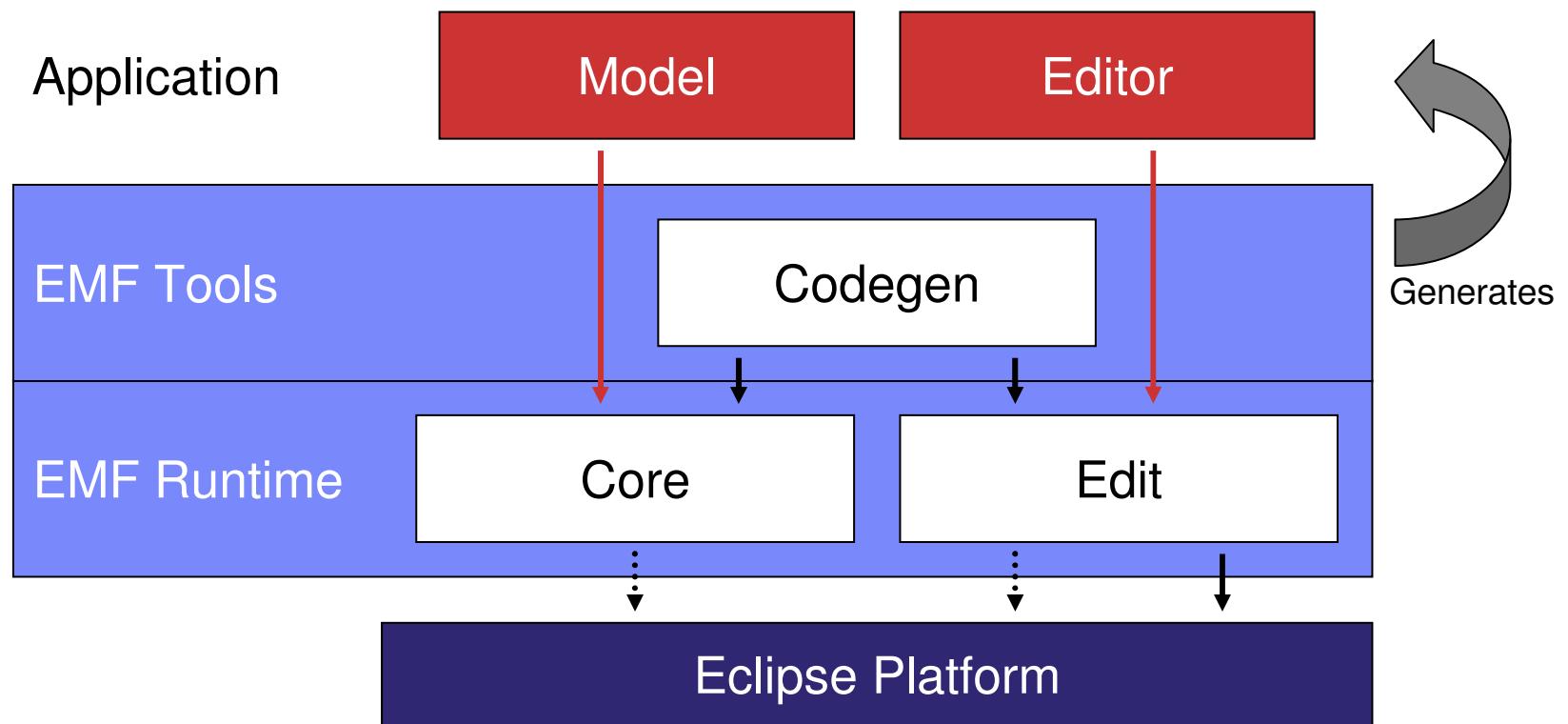
Unifying UML, XML and Java

- All three forms provide the same information
  - ◆ Different visualization/representation
  - ◆ The application's data "model" or structure
  - ◆ Model importers can be added for different model representations (e.g. RDB Schema)
- From a model definition, EMF can generate…
  - ◆ Java implementation code, including UI
  - ◆ XML Schemas
  - ◆ Eclipse "plug-in" artifacts

Agenda

- What is Modeling?
- Defining a Model with EMF
- EMF Architecture
- Code Generation
- Programming with EMF
- Summary

