



# The Eclipse Project

An introduction to the Eclipse Project



# What is Eclipse?

- Eclipse is an open source project
  - <http://www.eclipse.org>
  - Consortium of companies, including IBM
  - Launched in November, 2001
  - Designed to help developers with specific development tasks



# Projects

- Needs to be taken from the Eclipse website  
[Todo: TM]



# Brief History of Eclipse

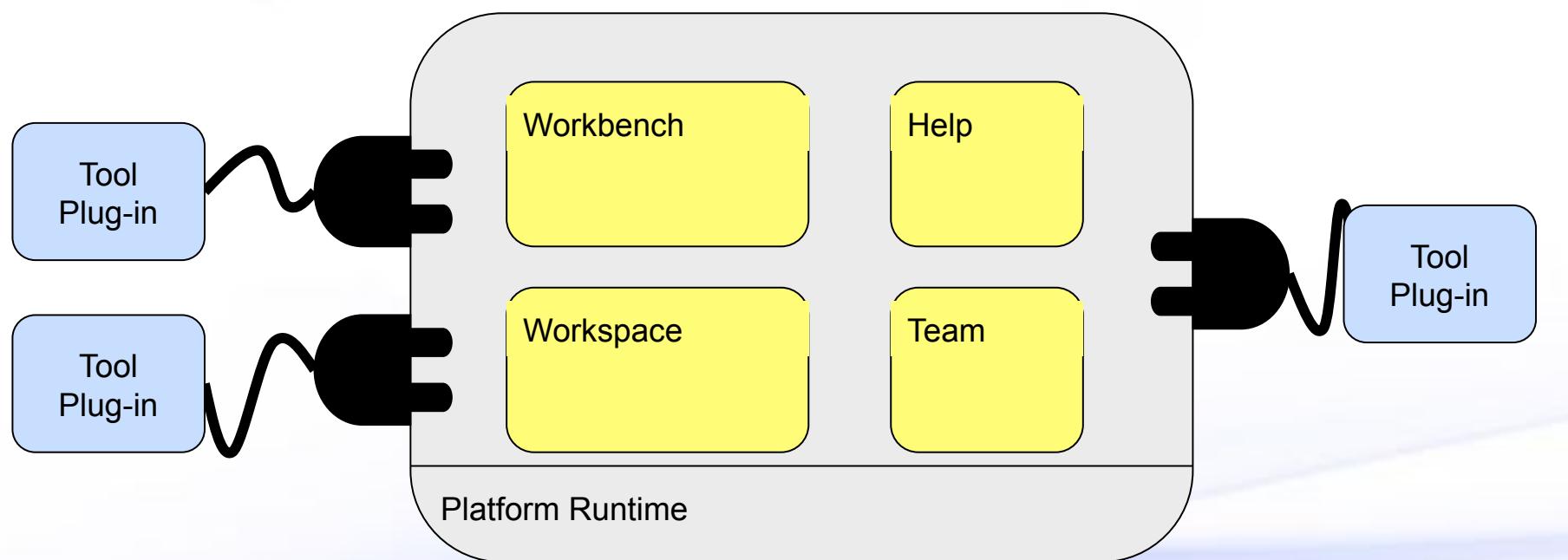
- 1994
  - VisualAge for Smalltalk
- 1996
  - VisualAge for Java
- 1996-2001
  - VisualAge Micro Edition
- 2001
  - Eclipse Project



# The Motivation Behind Eclipse

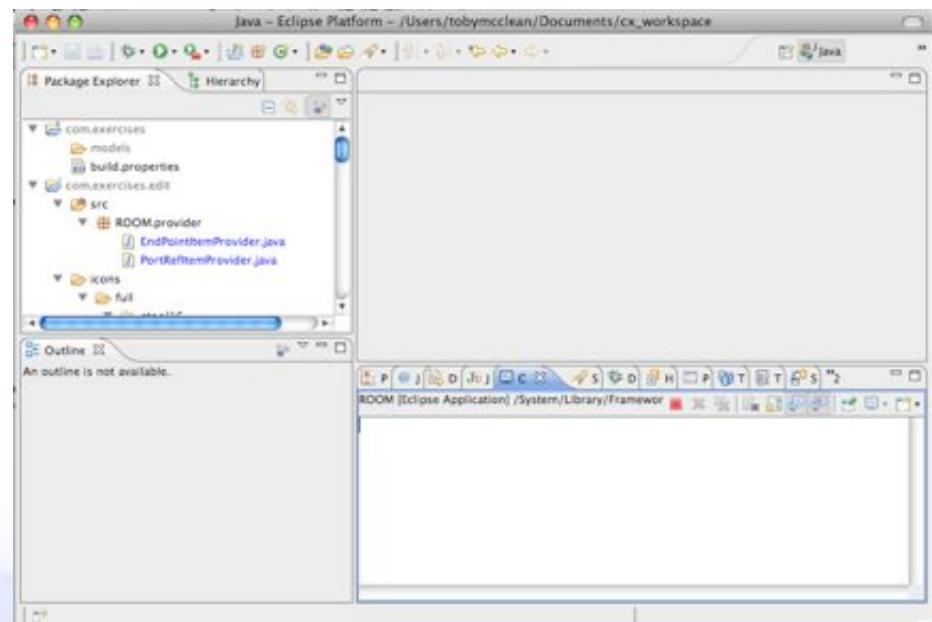
- Support for the construction of application development tools
- Support for the development of GUI and non-GUI application development
- Support for multiple content types
  - Java, HTML, C, XML
- Facility the integration of tools
- Cross-platform support

# Eclipse Architecture



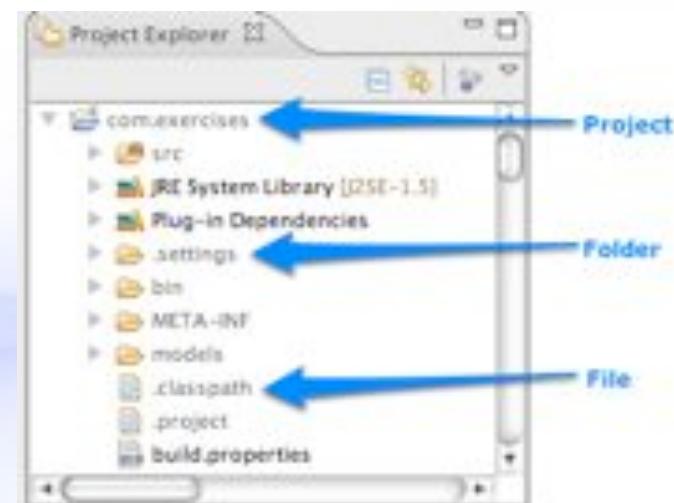
# Workbench

- The desktop development environment
  - Tools for resource management
  - Common of navigating through resources
- Multiple workbenches can be opened at the same time



# Workspace

- Represents the user data
- A set of resources
  - Projects
    - Collection of files and folders
  - Folders
    - Contain other folders or files
  - Files



# Help

- For the creation and publishing of documentation
- Supports both
  - User guides
  - Programmer guides
- Created using
  - HTML for content
  - XML for navigation
- Support for context sensitive help





# Team

- Provides support for
  - Versioning
  - Configuration management
  - Integration with team repository
- Allows team repository provider to hook into environment
  - Team repository providers specify how to intervene with resources
- Has optimistic and pessimistic locking support



# How is Eclipse Used?

- As an Integrated Development Environment
  - Supports the manipulation of multiple content types
  - Used for specifying and designing applications
    - Requirements, models, documentation, etc.
  - Used for writing code
    - Java, C, C++, C#, Python, ...
- As a product base
  - Supported through plug-in architecture and customizations



# Eclipse as an IDE

- Java Development Tooling
  - Editors, compiler integration, ant integration, debugger integration, etc.
- C/C++ Development Tooling
  - Editors, compiler integration, make integration, debugger integration, etc.
- Dynamic languages
  - Editors, interpreter and compiler integration, debugger integration, etc.
- Modeling projects



## Eclipse as a Product Base

- Eclipse can be used a Java product base
- Its flexible architecture can be used as product framework
  - Reuse plug-in architecture
  - Create new plug-ins
  - Customize the environment
- Support for branding
- Rich client platform support



## Rich Client Platform

- A minimal set of Eclipse plug-ins necessary for building a product
- Control over resource model
- Control over look and feel
- But still capable of leveraging existing plug-ins



# Summary

- In this module we explored
  - Eclipse, its background and the components that form its foundation
  - Eclipse use cases

