Introduction to the Eclipse Modeling Framework

Elena Litani and Marcelo Paternostro
IBM Rational Software
Toronto, Canada
EMF and XSD Projects
Agenda

- **EMF in a nutshell**
- EMF Components
- The Ecore Metamodel
- Model Definition
- Code Generation, Regeneration and Merge
- EMF Runtime Framework
- Change Model
- Validation Framework
- EMF.Edit Framework
- Service Data Objects (SDO)
- Summary and Q&A
What is EMF

- A modeling & data Integration framework for Eclipse

What is an EMF “model” (Ecore)?

- A general model of models from which any model can be defined
- Specification of an application’s data
  - Object attributes
  - Relationships (Associations) between objects
  - Operations available on each object
  - Simple constraints (ex: cardinality) on objects and relationships
- Essentially the Class Diagram subset of UML
What does EMF Provide?

- From a model definition -- Java code, XML documents, XML Schemas -- EMF can generate efficient, correct, and easily customizable implementation code

- EMF converts your models to Ecore (EMF Meta Model)

- Tooling support within the Eclipse framework (or command line), including support for generating Eclipse-base and RCP editors

- Reflective and dynamic model invocation

- Supports XML/XMI (de) serialization of instances of a model

- And more….
Why EMF?

- EMF is middle ground in the modeling vs. programming world
  - Focus is on class diagram subset of UML modeling (object model)
  - Transforms models into efficient, correct, and easily customizable Java code
  - Provides the infrastructure to use models effectively in your code

- Very low cost of entry
  - Full scale graphical modeling tool not required
  - EMF is free
  - Small subset of UML
EMF History

- Originally based on MOF (Meta Object Facility)
  - From OMG (Object Management Group)
  - Abstract language and framework for specifying, constructing, and managing technology neutral meta-models
- EMF evolved based on experience supporting a large set of tools
  - Efficient Java implementation of a practical subset of the MOF API
- Foundation for model based WebSphere Studio family product set
  - Example: J2EE model in WebSphere Studio Application Developer
- 2003: EMOF defined (Essential MOF)
  - Part of MOF 2 specification; UML2 based; recently approved by OMG
  - EMF is approximately the same functionality
    - Significant contributor to the spec; adapting to it
- 2004: Service Data Objects (SDO) reference implementation
  - Currently being standardized through the Java Community Process
    - JSR 235
Who is using EMF today?

- IBM WebSphere/Rational product family
- Other Eclipse projects (XSD, UML2, VE, Hyades)
- ISV’s (TogetherSoft, Ensemble, Versata, Omondo, and more)
- SDO reference implementation
- Large open source community
  - O(1K) downloads/day
  - and growing …
EMF Model Sources

- EMF models can be defined in (at least) three ways:
  1. Java Interfaces
  2. UML models expressed in Rose files
  3. XML Schema

- Choose the one matching your perspective or skills
EMF Model Sources (cont)

1. Java Interfaces

```java
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);
}
```
EMF Model Sources (cont)

2. UML Class Diagram

```
<table>
<thead>
<tr>
<th>PurchaseOrder</th>
</tr>
</thead>
<tbody>
<tr>
<td>shipTo : String</td>
</tr>
<tr>
<td>billTo : String</td>
</tr>
</tbody>
</table>

Item
- productName : String
- quantity : int
- price : float

items 0..*
3. XML Schema