EMF Architecture

Fundamentals of the Eclipse Modeling Framework | Copyright © IBM Corp., 2005-2008
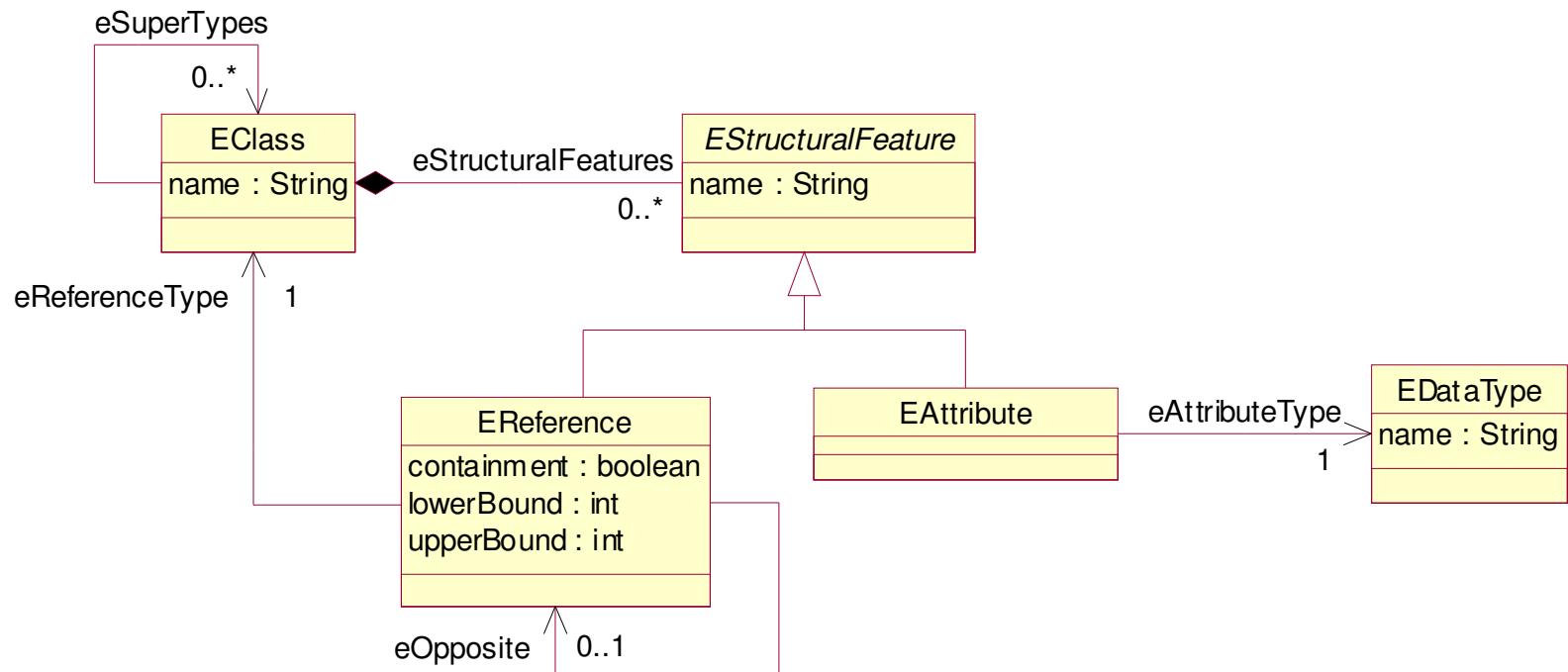
EMF Components

- ## Core Runtime
  - ◆ Notification framework
  - ◆ Ecore metamodel
  - ◆ Persistence (XML/XMI), validation, change model

- ## EMF.Edit
  - ◆ Support for model-based editors and viewers
  - ◆ Default reflective editor

- ## Codegen
  - ◆ Code generator for application models and editors
 - ◆ Extensible model importer/exporter framework
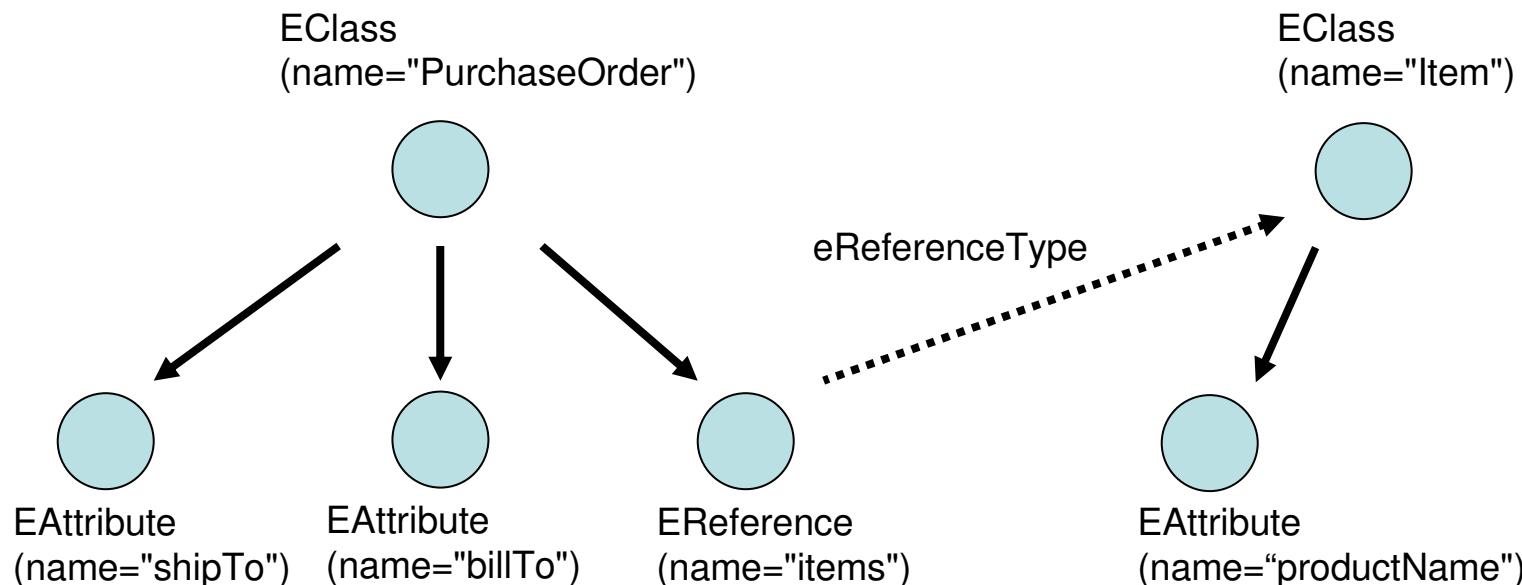
Ecore