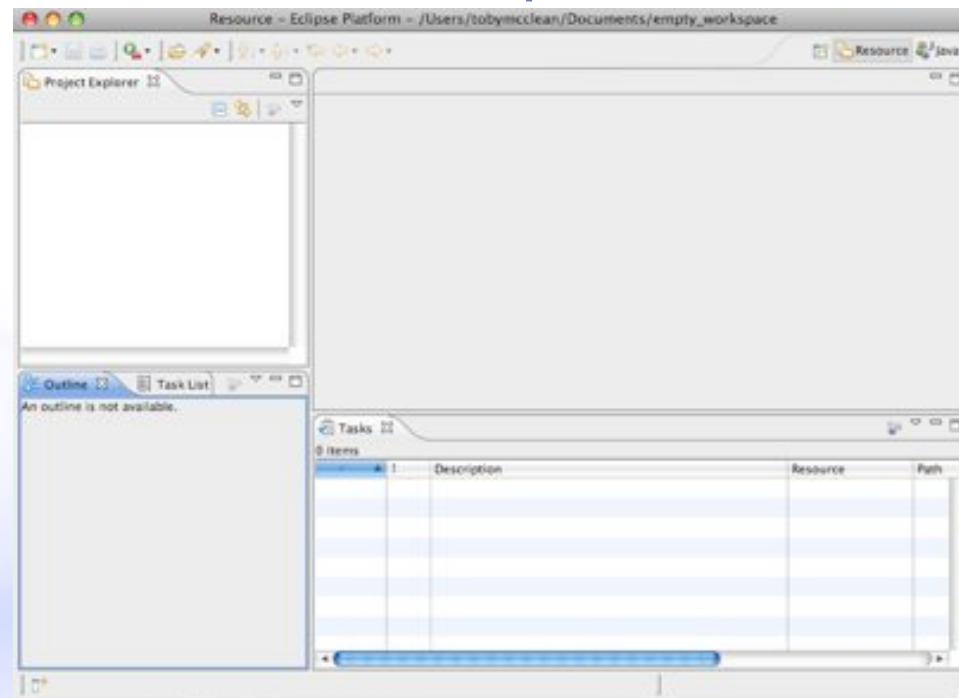


# The Eclipse Project

The Eclipse Workbench

# What is the Workbench?

- The working environment in Eclipse





## Multiple Workbench Instances

- Instance of Workbench comes up when Eclipse is launched
- It is possible to open another instance of the Workbench
  - Window → New Window
  - This opens up a new Workbench window
  - Can have different perspectives open in the different windows
  - Same result is not achieved by launching Eclipse twice



# Resources in a Workbench

- When working in Eclipse, you work with Resources
- Resources are organized in file/directory structure in the Workbench
  - They correspond to the actual files and directories in the Workspace
  - There are three different levels of resources:
    - Projects
    - Folders
    - Files
- Resources from the file system can be dragged into the workbench

# Importing Resources

- Available through
  - Menu option **File → Import...**
  - Right-click menu **Import...** option on a Folder or Project



# Exporting Resources

- Available through
  - Menu option **File → Export...**
  - Right-click menu **Export...** option on a Resource in the Workbench





# Refreshing Workbench

- Used for refreshing resources that change in the Workspace outside the workbench
- For example, if a file added to a directory in the Workspace
  - Select the project or directory in the workbench
  - Choose **Refresh** from its context menu
  - Alternatively you can press the **F5 button** on your keyboard
- Best practice
  - Work with the resources from within the Workbench when possible



# Resource History

- Changing and saving a resource results in a new version of the resource
  - All resource versions are stored in local history
  - Each resource version is identified by a time stamp
  - This allows you to compare different versions of the resource
- There are two actions to access the local history of a resource, from its context menu
  - **Compare With → Local History...**
  - **Replace With → Local History...**



# Workbench Components

- The Workbench contains Perspectives
- A Perspective has Views and Editors

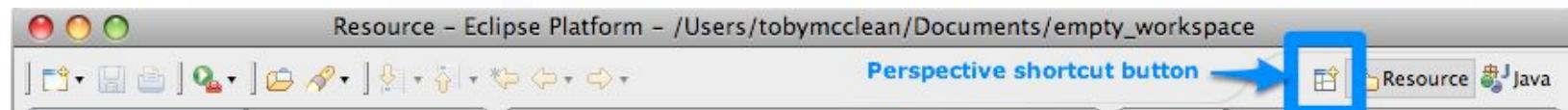


# Perspectives

- Perspective defines the initial layout of views in the Workbench
- Perspectives are task oriented, i.e. they contain specific views for doing certain tasks

# Choosing a Perspective

- To change the current perspective of the Workbench
  - **Window → Open Perspective →**
  - Clicking on the Perspective shortcut button





## Saving a Perspective

- Arrangement of views and editors can be modified and saved for perspectives
  - **Window → Save Perspective As...**
  - It can be saved under an existing name or a new name creating a user-defined perspective



## Resetting a Perspective

- Sometimes views and editors for a perspective need to be reset to the defaults
  - **Window → Reset Perspective...**
- This only applies to default Eclipse perspectives not user-defined ones



# Customizing Perspectives

- The shortcuts and commands for the current perspective can be customized
  - **Window → Customize Perspective...**
- This is in addition to the customization of the views that are available in the perspective

# Closing Perspectives

- The upper right corner of the workbench shows the open perspectives



- It is possible to close a perspective
  - Window → Close Perspective
  - Window → Close All Perspectives



## Editors

- An editor for a resource opens when you double-click on a resource
  - Editor type depends on the type of resource
  - An editor stays open when the perspective is changed
  - Active editor contains menus and toolbars specific to the editor
  - When a resource has been changed an asterisk in the editor's title bar indicates unsaved changes



# Editors and Resource Types

- It is possible to associate an editor with a resource type by the following actions
  - **Window → Preferences**
  - Select **General**
  - Select **Editors**
  - Select **File Associations**
  - Select the resource type
  - Click the **Add** button to associate it with a particular editor
- In same dialog the default editor can be set
- Others are available from **Open With** in the resources context menu



# Views

- The main purpose of a view is
  - Support editors
  - Provide alternative presentation and navigation
- Views can have their own menus and toolbars
  - Items available in menus and toolbars are only available in that view



## More Views

- Views can
  - Appear on their own
  - Appear stacked with other views
- Layout of views can be changed by clicking on the title bar and moving views
  - Single views can be moved together with other views
  - Stacked views can be moved to be single views



# Adding Views to the Perspective

- To add a view to the current perspective
  - **Window → Show View → Other...**
  - Select the desired view
  - Click the **Ok** button



# Stacked Views and Stacking Views

- Stacked views appear in a notebook form
  - Each view is a page in the notebook
- A view can be added to a stack by dragging into the area of the tab area of the 'notebook'
- Similarly a view can be removed from a stack by dragging its tab away from the 'notebook'



# Fast Views

- A fast view is hidden and can be quickly opened and closed
- Created by
  - Dragging an open view to the shortcut bar
  - Selecting Fast View from the view's menu
- A fast view is activated by clicking on its Fast View pop-up menu option
  - Bottom right of the workbench
- Deactivates as soon as focus is given to another view or editor



# Summary

- In this module we explored
  - Components of the Eclipse workbench
    - Perspectives;
    - Editors; and
    - Views



# The Eclipse Project

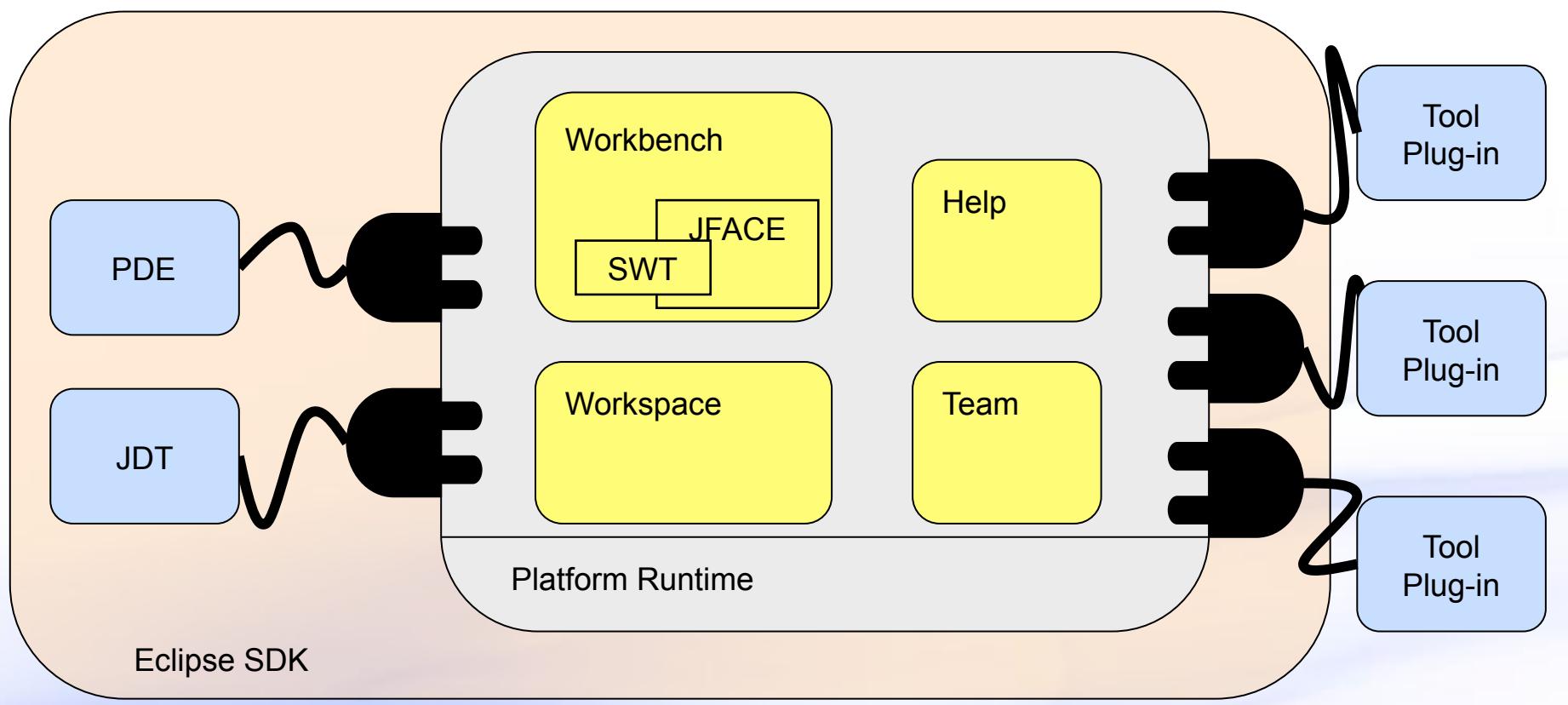
Eclipse Architecture



# Eclipse Architecture

- Flexible, structured around
  - Extension points
  - Plug-ins
- Architecture allows for
  - Other tools to be used within the platform
  - Other tools to be extended
  - Integration between tools and the platform