```xml
<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>
```
EMF Model Definition (cont)

Unifying Java™, XML, and UML 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-ins
A Typical EMF Usage Scenario

1. Create EMF model
   - Import UML (e.g. Rational Rose .mdl file)
   - Import XML Schema
   - Import annotated Java interfaces
   - Create Ecore model directly using EMF Ecore editor or Omondo’s (free) EclipseUML graphical editor

2. Generate Java code for model

3. Iteratively develop Java application

4. Refine model; regenerate Java code

5. Prime the model with instance data using generated EMF model editor

6. Use EMF.Edit to build customized user interface
Agenda

- EMF in a nutshell
- **EMF Components**
- The Ecore Metamodel
- Model Definition
- Code Generation, Regeneration and Merge
- EMF Runtime Framework
- Change Model
- Validation Framework
- EMF.Edit Framework
- Service Data Objects (SDO)
- Summary and Q&A
EMF Components

- **EMF Core**
  - Ecore meta model
  - Model change notification
  - 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
  - Model importers from Rose, XML, or Java interfaces
EMF Tooling
Model Import and Generation

Generator features:
- Customizable JSP-like templates (JET)
- Command-line or integrated with Eclipse JDT
- Fully supports regeneration and merge

* requires Eclipse to run
Model Import

- Import from
  - UML
    - Rational Rose .mdl file directly supported
    - Other UML editors possible
  - Annotated Java
    - Consists of Java interfaces for each class model
    - Annotations using @model tags added to interface to express model definition not possible with code
    - Lowest cost approach
  - XML Schema
    - May produce more complicated EMF model than from Java or UML
  - Ecore model (*.ecore file)
    - Just creates the generator model (discussed later)
Model Creation

- Ecore model created within an Eclipse project via wizard using a sources described in previous slide

Output is:

- `modelname.ecore` file
  - Ecore model file in XMI format
  - Canonical form of the model
- `modelname.genmodel` file
  - A “generator model” for specifying generator options
  - Decorates `.ecore` file
  - EMF code generator is an EMF `.genmodel` editor
  - `.genmodel` and `.ecore` files automatically kept in sync
Ecore Model Editor

- A generated (and customized) EMF editor for the Ecore model
- Models can be edited using tree view in conjunction with property view
  - New components (EClass, EAttribute, EReference, etc.) created using popup actions in tree view
  - Set names, etc., in property view
- Note: a graphical editor (e.g., Omondo) is better approach
Generator Model Editor

- Looks nearly the same as the Ecore model editor
- Kept in sync with changes to .ecore file
- Context menu actions generate code
  1. The **Generate Model Code** action produces Java code to implement the model
  2. The **Generate Edit Code** action produces adapter code to support viewers
  3. The **Generate Editor Code** action produces a full function Eclipse editor
  4. The **Generate All** action produces all three
- Generation options expressed in Properties view
- Command line API also available
Agenda

- EMF in a nutshell
- EMF Components
- The Ecore Metamodel
- Model Definition
- Code Generation, Regeneration and Merge
- EMF Runtime Framework
- Change Model
- Validation Framework
- EMF.Edit Framework
- Service Data Objects (SDO)
- Summary and Q&A
The Ecore (Meta) Model

- Ecore is EMF’s model of a model (metamodel)
  - Persistent representation is XMI
Ecore Meta Model

EObject is the root of every model object. Equivalent to java.lang.Object
# Partial List of Ecore Data Types

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

*Note: Ecore datatypes are serializable; support for custom datatypes*
PurchaseOrder Ecore Model

EClass (name="PurchaseOrder")

EAttribute (name="shipTo")
EAttribute (name="billTo")
EReference (name="items")

EClass (name="Item")

eReferenceType

EAttribute (name="productName")
PurchaseOrder Ecore XMI

```xml
<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
  - Part of OMG MOF 2 Standard
Agenda

- EMF in a nutshell
- EMF Components
- The Ecore Metamodel
- Model Definition
- Code Generation, Regeneration and Merge
- EMF Runtime Framework
- Change Model
- Validation Framework
- EMF.Edit Framework
- Service Data Objects (SDO)
- Summary and Q&A
The EMF Subset of UML

Classes*, Abstract classes, and Interfaces

<table>
<thead>
<tr>
<th>ClassOrInterfaceName</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute1 : type1</td>
</tr>
<tr>
<td>attribute2 : type2 = initval</td>
</tr>
<tr>
<td>&lt;&lt;0..*&gt;&gt; attribute3 : type3</td>
</tr>
<tr>
<td>operation1(arg1 : type1) : return1</td>
</tr>
<tr>
<td>operation2(arg1 : type1, arg2 : type2) : return2</td>
</tr>
</tbody>
</table>