- EMF's metamodel (model of a model)

Ecore

- Application models (e.g. purchase order model) are instances of Ecore

EClass
(name="PurchaseOrder")

EClass
(name="Item")

eReferenceType

EAttribute
(name="shipTo")

EAttribute
(name="billTo")

EReference
(name="items")

EAttribute
(name="productName")
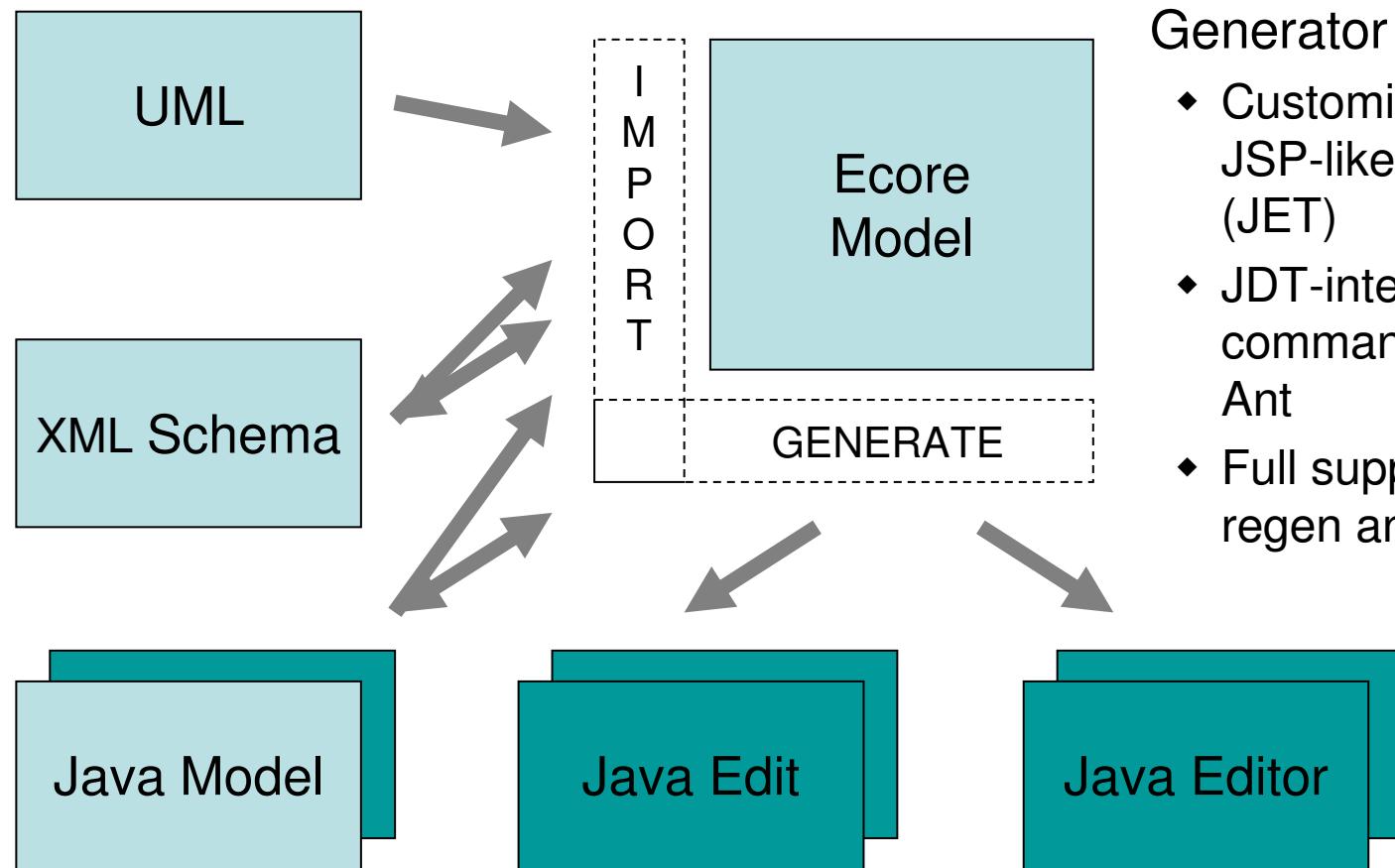
Ecore

- Persistent format is XMI (.ecore file)

```xml
<eClassifiers xsi:type="ecore:EClass"
  name="PurchaseOrder">
 <eStructuralFeatures xsi:type="ecore:EReference"
   name="items" eType="#//Item"
   upperBound="-1" containment="true"/>
 <eStructuralFeatures xsi:type="ecore:EAttribute"
   name="shipTo"
   eType="ecore:EDataType http:...Ecore#//EString"/>
 ...
</eClassifiers>
```