# More Eclipse Architecture





# Platform Runtime

- Everything is a plug-in except the Platform Runtime
  - A small kernel that represents the base of the Platform
- All other subsystems build up on the Platform Runtime following the rules of plug-ins
  - i.e. they are plug-ins themselves



## Extension Points

- Describe additional functionality that could be integrated with the platform
  - External tools extend the platform to bring specific functionality
- Two levels of extending Eclipse
  - Extending the core platform
  - Extending existing extensions
- Extension points may have corresponding API interface
  - Describes what should be provided in the extension

# Plug-ins

- External tools that provide additional functionality to the platform
- Define extension points
  - Each plug-in defines its own set of extension points
- Implement specialized functionality
  - Usually key functionality that does not already exist in the platform
- Provide their own set of APIs
  - Used for further extension of their functionalities



## More Plug-ins

- Plug-ins implement behavior defined through extension point API interface
- Plug-in can extend
  - Named extension points
  - Extension points of other plug-ins
- Plug-in can also declare an extension point and can provide an extension to it
- Plug-ins are developed in the Java programming language



# What Makes Up a Plug-in?

- Plug-ins consist of
  - Java code
    - Binaries
    - Source (optional)
  - plugin.xml
    - describes plug-in extensions and extension points
  - plugin.properties
    - For localization and configuration
  - manifest.mf
    - describes the plug-in

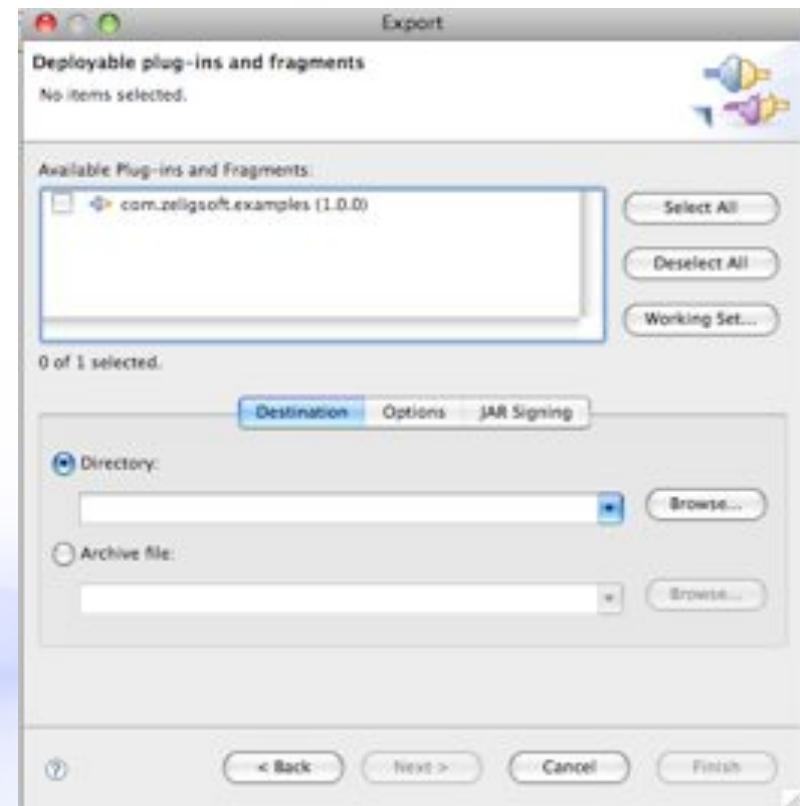


## Publishing Plug-ins

- Prepares plug-in for deployment on a specific platform
- Manual publishing
  - Using Export Wizard
  - Using Ant Scripts
- Automatic publishing is available by using PDE Build

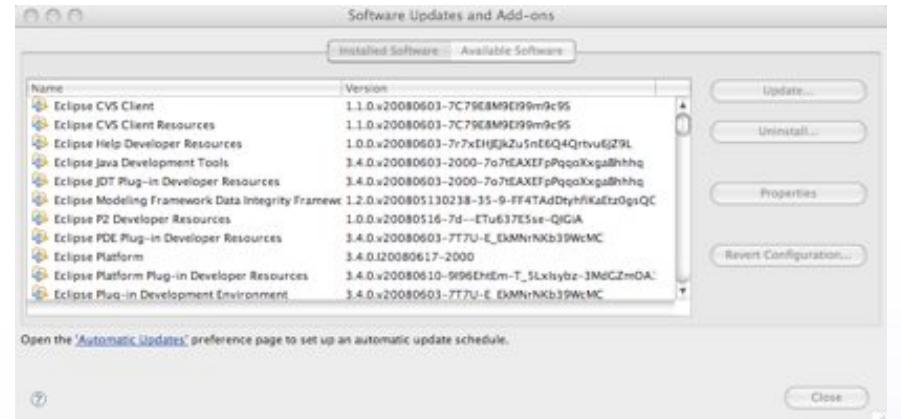
# Manually Publishing a Plug-in

- A plug-in can be published manually by
  - **File → Export...**
  - Select Deployable plug-ins and fragments
  - Click **Next >** button
- Parameters configure how the plug-in published
- Can be saved as an Ant script



# Installing Plug-ins

- Plug-ins are installed under \plugins\ directory of Eclipse installation
- Plug-ins can be installed from a update site
  - Help → Software Updates..
  - Typically as part of a feature
- Restart workbench

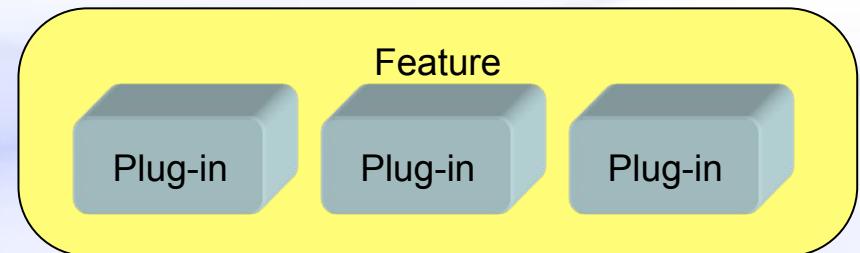


# Plug-in Fragments

- Used for extending existing plug-ins
  - Provide an additional functionality to existing plug-ins
  - Ideal for providing add-on functionality to plug-ins
- Packaged in separate files
  - Fragment content is treated as it was original plug-in archive
  - During runtime platform detects fragments and merges their content with original plug-ins
- Described in the fragment.xml files
  - Similar to plugin.xml manifest files
  - Plug-in archive can contain plug-ins or fragments

# Features

- An Eclipse feature is a collection of plug-ins
  - Represents smallest unit of separately downloadable and installable functionality
- Installed into \features\ directory of Eclipse installation
- Features can be installed from an update site





# Product

- Stand-alone application built on Eclipse platform
- Typically include the JRE and Eclipse platform
- Defined by using an extension point
- Configurable
  - Splash screen
  - Default preference values
  - Welcome pages
- Customized installation



# Summary

- In this module we explored
  - Eclipse architecture
  - Extension points
  - Plug-ins
  - Features
  - Products



# The Eclipse Project

Plug-ins



# Integration Between Plug-ins

- Supported through the ability to contribute actions to existing plug-ins
  - New plug-in contributes an action to existing plug-in
  - Allows for tight integration between plug-ins
- There are many areas where actions can be contributed
  - Context menus and editors
  - Local toolbar and pull-down menu of a view
  - Toolbar and menu of an editor - appears on the Workbench when editor opens
  - Main toolbar and menu for the Workbench



## Extending Views

- Changes made to a view are specific to that view
- Changes can be made to view's
  - Context menu
  - Menu and toolbar



# Extending Editors

- When extending an editor changes can be made to
  - Editor's context menu
  - Editor's menu and toolbar
- Actions defined are shared by all instances of the same editor type
  - Editors stay open across multiple perspectives, so their actions stay the same
  - When new workspace is open, or workbench is open in a new window, new instance of editor can be created



# Action Set

- Allows extension of the Workbench that is generic
  - Should be used for adding non-editor, or non-view specific actions to the Workbench
  - Actions defined are available for all views and editors
- Allows customization of the Workbench that includes defining
  - Menus
  - Menu items
  - Toolbar items