Attributes and operations

* EMF classes correspond to both an interface and corresponding implementation class in Java
The EMF Subset of UML (cont)

One-way Associations

Bi-directional Associations
The EMF Subset of UML (cont)

Class Inheritance

Containment Associations
Enumerations and Datatypes

<<enumeration>>
EnumName

| literal1 | literal2 | literal3 = value |

<<datatype>>
DataTypeName

<<javaclass>> com.example.JavaClass1
An Example Model

© Copyright IBM Canada Ltd., 2005. All rights reserved.
Java Annotations

- Imbedded as Java comments before classes and methods
- **@model** tag specifies model metadata not contained in Java interface code
- Form
  ```java
  @model [property = "value" | property = "value"] ...
  ```
- Example: Containment relationship between Library and Book classes
  ```java
  public interface Library extends EObject {
  /**
   * @model type="Book" containment="true"
   * @generated
   */
  EList getBooks()
  ```
## Sampling of @model Annotations

<table>
<thead>
<tr>
<th>Ecore Object</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>EClass</td>
<td>abstract, interface</td>
</tr>
<tr>
<td>EAttribute</td>
<td>defaultValue, unique, many, dataType, changeable, required, upperBound, lowerBound, unsettable …</td>
</tr>
<tr>
<td>EReference</td>
<td>opposite, containment, unique, many, changeable, required lowerBound, upperBound, unsettable, …</td>
</tr>
<tr>
<td>EOperation</td>
<td>parameters, dataType</td>
</tr>
<tr>
<td>EDataType</td>
<td>instanceClass, serializable</td>
</tr>
</tbody>
</table>
Agenda

- EMF in a nutshell
- EMF Components
- The Ecore Metamodel
- Model Definition
- **Code Generation, Regeneration and Merge**
- EMF Runtime Framework
- Change Model
- Validation Framework
- EMF.Edit Framework
- Service Data Objects (SDO)
- Summary and Q&A
Generated Java Code

- EMF framework is lightweight
  - Generated code is clean, simple, efficient
- Model Generator produces
  - Interface classes for every modeled object
  - Includes get/set methods for all object attributes
  - Implementation classes for every interface
  - Support for model change notification
  - A factory class to create instances of our model objects
  - A package class that provides access to models metadata
Generated Java Code

View support code is divided into two parts:

- UI-independent code (placed in an edit plug-in)
  - Item providers (adapters)
- UI-dependent code (placed by default in a separate editor plug-in)
  - Model editor
  - Model creation wizard
Regeneration and Merge

- Hand-written code can be added to generated code and preserved during regeneration
- All generated classes, interfaces, methods include `@generated` marker
- Override generated code by removing `@generate` marker
  - or include additional text like `@generated NOT`
- Methods without `@generated` marker are left alone during regeneration
- If duplicate methods names exist your code takes precedence
  - generated code is discarded
Regeneration and Merge

- Extend (vs. replace) generated method through redirection
- To override the `getQuantity` generated method:
  - Add suffix `Gen` to generated method
    ```
    getQuantity() becomes getQuantityGen()
    ```
  - During regen, the `...Gen()` method will be regenerated
  - Create your own `getQuantity()` method and call `getQuantityGen()`
Regeneration and Merge

- Other merge behavior is as expected
  - Extend a generated interface
  - Add interfaces to a generated class

- Support for regeneration from updated Rose model, XML Schema or @model annotations
Agenda

- EMF in a nutshell
- EMF Components
- The Ecore Metamodel
- Model Definition
- Code Generation, Regeneration and Merge
- **EMF Runtime Framework**
- Change Model
- Validation Framework
- EMF.Edit Framework
- Service Data Objects (SDO)
- Summary and Q&A
EMF Runtime Framework