- Alternate format is Essential MOF XMI (.emof file)

Model Import and Generation

UML

XML Schema

IMPORT

Ecore Model

GENERATE

Java Model

Java Edit

Java Editor
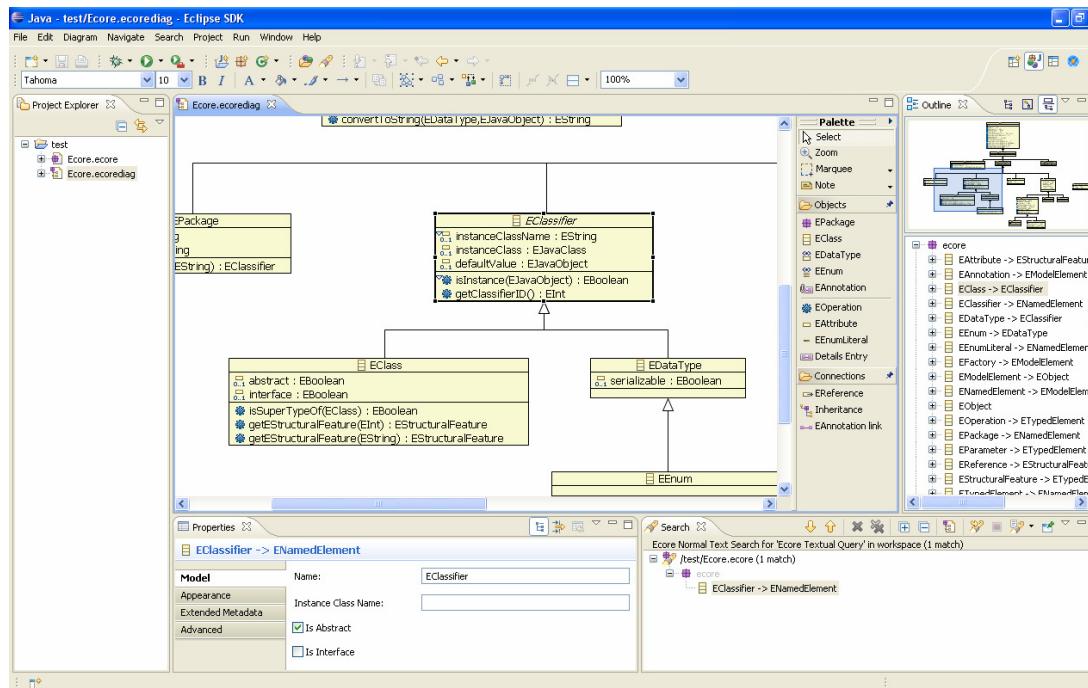
Generator features:
- ◆ Customizable, JSP-like templates (JET)
- ◆ JDT-integrated, command-line or Ant
- ◆ Full support for regen and merge

Direct Ecore Modeling

- Ecore models can be created directly
  - ◆ Sample Ecore editor (in EMF)
  - ◆ Ecore Tools graphical editor (from EMFT)

Fundamentals of the Eclipse Modeling Framework | Copyright © IBM Corp., 2005-2008

EMF and Java 5.0

- EMF 2.3 began exploiting the capabilities of Java 5.0
  - Generics
  - Typesafe enums
  - Annotations
  - Enhanced for loop
  - Autoboxing

- EMF 2.3 and later provide 5.0-targeted runtime that only runs on a Java 5.0 or later VM
- The code generator can also produce application code that runs on a 1.4 VM against the 2.2 runtime
- This is a recognition that Java 5.0 is the way forward, but the transition need not be painful for the user

Agenda

- What is Modeling?