## Choosing What to Extend

- Eclipse has a set of predefined extension points
  - Called Platform extension points
    - Comprehensive set of extension points
    - Detailed in the on-line help
- It is possible to create new extension points
  - Requires id, name, and schema to be defined
  - Done through Plug-in Development Environment (PDE)



# Some Common Extension Points

- Popup Menus for editors and views
  - org.eclipse.ui.popupMenus
- Menu and toolbar for views
  - org.eclipse.ui.viewActions
- Menu and toolbars for editors
  - org.eclipse.ui.editorActions
- Menu and toolbar for the Workbench
  - org.eclipse.ui.actionSets
- Complete set of extension points located in Eclipse help
  - Search on “Platform Extension Points”



# How to Extend the Workbench?

- Steps for adding a plug-in
  - Write the plug-in code
    - Define Java project
    - Create Java class
    - Add appropriate protocols to the class
  - Package the class
  - Create plugin.xml and MANIFEST.MF
  - Test the plug-in
  - Deploy the plug-in



## Creating Java Project

- Project will contain source code for the plug-in
- Classes will be defined in the package



# Updating Project's Build Path

- To make classes required for the plug-in visible for the project, update its path
  - jFace.jar, swt.jar runtime, and workbench.jar files must be added to the project's build path



# Creating a Class

- For menu actions, define a class that
  - Subclasses client delegate class
  - Implements interface that contributes action to the Workbench



# Interface IWorkbenchWindowActionDelegate

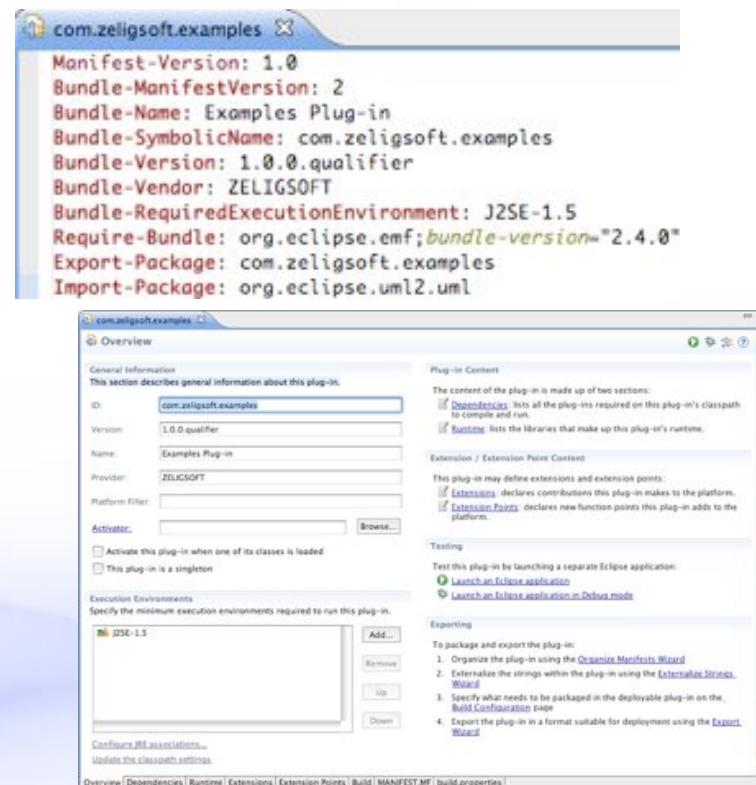
- Extend **IActionDelegate** and define methods
  - `init(IWorkbenchWindow)` - Initialization method that connects action delegate to the Workbench window
  - `dispose()` - Disposes action delegate, the implementer should unhook any references to itself so that garbage collection can occur
- Interface is for an action that is contributed into the workbench window menu or toolbar

# Class ActionDelegate

- Abstract base implementation for client delegate action (defines same methods as interface)
- In addition it also defines
  - runWithEvent(IAction, Event)
    - Does the actual work when action is triggered
    - Parameters represent the action proxy that handles the presentation portion of the action and the SWT event which triggered the action being run
    - Default implementation redirects to the run() method
  - run(IAction)
    - Inherited method, does the actual work as it's called when action is triggered
  - selectionChanged(IAction, ISelection)
    - Inherited method, notifies action delegate that the creation in the Workbench has changed

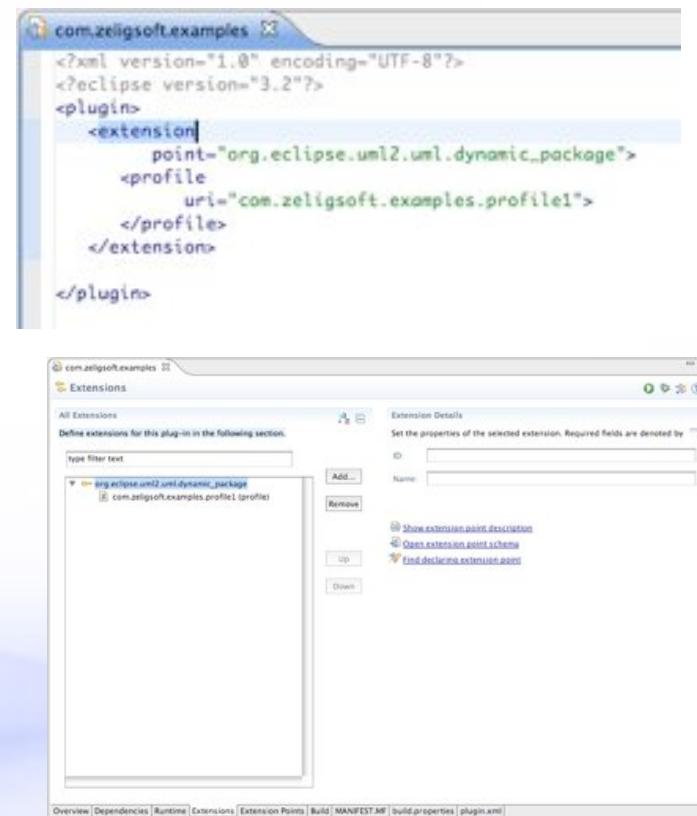
# Defining Manifest

- MANIFEST.MF file
  - In the META-INF directory
- Defines
  - Plug-in name
  - Plug-in version
  - Plug-in vendor/provider
  - Plug-in dependencies
- Manifest editor available



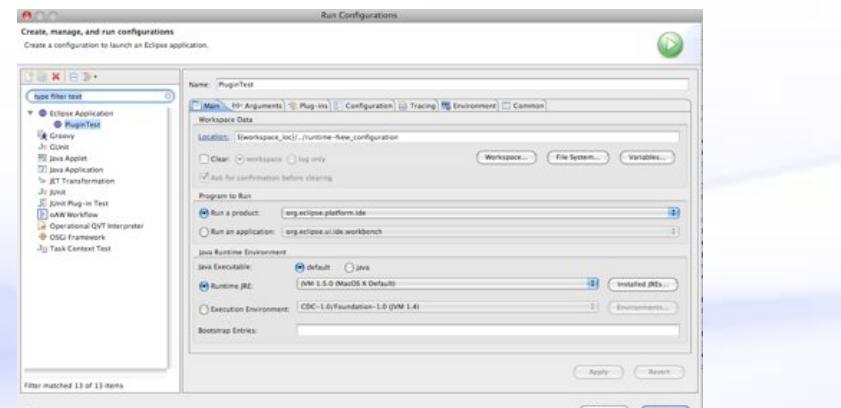
# Defining plugin.xml

- In the root of the project
- Defines
  - Extensions
  - Extension points
- Extension editor
  - Includes wizards
- Extension point editor



# Testing Plug-in from Workbench

- Available by opening new Workbench instance
  - Run → Run As → Eclipse Application
- Used when developing plug-in within the platform
- Can be configured
  - Run → Run Configurations...
  - Eclipse Application
- Debug available
  - Run → Debug As → Eclipse Application



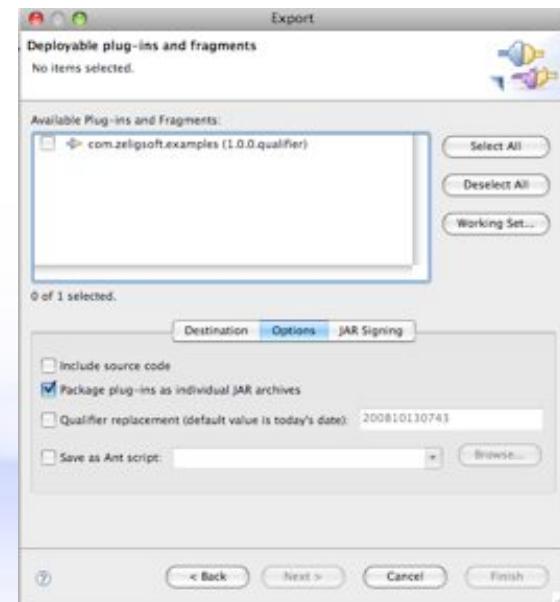


## Getting Ready to Export Plug-in

- Create a build.properties file in your project
  - source.. = src/
  - output.. = bin/
  - bin.includes = META-INF, \
  - .,\
  - plugin.xml
- Build properties editor available
  - Open with → Build Properties Editor

# Exporting a Plug-in

- **File → Export...**
  - Deployable Plug-ins and Fragments
- **Configure export**
  - Archive vs. Directory
  - Packaging
  - Include source?
- **Can save as ant script**
- **Ready to install**