- 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 reflection API
- Runtime environments
  - Eclipse
  - Standalone Java
Persistence and Serialization

- Serialized data is referred to as a **Resource**
  - Data can be spread out among a number of resources becoming a **Resource Set**
- When a model is loaded only the required resources of the resource set are loaded
  - Proxies exist for other resources in the set
  - Lazy or demand-loading of other resources as needed
  - A resource can be unloaded
Persistence and Serialization

- Save model objects using EMF Resources
  - Generic XML Resource implementation
  - Other Resource implementations possible

```java
poResource = ...createResource(..."p1.xml"...);
poResource.getContents().add(p1);
poResource.save(...);
```

```xml
p1.xml:

<PurchaseOrder>
  <shipTo>John Doe</shipTo>
  <next>p2.xml#p2</next>
</PurchaseOrder>
```
Proxy Resolution and Demand Load

```
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 = ... 
aPurchaseOrder.eAdapters().add(poObserver);

Adapter

adapter.notifyChanged()

setBillTo()

PurchaseOrder
Model Change Notification

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

```java
PurchaseOrder aPurchaseOrder = ...
AdapterFactory somePOAdapterFactory = ...
Object poExtensionType = ...
if
    (somePOAdapterFactory.isFactoryForType(poExtensionType)) {
        Adapter poAdapter =
            somePOAdapterFactory.adapt(aPurchaseOrder,
                                       poExtensionType);
    ...
}
```
Model Change Notification

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

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

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

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

\[
p1 \texttt{.setNext(p3);}
\]
Reflection

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

```java
public interface EObject {
    EClass eClass();
    Object eGet(EStructuralFeature f);
    void eSet(EStructuralFeature f, Object v);
    ...
}
```
Reflection

- Efficient generated switch implementation of reflective methods

```java
public Object eGet(EStructuralFeature eFeature) {
    switch (eDerivedStructuralFeatureID(eFeature)) {
        case POPackage.PURCHASE_ORDER__SHIP_TO:
            return getShipTo();
        case POPackage.PURCHASE_ORDER__BILL_TO:
            return getBillTo();
        ...
    }
}
```
Reflection Example

- Adding a book in a Library model using generated API

```java
Book book = LibraryFactory.eINSTANCE.createBook();
book.setTitle("King Lear");
book.setLibrary(branch);
```

- Adding a book to the model using reflection

```java
Book bookR = (Book)LibraryFactory.eINSTANCE.create(
    LibraryPackage.eINSTANCE.getBook());
bookR.eSet(LibraryPackage.eINSTANCE.getBook_Title(),
    "King Lear");
bookR.eSet(LibraryPackage.eINSTANCE.getBook_Library(),
    branch);
```
Reflection and Dynamic EMF

- Given an Ecore model, EMF also supports dynamic manipulation of instances
  - 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
Reflection and Dynamic EMF

- Reflection allows generic access to any EMF model
  - Similar to Java’s introspection capability
  - Every EObject (which 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 Demo
Agenda

- EMF in a nutshell
- EMF Components
- The Ecore Metamodel
- Model Definition
- Code Generation, Regeneration and Merge
- EMF Runtime Framework
- Change Model
- Validation Framework
- EMF.Edit Framework
- Service Data Objects (SDO)
- Summary and Q&A
Change Model

3 components

- The Change Recorder
- The Change Model
- Apply changes capability
The Change Recorder

- Adapter that populates the Change Model with the description of changes made to instances of any “EMF” model
- Transaction like notation