- Defining a Model with EMF

- EMF Architecture

- Code Generation

- Programming with EMF

- Summary

Generated Model Code
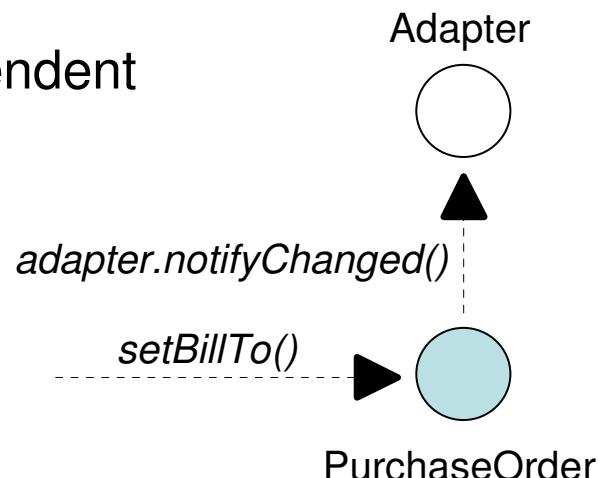
- Interface and implementation for each modeled class
  - ◆ Includes get/set accessors for attributes and references

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

```
public class PurchaseOrderImpl extends EObjectImpl
  implements PurchaseOrder
{
  ...
}
```

Change Notification

- Every EObject is also a notifier
    - Sends notification whenever an attribute or reference is changed
    - Observers can update views, dependent objects
    - Observers are also adapters

Adapter

adapter.notifyChanged()

setBillTo()

PurchaseOrder

```
Adapter adapter = ...
purchaseOrder.eAdapters().add(adapter);
```
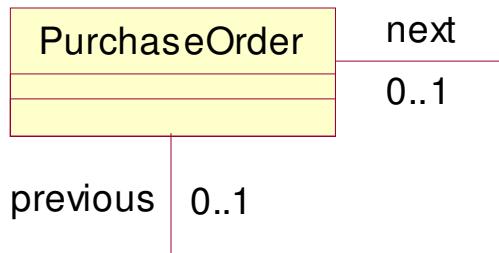
Change Notification: Feature Accessors

- Efficient notification from set methods

```
public String getBillTo()
{
  return billTo;
}


public void setBillTo(String newBillTo)
{
  String oldBillTo = BillTo;
  billTo = newBillTo;
  if (eNotificationRequired())
    eNotify(new ENotificationImpl(this, ... ,
                                  oldBillTo, billTo);
}
```

Bidirectional References

PurchaseOrder	next
	0..1
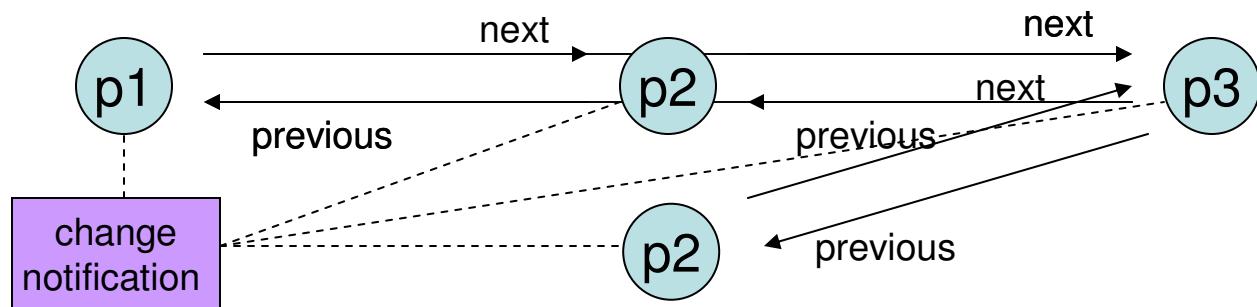
previous 0..1

Bidirectional reference imposes invariant:

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

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

Bidirectional Reference Handshaking

- Multi-step procedure implemented using InternalEObject methods:
    - ◆ eInverseRemove()
    - ◆ eInverseAdd()

- Framework list implementations and generated setters directly invoke them on target objects as needed

- Notifications accumulated using a NotificationChain and fired off at the end
    - ◆ Not used to implement the handshaking

Bidirectional Reference Handshaking


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

Typesafe Enums

- Typesafe enum pattern for enumerated types

```java
public enum Status implements Enumerator
{
  PENDING(0, "Pending", "Pending"),
  BACK_ORDER(1, "BackOrder", "BackOrder"),
  COMPLETE(2, "Complete", "Complete");

  public static final int PENDING_VALUE = 0;
  public static final int BACK_ORDER_VALUE = 1;
  public static final int COMPLETE_VALUE = 2;

  private final int value;
  private final String name;
  private final String literal;

  private Status(int value, String name, String literal)
  { ... } ...
}
```