## Plug-in Registry

- To see what plug-ins are registered within a Workbench instance
  - **Window → Show View → Other...**





# Eclipse and Rational Modeling Tools



# Eclipse Modeling Project

- Model-based development technologies
  - Frameworks
  - Tooling
  - Implementations of standards





Model  
Development Tools  
(MDT)

Eclipse Modeling  
Framework Technology  
(EMFT)

Model to Text  
(M2T)

Eclipse Modeling  
Framework  
(EMF)

Model to Model  
(M2M)

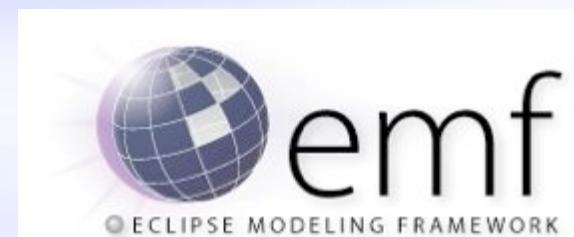
Textual Modeling  
Framework  
(TMF)

Graphical Modeling  
Framework  
(GMF)



# Eclipse Modeling Framework (EMF)

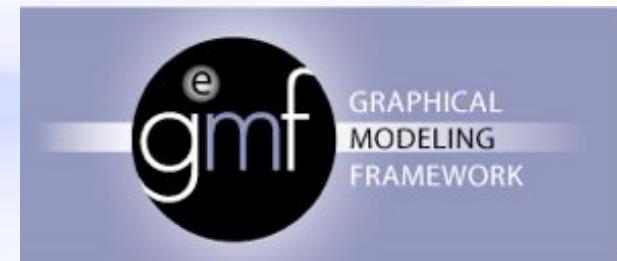
- Framework and generation for metamodels
- Sub-projects
  - CDO - distributed shared EMF models and server based O/R mapping
  - Model Query - specification and execution of queries against an EMF model
  - Model Transaction - model management layer
  - Net4j - extensible client-server system using Eclipse Runtime and Spring Framework
  - SDO - EMF implementation of service data objects, data application development in a SOA architecture
  - Teneo - database persistency for EMF models
  - Validation Framework - model integrity





# Graphical Modeling Framework (GMF)

- Runtime and tooling for developing a graphical concrete syntax
- Built on-top of EMF and GEF
- Feature rich graphical model editors





# Textual Modeling Framework (TMF)

- Runtime and tooling for developing textual concrete syntax
- Generation of editors and other tooling from a grammar
- Sub-projects
  - TCS
  - xtext

## TMF - TCS

- TCS - Textual Concrete Syntax
- Built on top of EMF
- Proprietary language for annotating an abstract syntax model with textual syntax
- Able to generate
  - Grammar
  - Editor
  - Model to text transformation with traceability



## TMF - xttext

- Developed as part of the openArchitectureWare
- Built on-top of EMF and Antlr
- From proprietary grammar language
  - Generate editor - syntax highlighting and code completion
  - Generate EMF abstract syntax model
  - Generate integration with generation framework
  - Generate integration with semantic validation framework





# Model Development Tools (MDT)

- Project that focuses on industry standard models
  - e.g. UML
- Sub-projects include
  - UML2
  - UML2 Tools
  - Object Constraint Language



# Eclipse Modeling Framework Technology (EMFT)

- A proving ground for EMF projects
  - Research projects
  - Incubation projects
- When a project matures it is moved or promoted
  - To another modeling sub-project
  - To its own sub-project
- Examples
  - EMF runtime for the .NET platform
  - Tools for building Ecore models



## Model to Model (M2M)

- Project that focuses on technologies for model to model transformation
- Work with EMF based as well as other input models
- Sub-projects include
  - QVT
  - ATL



## M2M - QVT

- Implementation of OMG's Query/View/Transformation specification
- Three languages for performing M2M transformation
  - Operational - currently supported
  - Relational - in development
  - Core - planned
- QVT editor, debugger and interpreter



## M2M - ATL

- ATL - ATLAS Transformation Language
- Developed by ATLAS group at INRIA & LINA
- Proprietary M2M language
  - Similar to the QVT operational language
- ATL editor, debugger and virtual machine



# Model to Text (M2T)

- Project that focuses on technologies for model to text transformation
- Work with EMF based as well as other input models
- Sub-projects
  - JET
  - xPand



## M2T- JET

- JET - Java Emitter Templates
- Template based language with an XML like syntax
- Editor for developing templates/transformations
- Execution engine for executing transformation
- Used by EMF to generate code



## M2T - xPand

- Developed as part of the openArchitectureWare
  - Maturing into an Eclipse project
- Template based language with a proprietary syntax
- Editor for developing templates/transformations
- Execution engine for interpreting templates
- Used by GMF to generate code



# Summary





# Exploring EMF

What is the Eclipse Modeling Framework?



# Overview

- Learn about the Eclipse Modeling Framework the basis for modeling related technologies in Eclipse



# Agenda



# Eclipse Modeling Framework

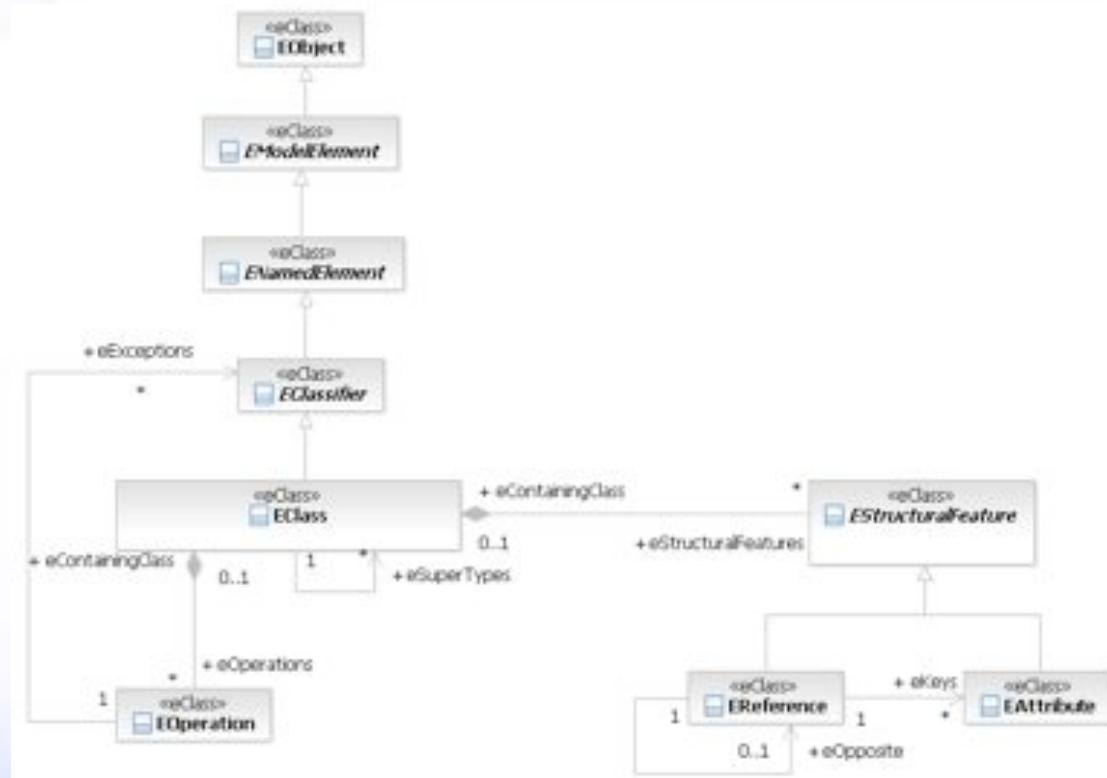
- Originally based on the OMGs' MOF
  - Supported a subset of the MOF
- Is now an implementation of EMOF
  - Essential MOF is part of MOF 2
- Used as a framework for
  - Modeling
  - Data integration
- Used in commercial products for 5+ years



# What is an EMF Model?

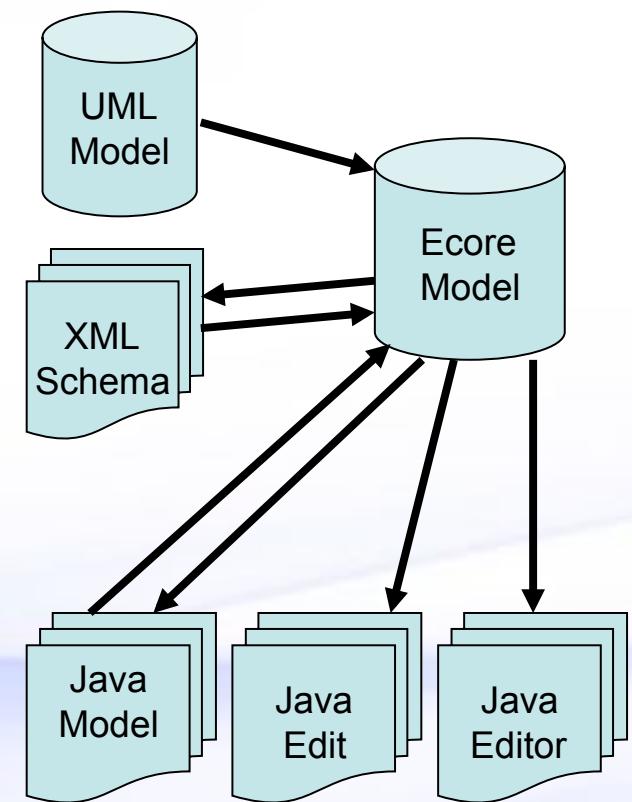
- Description of data for a domain/application
  - The attributes and capabilities of domain concepts
  - Relationships between the domain concepts
  - Cardinalities of attributes and relationships
- It defines the *metamodel* or abstract syntax for your domain
- Defined in the Ecore modeling language