```java
... resource.getContents().add(library);
ChangeRecorder changeRecorder = new ChangeRecorder(resource);
library.getWriters().add(writer1);
writer1.setName("Frank");
library.getBooks().add(book1);
book1.setAuthor(writer1);
ChangeDescription changeDescription = changeRecorder.endRecording();
```
The Change Model
Apply Changes Capability

- The model stores the change information in reverse order
- The apply and applyAndReverse methods on ChangeDescription are able to undo the changes
Agenda

- EMF in a nutshell
- EMF Components
- The Ecore Metamodel
- Model Definition
- Code Generation, Regeneration and Merge
- EMF Runtime Framework
- Change Model
- *Validation Framework*
- EMF.Edit Framework
- Service Data Objects (SDO)
- Summary and Q&A
Validation Framework

Model objects validated by external EValidator:

```java
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);
    ...
}
```
Validation Framework

- 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
EValidator Implementations

**Framework:**

- 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
EValidator Implementations

Framework (cont):

- EObjectValidator validates basic EObject constraints
  - Multiplicities are respected
  - Proxies resolve
  - All referenced objects are contained in a resource
  - Data type values are valid
EValidator Implementations

- **Generated:**
  - Dispatch validation to type-specific methods
  - For model objects, one method is called for each…
    - Invariant: defined directly on the class, as an operation with \texttt{<<inv>>} stereotype
    - Constraint: externally defined for the class via the validator method
  - In either case, method body must be hand-coded
EValidator Implementations

- Generated:
  - Constraints generated, with implementation, for simple type facets defined in XML Schema
  - Basic constraints inherited from EObjectValidator
Agenda

- EMF in a nutshell
- EMF Components
- The Ecore Metamodel
- Model Definition
- Code Generation, Regeneration and Merge
- EMF Runtime Framework
- Change Model
- Validation Framework
- EMF.Edit Framework
- Service Data Objects (SDO)
- Summary and Q&A
The EMF.Edit Framework

- The EMF.Edit framework extends the base EMF model framework with generic reusable classes to support:
  - Model viewers
  - Command-based model modifications
  - Building editors for EMF models

- It provides:
  - Adapters that link an EMF Model to the requirements of Eclipse editors, views and property sheets
  - A command framework, including a set of generic command for building editors
  - A generator that produces a fully functional structured editor
EMF.Edit Model Viewers and Editors

- Can generate complete Eclipse SWT/JFace based editors
- Partial view support for other UI libraries (e.g., Swing)
- EMF command framework provides full undo/redo support
The Eclipse UI framework includes viewer classes for displaying structure models.

Hierarchy provides common framework for all viewers
JFace Viewers

- JFace viewers work with any kind of object
- Viewers access the model objects through an adapter object called a content provider
- Viewers are adapters that add model-based content to certain kinds of SWT widgets (list, tree, table)
- Content viewers interact with a content provider and label provider to obtain viewer structure and visual content
Content and Label Providers

- Mapping between an Eclipse view and a model
  1. **ContentProvider**: Maps the input object to the set of objects and associated structure
  2. **LabelProvider**: Provides a text label and icon for each displayed object
EMF.Edit ItemProviders

- EMF.Edit introduces Item Providers which mesh nicely with JFace content and label providers
- ItemProviders are the key to EMF edit with the following functions:
  1. Implement content and label provider functions
  2. Provide property descriptors
  3. Forward EMF model change notifications to viewers
- Provide functions on behalf of an editable model object (item)
- Reflective Item Provider
EMF.Edit ItemProviders

- EMF.Edit’s AdapterFactoryContentProvider implements all generic JFace content provider interfaces
- EMF.Edit’s AdapterFactoryLabelProvider works similarly for label providers
- Both Delegate to corresponding ItemProvider interfaces
  - for example, ITreeItemProviders know how to navigate the model objects (items) for a TreeViewer