Factories and Packages

- Factory to create instances of model classes

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

- Package provides access to metadata

```
POPackage poPackage = POPackage.eINSTANCE;
EClass itemClass = POPackage.Literals.ITEM;
  //or poPackage.getItem()

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

Fundamentals of the Eclipse Modeling Framework | Copyright © IBM Corp., 2005-2008

Reflective EObject API

- All EMF classes implement EObject interface
- Provides an efficient API for manipulating objects reflectively
  - ◆ Used by framework (e.g. persistence framework, copy utility, editing commands)
  - ◆ Key to integrating EMF-based tools and applications

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

Reflective EObject API

- Efficient generated switch-based implementation of reflective methods

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

Summary of Generated Artifacts

- Model
  - Interfaces and classes
  - Type-safe enums
  - Package (metadata)
  - Factory
  - Switch utility
  - Adapter factory base
  - Validator
  - Custom resource
  - XML Processor

- Manifests, plug-in classes, properties, icons…

- Edit (UI Independent)
  - Item providers
  - Item provider adapter factory

- Editor
  - Model Wizard
  - Editor
  - Action bar contributor
  - Advisor (RCP)

- Tests
  - Test cases
  - Test suite
  - Stand-alone example

Regeneration and Merge

- The EMF generator is a merging generator

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

- @generated elements are replaced/removed
- To preserve changes mark @generated NOT

Regeneration and Merge

- Generated methods can be extended through redirection

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

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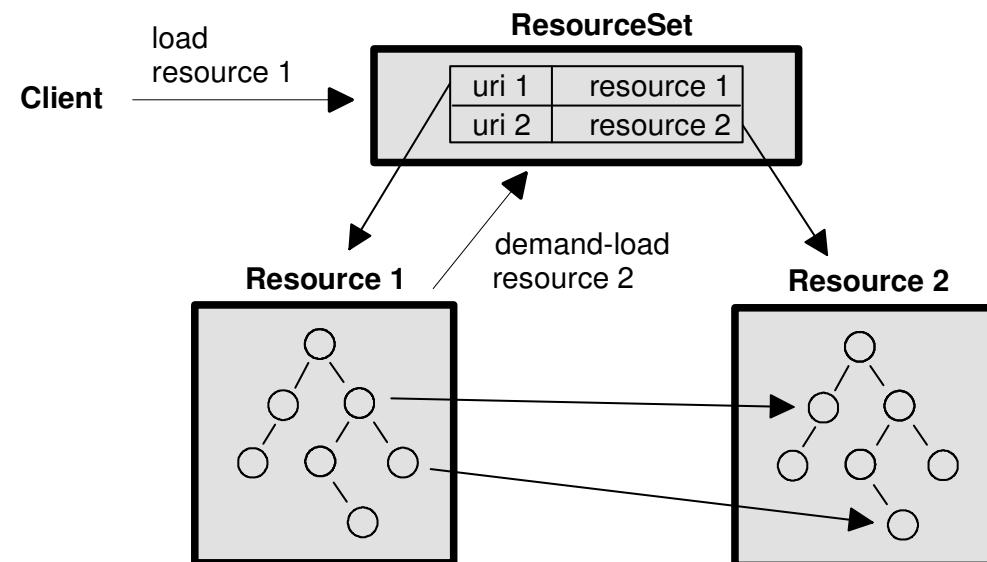
Agenda

- What is Modeling?

- Defining a Model with EMF

- EMF Architecture

- Code Generation

- Programming with EMF

- Summary

Persistence

- Persisted data is referred to a resource
- Objects can be spread out among a number of resources, within a resource set
- Proxies represent referenced objects in other resources

Resource Set

- Context for multiple resources that may have references among them

- Usually just an instance of ResourceSetImpl

- 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

Registries

- **Resource Factory Registry**
  - ◆ Returns a resource factory for a given type of resource
  - ◆ Based on URI scheme, filename extension or content type
  - ◆ Determines the format for save/load
  - ◆ If no registered resource factory found locally, delegates to global registry: Resource.Factory.Registry.INSTANCE

- **Package Registry**
  - ◆ Returns the package identified by a given namespace URI
  - ◆ Used during loading to access factory for instantiating classes
  - ◆ If no registered package found locally, delegates to global registry: EPackage.Registry.INSTANCE

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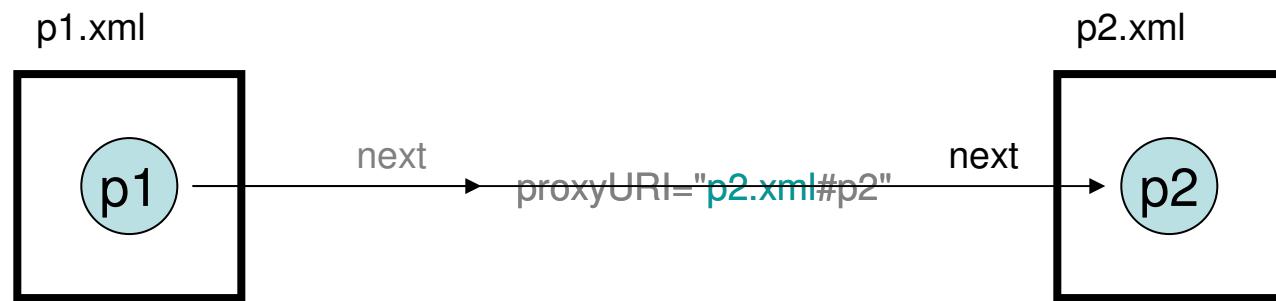
- Container for objects that are to be persisted together
  - ◆ Convert to and from persistent form via save() and load()
  - ◆ Access contents of resource via getContents()

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

- EMF provides generic XMLResource implementation
 - ◆ Other, customized implementations, too (e.g. XMI, EMOF)

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

Proxy Resolution and Demand Load

p1.xml

p2.xml

next

next

p1 → proxyURI="p2.xml#p2" → p2
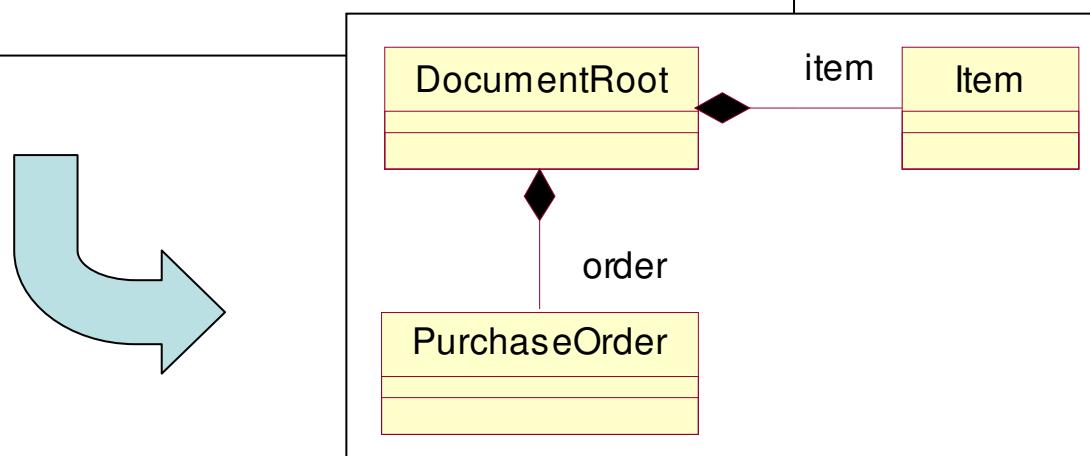
```
PurchaseOrder p2 = p1.getNext();
```

XML Schema-Conformant Serialization

- A DocumentRoot class is included in models created from XML Schema
  - ◆ Includes one containment reference per global element