# The Ecore Metamodel



# Creating an EMF Model

- EMF models can be created from
  - Java Interfaces
  - UML Class diagram
  - XML Schema
  - Ecore Diagram Editor
- With EMF if you have one of the above then you can generate the others



# EMF from Java

- EMF models can be created from annotated Java
  - Typically for EMFizing legacy software
- @model annotation indicates parts of the code that correspond to model elements
  - interface - creates a class in EMF
  - method - creates operation in EMF
  - get method - creates attribute in EMF
- Supports reloading
  - i.e. can update the model by editing Java

```
/** * * @model */  
public interface Capsule extends NamedElement {  
    /** * @model containment="true" */  
    List<Port> getPorts();  
  
    /** * @model */  
    Port getProtocol();  
  
    /** * @model */  
    boolean isConjugated();  
}
```

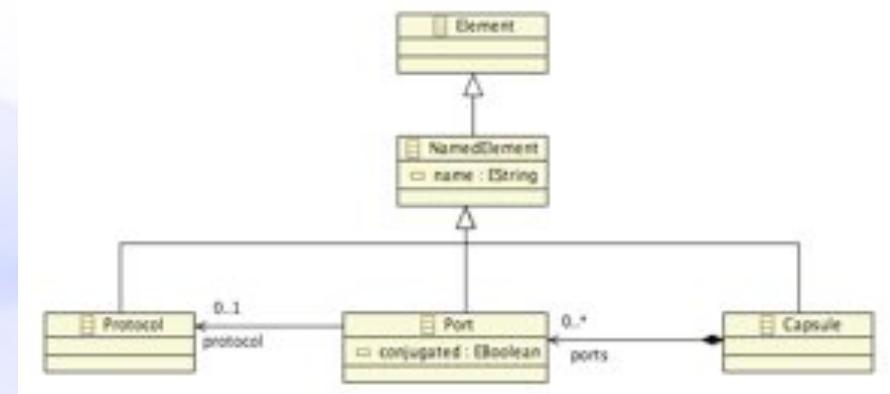


## More EMF from Java

- Create Java interfaces for metamodel
- Create EMF Model
  - Right-click on project/folder New --> Other...
  - Use Annotated Java Model Importer
  - Choose the Java package containing the annotated Java
  - Creates ecore and genmodel resources for the metamodel
- To update
  - Open the genmodel resource using the Ecore Generator editor
  - Select Generator --> Reload... from the main menu

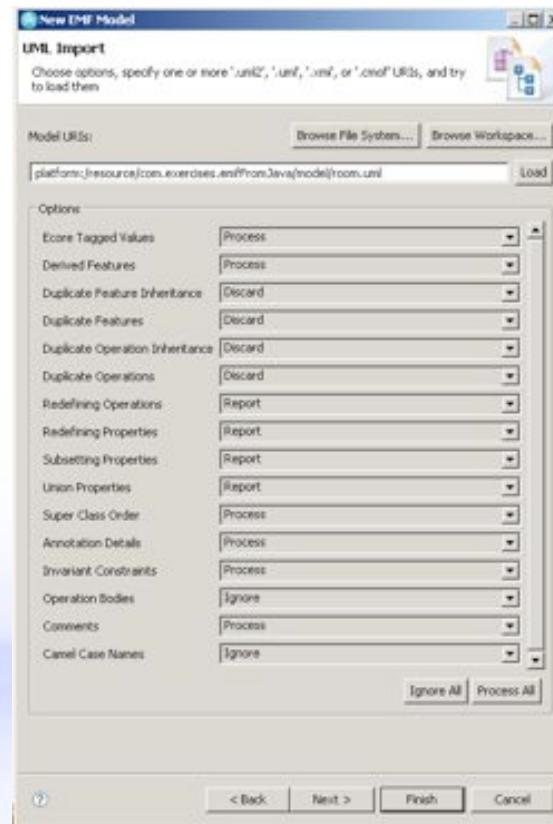
# EMF from UML Class Model

- EMF models can be created from a UML class model
- The classes are assumed to be metaclasses
- Supports reloading from the UML model
- Some restrictions



# More EMF from UML Class Model

- Create UML model
- Create EMF model
  - New --> Other...
  - Use UML Model Importer
  - Choose the UML model
  - Configure import
  - Creates ecore and genmodel resources for the metamodel
- To update
  - Open the genmodel resource using the Ecore Generator Editor
  - Select Generator --> Reload...





## EMF from XML Schema

- Many industry standards produce XML schemas to define their data format
- EMF model can be created from an XML schema
  - Model instances are schema compliant
- Supports reloading from XML schema
- Schema can be regenerated from EMF model



## More EMF from XML Schema

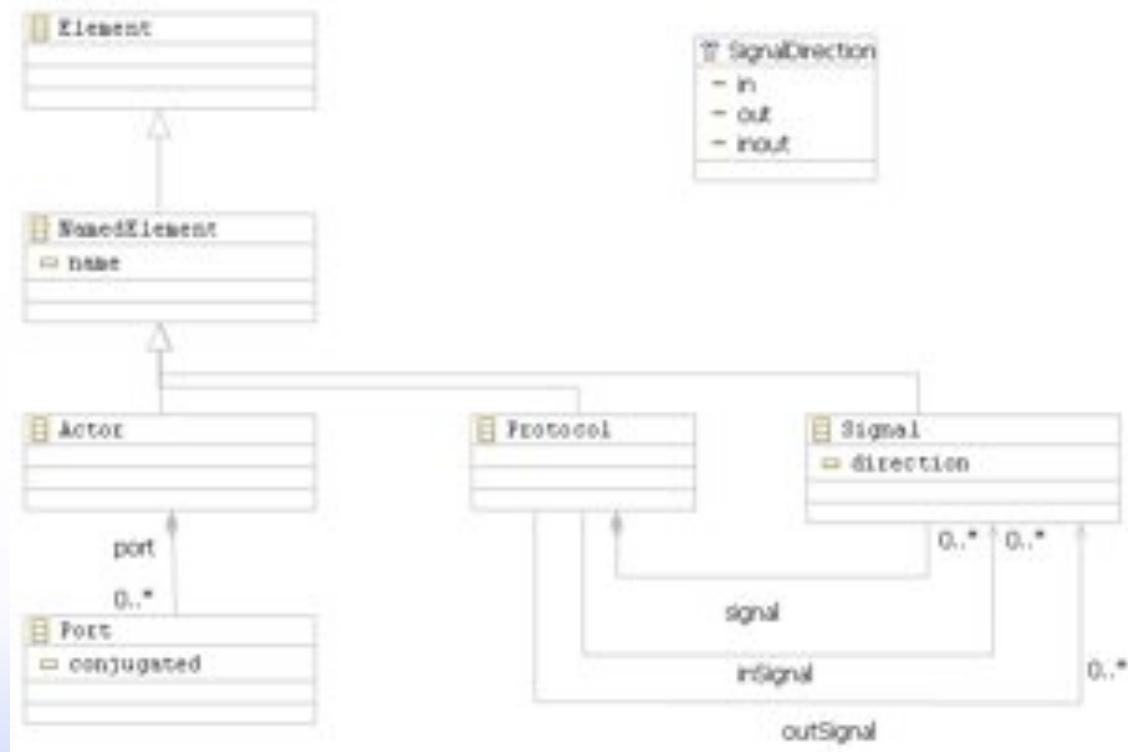
- Create/Find XML Schema
- Create EMF model
  - New --> Other...
  - Use XML Schema Importer
  - Choose the XML Schema
  - Creates ecore and genmodel resources for the metamodel
- To update
  - Open the genmodel resource using the Ecore Generator Editor
  - Select Generator --> Reload...