```java
public interface ITreeItemProvider {
    public Collection getChildren(Object object);
    public Object getParent(Object object);
    ..........}
```

![Diagram of ItemProviders](image)
Editing Commands

So far, we’ve looked at how to view an EMF model, what about editing?

Editing means undoable modification

Editing command support:

- Interact with EMF objects
- Automatic undo and redo
- Commands work on any EMF model
- Handle simple or complex operations
EMF.Edit Commands

The Basic EMF.Edit Commands are:

1. SetCommand
2. AddCommand
3. RemoveCommand
4. MoveCommand
5. ReplaceCommand
6. CopyCommand
7. DragAndDrop uses CopyCommand, RemoveCommand, and AddCommand
Command-based Editor Actions

- EMF.Edit also provides editor actions for the generic commands
How to refresh a viewer after a command changes something in a model?

- AdapterFactoryContentProvider (INotifyChangedListener) 
  - TreeViewer
  - viewer.update()
  - listener.fireNotifyChanged()

- ItemProviderAdapterFactory (IChangeNotifier) 
  - LibraryItemProvider 
    - Library
    - notifier.fireNotifyChanged()
    - adapter.notifyChanged()
EMF.Edit EditingDomain

- EMF.Edit framework uses an Editing Domain to manage an editor’s command-based modification of a model.

- The Editing Domain provides a context for executing commands.

- An Editing Domain interface is used to provide editing access to an EMF model.

- The Editing Domain provide three main functions:
  1. Creating commands
  2. Managing the command undo stack
  3. Providing convenient access to the set of EMF resources being edited.
Command Creation

- EditingDomains delegate command creation to ItemProviders
  - Makes it easy to override any command for specific types of objects

- To customize a command:
  - Override the required `createCommand()` method from the framework item provider base class, `ItemProviderAdapter`

```java
protected Command createSetCommand(
    EditingDomain domain,
    EObject owner,
    EStructuralFeature feature,
    Object value)
{
    if (feature == LibraryPackage.eINSTANCE.getMedia_Status())
        return new setMediaStatusCommand(domain, owner, feature, value);
    return super.createSetCommand(domain, owner, feature, value);
}
```
Agenda

- EMF in a nutshell
- EMF Components
- The Ecore Metamodel
- Model Definition
- Code Generation, Regeneration and Merge
- EMF Runtime Framework
- Change Model
- Validation Framework
- EMF.Edit Framework
- Service Data Objects (SDO)
- Summary and Q&A
## SDO Goals and Requirements

- SDO is designed to simplify and unify the way in which applications handle data.
- Some of the technologies:

<table>
<thead>
<tr>
<th>Data Source</th>
<th>MetaData API</th>
<th>Query Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connected</td>
<td>Dynamic</td>
<td>SQL</td>
</tr>
<tr>
<td>Relational</td>
<td>Relational</td>
<td>N/A</td>
</tr>
<tr>
<td>Disconnected</td>
<td>Static</td>
<td>Java Introspection</td>
</tr>
<tr>
<td>XML</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JAX-RPC</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>JAXB</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>DOM and SAX</td>
<td>Dynamic</td>
<td>XML InfoSet</td>
</tr>
<tr>
<td>XML</td>
<td></td>
<td>XPath, XQuery</td>
</tr>
</tbody>
</table>

© Copyright IBM Canada Ltd., 2005. All rights reserved.
SDO Goals and Requirements

- SDO allows uniformly access and manipulate data from heterogeneous data sources:
  - Relational and XML databases
  - EJBs
  - XML
  - Web Services
- SDO metadata is just like EMOF (UML)
SDO Concepts

- Supports **disconnected** browsing and modifications of data
- SDO enables both a static (or **strongly typed**) programming model and a dynamic (or **loosely typed**) programming model.
  - `purchaseOrderDO.getShipTo();`
  - `purchaseOrderDO.get("shipTo");`
- **DataObject** – a business data object
  - Holds properties of data and its values
- **DataGraph**
  - Contains DataObjects and a ChangeSummary
SDO Architecture
SDO Java API
DataObject

- Interface that provides access to a business data of any type
  - Provides methods to retrieve and update data
  - Provides access to the metamodel
  - DataObject is composed of Properties
  - Data can be accessed dynamically using type and property
DataObject Methods

- Generic accessors
  - Retrieving value by name, Property, SDO-Path…

- Typed accessors
  - `getXXX()` methods to retrieve typed value, e.g. `getBoolean("isManager")`
  - Automatic type conversion
Example: Purchase Order update

DataObject purchaseOrder = dms.load("purchaseOrder.xml");
purchaseOrder.setString("orderDate", "1999-10-20");
DataObject shipTo = purchaseOrder.createDataObject("shipTo");
shipTo.set("country", "US");
shipTo.set("name", "Alice Smith");
...
DataObject billTo = purchaseOrder.createDataObject("billTo");
billTo.set("country", "US");
billTo.set("name", "Robert Smith");
...
DataObject item1 = items.createDataObject("item");
item1.set("partNum", "872-AA");
item1.set("productName", "Lawnmower");
item1.setInt("quantity", 1);
Sequence

- Used when dealing with semi-structured content, e.g. XML “mixed”
  - List of Property-value pairs
  - \{“numbers”, 1\}, \{TEXT, “mixed text”\}
Sequence API