```
<xsd:schema ... >
  <xsd:element name="order" type="po:PurchaseOrder">
  <xsd:element name="item" type="po:Item">
  ...
</xsd:schema>
```

XML Schema-Conformant Serialization

- To serialize an instance document
  - Use an XMLResource with extended metadata enabled (e.g. use the resource factory generated with your model)
  - Use an instance of the DocumentRoot in the resource's contents
  - Set the appropriate reference to select the root element

```
PurchaseOrder p1 = ...

URI uri = URI.createFileURI("C:/data/order.po");
resource resource = rs.createResource(uri);
DocumentRoot root = POFactory.eINSTANCE.createDocumentRoot();
resource.getContents().add(root);
root.setOrder(p1);
resource.save(null);
```

Reflection

- Setting an attribute using generated API

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

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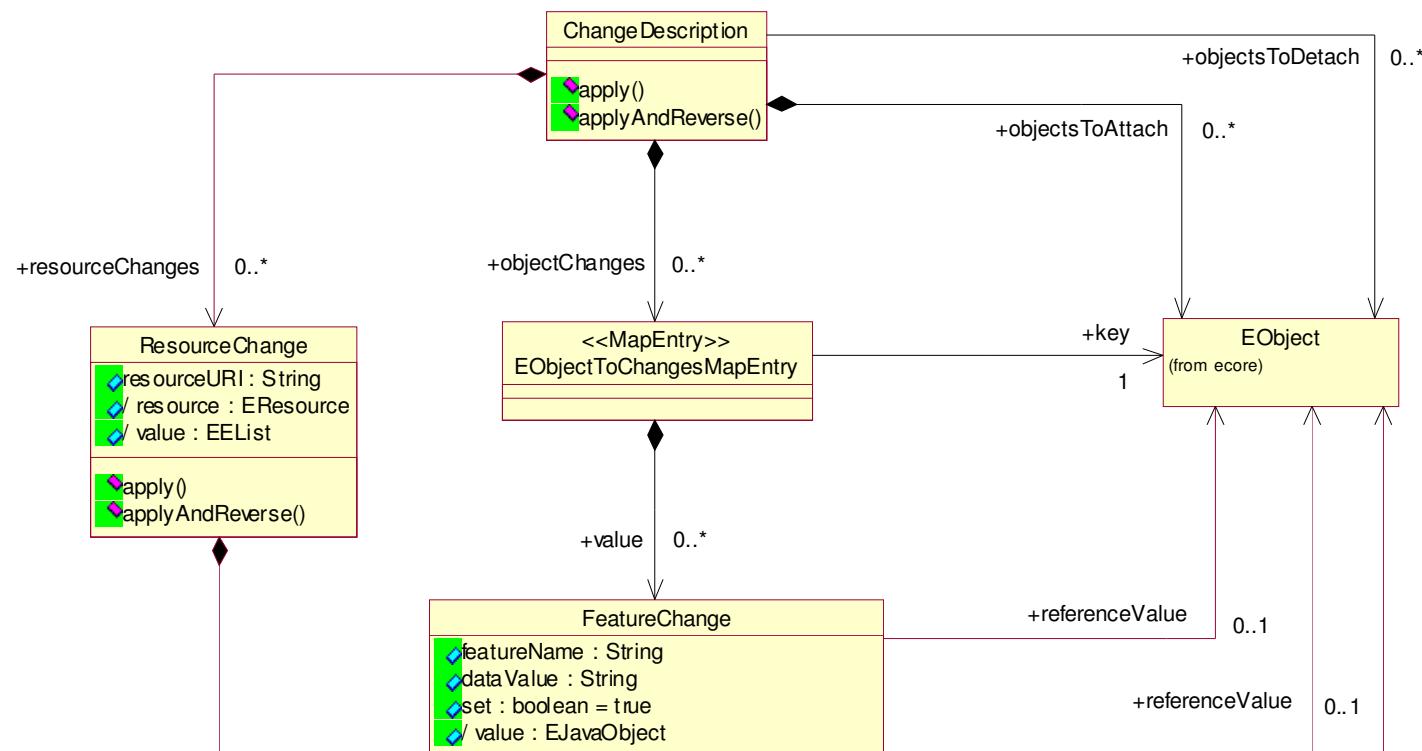
```
EObject po = ...
EClass poClass = po.eClass();
po.eSet(poClass.getEStructuralFeature("billTo"),
        "123 Elm St.");
```

Fundamentals of the Eclipse Modeling Framework | Copyright © IBM Corp., 2005-2008

Dynamic EMF

- An Ecore model can also be defined at runtime using Ecore API or loaded from persistent form
  - No generated code required
  - Dynamic implementation of reflective EObject API provides same runtime behavior as generated code
  - Can create dynamic subclasses of generated classes
- The framework treats all model instances the same, whether generated or dynamic

Change Recording

- The change model represents changes to instances of any EMF model

Change Recording

- Change recorder
  - Adapter that creates change description based on notifications, to describe reverse delta
  - Provides transaction capability

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

Validation

- Models can define named constraints and invariants for batch validation
    - ◆ Invariant
        - ▪ Defined directly on class, as <<inv>> operation
        - ▪ Stronger statement about validity than a constraint
    - ◆ Constraint
        - ▪ Externally defined via a validator

- Invariants and constraints are invoked by a validator, which is generated for a package, if needed