# EMF with Ecore Diagram Editor

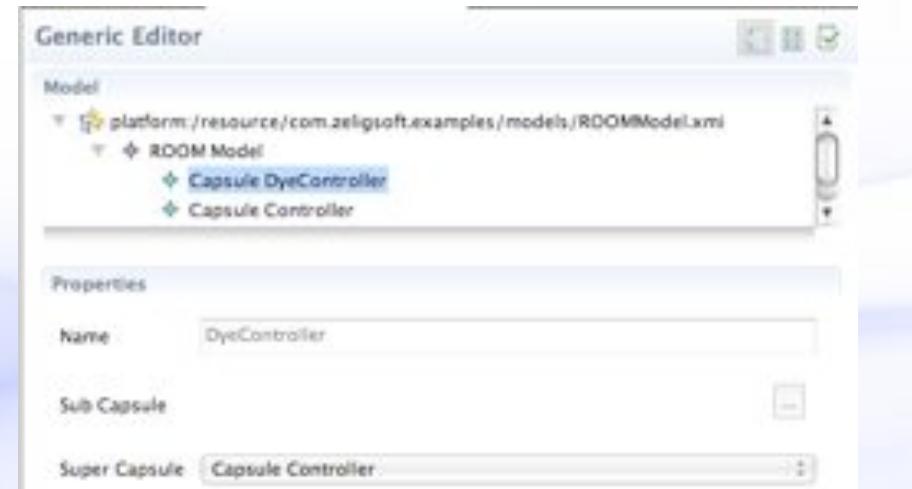
- The Ecore Diagram Editor provides a Class diagram like editor for graphically modeling
- Built on top of the GMF framework
- Editor is canonical
  - One diagram per EMF model
- Creates model and diagram resources
  - File --> New --> Other...
  - Ecore Diagram

# More EMF with Ecore Diagram Editor



# Dynamically Creating Models

- While developing a model it can be tested by creating dynamic instance
  - Does not require anything to be generated
  - Does not require a runtime workbench
  - Uses reflective capabilities of EMF
- In the Ecore editor
  - Right-click on the EClass
  - Create Dynamic Instance...
- Stored as XMI



# Registering the Model

- EMF maintains a registry of models
  - Access model through its namespace URI
- To register model
  - EPackage.Registry - programmatically
  - org.eclipse.emf.ecore.generated\_package extension point
  - org.eclipse.emf.ecore.dynamic\_package
- To access model
  - EPackage.Registry.INSTANCE.getEPackage(ns)

```
<extension
    point="org.eclipse.emf.ecore.dynamic_package">
<resource
    location="models/room.ecore"
    uri="http://www.zeligsoft.com/exercise/room/2008">
</resource>
</extension>
```



# Using Reflective API

- Model instances of an Ecore model can be created using the reflective API
- Reflective API
  - Generic API for working with EMF
  - Accessing and manipulating metadata
  - Instantiating classes
- Usage examples
  - EMF tool builders
  - Serialization and deserialization



## More Reflective API

```
//get the EPackage for the model we are working with using the  
//namespace URI that it is registered against  
EPackage ePackage = EPackage.Registry.INSTANCE.getEPackage("java.xmi");  
//using the package get an EClass that we want to instantiate  
EClass eClass = (EClass)ePackage.getEClassifier("JavaClass");  
//get a EStructuralFeature that we want to set on an instance of the EClass  
EStructuralFeature nameFeature = eClass.getEStructuralFeature("name");  
//instantiate the EClass  
EObject javaClass = ePackage.getEFactoryInstance().create(eClass);  
//set the name of the instantiated EClass  
javaClass.eSet(nameFeature, "DyeController");
```



# Summary

- In this module we explored
  - What EMF is...
  - How to create Ecore models
    - Java
    - XML Schema
    - UML model
    - From scratch with Ecore Diagram Editor
  - How to test an Ecore model under development



# Generating Code with EMF

- Code can be generated from the Ecore model to work with it
  - Model specific API rather than the reflective API
  - Support for editing model instances in an editor
- Driven by a generator model
  - Annotates the Ecore model to control generation
- Generated code can be augmented
  - Derived attributes
  - Transient or volatile attributes
  - Operations

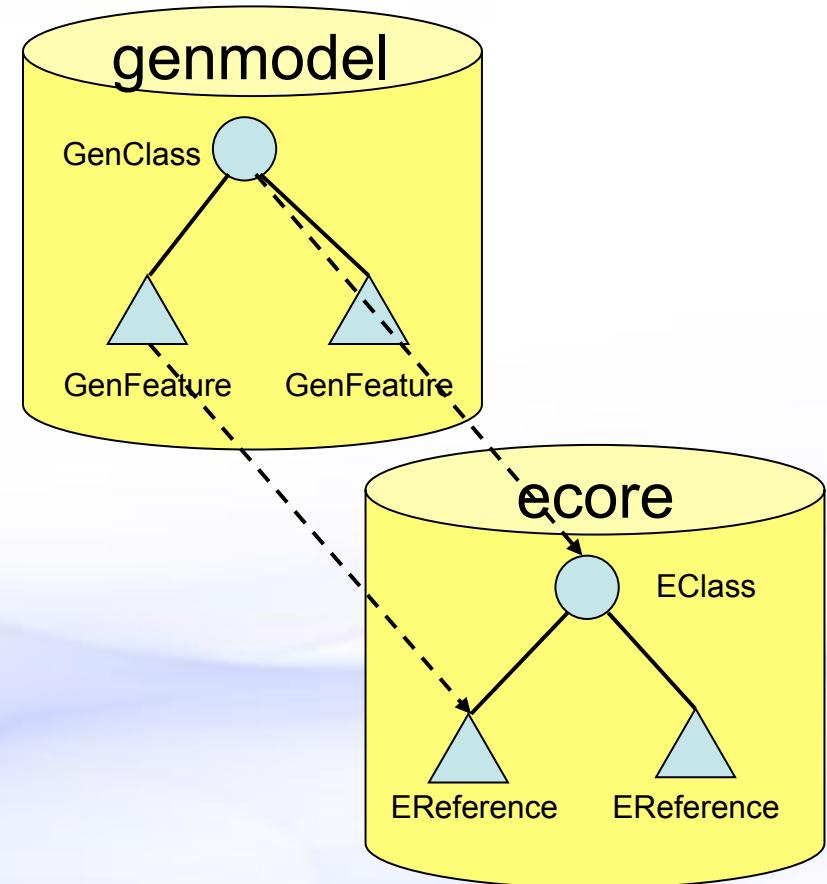


## More Generating Code with EMF

- Full support for regeneration and merge
- Code can be customized
  - Configuration parameters
  - Custom templates
- Code can be generated from
  - Workbench
  - Ant script
  - Command line

# The genmodel

- Wraps the Ecore model and automatically kept in sync
- Decorates the Ecore model
  - Target location for generated code
  - Prefix for some of the generated classes



# More genmodel

- Global generator configuration
- To generate
  - Context menu in the editor
  - Main menu in the editor
- Generates for
  - GenModel, GenPackage, GenClass and GenEnum
- What to generate
  - Model
  - Edit
  - Editor
  - Tests

| Property                 | Value                  |
|--------------------------|------------------------|
| All                      |                        |
| Bundle Manifest          | false true             |
| Compliance Level         | 3.0                    |
| Copyright Fields         | false                  |
| Copyright Text           |                        |
| Language                 |                        |
| Model Name               | Room                   |
| Model Plug-in ID         | com.zeligsoft.examples |
| Non-NLS Markers          | false                  |
| Runtime Compatibility    | false                  |
| Runtime Jar              | false                  |
| Runtime Version          | 2.4                    |
| ► Edit                   |                        |
| ► Editor                 |                        |
| ► Model                  |                        |
| ► Model Class Defaults   |                        |
| ► Model Feature Defaults |                        |
| ► Templates & Merge      |                        |
| ► Tests                  |                        |

# Model Code

- Java code for working with and manipulating model instances
- Interfaces, classes and enumerations
- Metadata
  - Package
- API for creating instances of classes
  - Factory
- Persistence
  - Resource and XML processor
- Utilities
  - E.g. Switch for visiting model elements



## More Model Code

| <b>Unit</b> | <b>Description</b>    | <b>File name</b>         | <b>Subpkg.</b> | <b>Opt.</b> |
|-------------|-----------------------|--------------------------|----------------|-------------|
| Model       | Plug-in Class         |                          |                | Y           |
|             | OSGi Manifest         | META-INF<br>/MANIFEST.MF |                | Y           |
|             | Plug-in Manifest      | plugin.xml               |                | N           |
|             | Translation File      | plugin.properties        |                | N           |
|             | Build Properties File | build.properties         |                | N           |

Source: EMF Eclipse Modeling Framework, Second Edition, Addison Wesley Professional



# More Model Code

| Unit    | Description       | File name                        | Subpkg. | Opt. |
|---------|-------------------|----------------------------------|---------|------|
| Package | Package Interface | <Prefix>Package.java             |         | N    |
|         | Package Class     | <Prefix>PackageImpl.java         | impl    | N    |
|         | Factory Interface | <Prefix>Factory.java             |         | N    |
|         | Factory Class     | <Prefix>FactoryImpl.java         | impl    | N    |
|         | Switch            | <Prefix>Switch.java              | util    | Y    |
|         | Adapter Factory   | <Prefix>AdapterFactory.java      | util    | Y    |
|         | Validator         | <Prefix>Validator.java           | util    | Y    |
|         | XML Processor     | <Prefix>XMLProcessor.java        | util    | Y    |
|         | Resource Factory  | <Prefix>ResourceFactoryImpl.java | util    | Y    |
|         | Resource          | <Prefix>ResourceImpl.java        | util    | Y    |

Source: EMF Eclipse Modeling Framework, Second Edition, Addison Wesley Professional



## More Model Code

| <b>Unit</b> | <b>Description</b> | <b>File name</b> | <b>Subpkg.</b> | <b>Opt.</b> |
|-------------|--------------------|------------------|----------------|-------------|
| Class       | Interface          | <Name>.java      |                | N           |
|             | Class              | <Name>Impl.java  | impl           | N           |
| Enum        | Enum               | <Name>.java      |                | N           |

Source: EMF Eclipse Modeling Framework, Second Edition, Addison Wesley Professional