```java
public interface Sequence
{
    int size();

    Property getProperty(int index);
    Object getValue(int index);

    Object setValue(int index, Object value);

    boolean add(String propertyName, Object value);
    boolean add(int propertyIndex, Object value);
    boolean add(Property property, Object value);
    void add(int index, String propertyName, Object value);
    void add(int index, int propertyIndex, Object value);
    void add(int index, Property property, Object value);

    void remove(int index);
    void move(int toIndex, int fromIndex);
}
```
DataGraph

- Consists of single root
- DataObjects must be contained in the graph

```java
public interface DataGraph {
    DataObject getRootObject();
    DataObject createRootObject(String namespaceURI, String typeName);
    DataObject createRootObject(Type type);
    ChangeSummary getChangeSummary();
    Type getType(String uri, String typeName);
}
```
ChangeSummary

- Provides access to history of changes

```java
public interface ChangeSummary {
    void beginLogging();
    void endLogging();
    boolean isLogging();

    DataGraph getDataGraph();

    List /*DataObject*/ getChangedDataObjects();
    boolean isCreated(DataObject dataObject);
    boolean isDeleted(DataObject dataObject);

    public interface Setting {
        Property getProperty();
        Object getValue();
        boolean isSet();
    }

    List /*ChangeSummary.Setting*/ getOldValues(DataObject dataObject);
}
```
public interface Type
{
    String getName();
    String getURI();

    Class getInstanceClass();
    boolean isInstance(Object object);

    List /*Property*/ getProperties();
    Property getProperty(String propertyName);
}

public interface Property
{
    String getName();
    Type getType();

    boolean isMany();
    boolean isContainment();

    Type getContainingType();

    Object getDefault();
}
SDO in the future…

- Currently only basic API subset is defined
- Hope to see:
  - DMS API
  - Factories: create DataObjects, retrieve Type and Property objects.
  - Copy, equality, XML helpers
  - XML Schema to SDO mapping
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 (cont)

- EMF provides the following:
  - A model (Ecore) with which your models can be built
    - Model created from Rose, XML Schema, or annotated Java interfaces
  - Generated Java code
    - Efficient and straightforward
    - Code customization preserved
  - Persistence and Serialization
    - Default is XMI (XML metadata interchange) but can be overridden
    - Serialized to resources and resource sets
  - Model change notification is built in
    - Just add observers (listeners) where needed
  - Reflection and Dynamic EMF
    - Full introspection capability
Resources

- EMF Project Web Site
  - http://www.eclipse.org/emf/
  - overviews, tutorials, newsgroup, mailing list, Bugzilla

- SDO:
  - http://dev2dev.bea.com/technologies/commonj/index.jsp

- Eclipse Modeling Framework by Frank Budinsky et al.
  - Addison-Wesley; 1st edition (August 13, 2003)

- IBM Redbook
  - publication number: SG24-6302-00