    - ◆ Bodies of invariants and constraints are usually hand-coded

Automatically Implemented Constraints

- EObjectValidator validates basic constraints on all EObjects (multiplicity, data type values, etc.)
  - ◆ Used as base of generated validators and directly for packages without additional constraints defined

- In XML Schema, constraints can be defined as facets on simple types
  - ◆ Full implementation generated in validator

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

Diagnostician

- Diagnostician walks a containment tree of objects, dispatching to package-specific validators
  - Diagnostician.validate() is the usual entry point
  - Detailed results accumulated as Diagnostics

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

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

for (Diagnostic child : diagnostic.getChildren())
{
  // handle child diagnostic
}
```

EMF Utilities

- Class EcoreUtil provides various static convenience utilities for working with EObjects

  ◆ Copying

  ```
  EObject copy = EcoreUtil.copy(original);
  ```

  ◆ Testing equality

  ```
  boolean equal = EcoreUtil.equals(eObject1, eObject2);
  ```

  ◆ Also, cross referencing, contents/container navigation, annotation, proxy resolution, adapter selection, and more…

Agenda
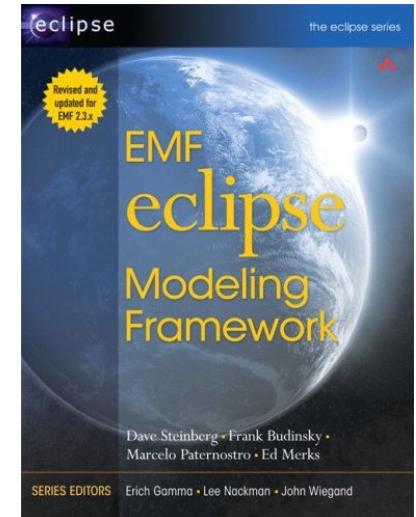
- What is Modeling?
- Defining a Model with EMF
- EMF Architecture
- Code Generation
- Programming with EMF
- Summary

Summary

- EMF is low-cost modeling for the Java mainstream
- Leverages the intrinsic model in an application
  - ◆ No high-level modeling tools required
- Mixes modeling with programming to maximize the effectiveness of both
- Boosts productivity and integrates integration
- The foundation for model-driven development and data integration in Eclipse

Resources

- EMF documentation in Eclipse Help
  - ◆ Overviews, tutorials, API reference

- EMF project Web site
  - ◆ http://www.eclipse.org/modeling/emf/
  - ◆ Downloads, documentation, FAQ, newsgroup, Bugzilla, Wiki

- Eclipse Modeling Framework, by Frank Budinsky et al.
  - ◆ ISBN: 0131425420
  - ◆ Rough cut of second edition available
  - ◆ http://my.safaribooksonline.com/9780321331885

Legal Notices