## Model Edit Code

- User interface independent editor code
- Interfaces to support viewing and editing of model objects
  - Content and label provider functions
  - Property descriptors
  - Command factory
  - Forwarding change notifications
- Sample icons are generated



## More Model Edit Code

| <b>Unit</b> | <b>Description</b>    | <b>File Name</b>                        |
|-------------|-----------------------|---|
| Model       | Plug-in Class         | EditPlugin.java                         |
|             | OSGi Manifest         | META-INF/MANIFEST.MF                    |
|             | Plug-in Manifest      | plugin.xml                              |
|             | Translation file      | plugin.properties                       |
|             | Build properties file | build.properties                        |
| Package     | Adapter Factory       | <Prefix>ItemProviderAdapterFactory.java |
| Class       | Item Provider         | <Name>ItemProvider.java                 |

Source: EMF Eclipse Modeling Framework, Second Edition, Addison Wesley Professional



## Model Editor Code

- User interface specific editor code
- A default tree based editor
  - With toolbar, context menu and menu bar actions for creating an instance of the model
  - Full undo and redo support
- A default model creation wizard
- Icons for the editor and wizard



## More Model Editor Code

| <b>Unit</b> | <b>Description</b>     | <b>File Name</b>                  |
|-------------|------------------------|-----------------------------------|
| Model       | Plug-in Class          | EditorPlugin.java                 |
|             | OSGi Manifest          | META-INF/MANIFEST.MF              |
|             | Plug-in Manifest       | plugin.xml                        |
|             | Translation file       | plugin.properties                 |
|             | Build properties file  | build.properties                  |
| Package     | Editor                 | <Prefix>Editor.java               |
|             | Advisor                | <Prefix>Adviser.java              |
|             | Action Bar Contributor | <Prefix>ActionBarContributor.java |
|             | Wizard                 | <Prefix>ModelWizard.java          |

Source: EMF Eclipse Modeling Framework, Second Edition, Addison Wesley Professional

# Regeneration and Merge

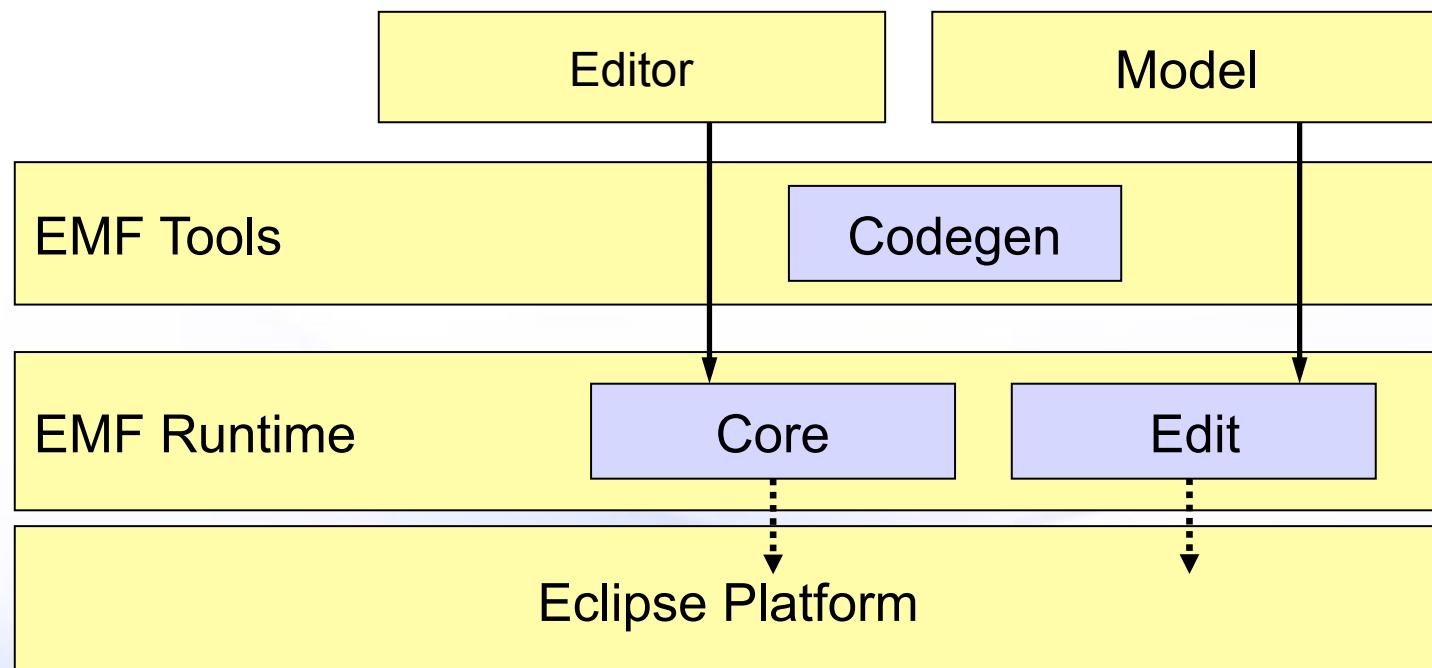
- EMF generator merges with existing code
- Generated elements annotated
  - @generated
  - Will be replaced on regeneration
- Preserving changes
  - Remove @generated tag
  - e.g. @generated NOT
  - Changes will be preserved on regeneration
- Redirection
  - Add Gen to the end of the generated operation



# Summary

- In this module we explored

# EMF Architecture





## EMF Runtime - Core

- Notification framework
- Ecore metamodel
- Persistence
- Validation
- Change model



## EMF Runtime - Edit

- Support for model-based editors and viewers
- Notification framework
  - Sends out notification whenever attribute or reference is changed
  - Observers (also an adapter) receive notification and can act
- Default reflective editor



## EMF Tools - Codegen

- Code generator for application models and editors
  - Interface and class for each class in the model
  - ItemProviders and AdapterFactory for working with Edit framework
  - Default editor
- Extensible model import/export framework
  - Contribute custom model source import modules
  - Contribute custom export modules

# Persistence

- EMF refers to persisted data as a Resource
- Model objects can be spread across resources
  - Proxy is an object referenced in a different resource
- EMF refers to the collection of resources as a ResourceSet
  - Resolve proxies between resources in the set
- Registry maintains the resource factory for different types of resources
  - For the creation of resources of a specific type



# Creating a Resource

- [TM TODO]
- ResourceSet
- Resource.Factory
- Saving Resource
- Loading Resource



## Notification

- [TM TODO]

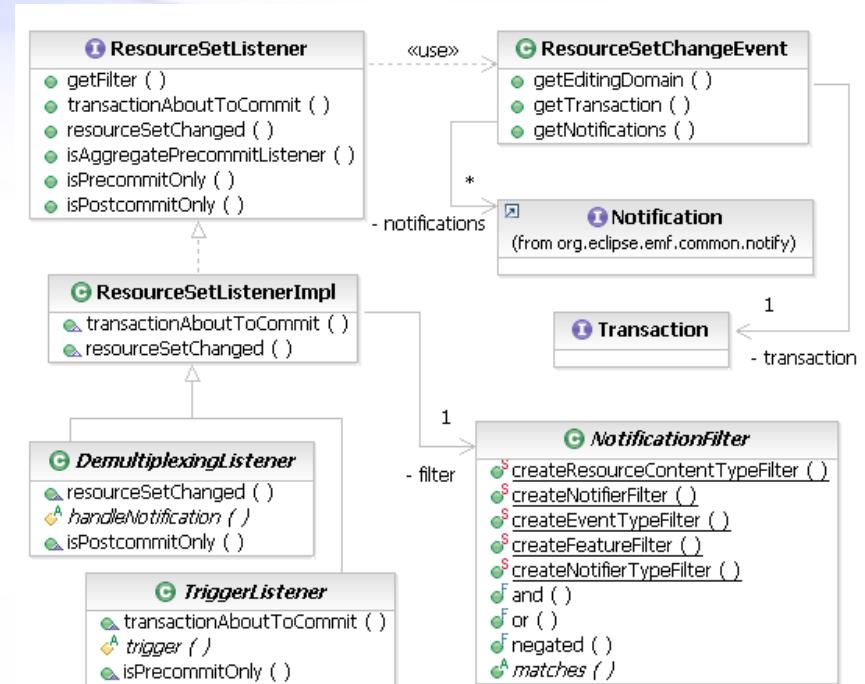


# Change Listeners

- EMF provides an Adapter mechanism to notify listeners when objects change
  - In a transactional environment, though, we can end up reacting to changes only to find that they are reverted when a transaction rolls back
- Enter the ResourceSetListener
  - Post-commit event notifies a listener of all of the changes, in a single batch, that were committed by a transaction
  - If a transaction rolls back, no event is sent because there were no changes
    - There are exceptions for changes that are not (and need not be) undone, such as resource loading and proxy resolution

# Change Listeners

- `ResourceSetChangeEvent` provides the changes that occurred
  - Transaction additionally has a `ChangeDescription` summarizing the changes
- Listeners can declare filters to receive only events of interest to them
- `ResourceSetListenerImpl` is a convenient base class providing no-ops for the listener call-backs





## Change Listeners

- Resource set listeners are added to the transactional editing domain
- Listeners can be registered statically against an editing domain ID
  - Ensures that the listener is attached as soon as the domain comes into being
  - Resolves the problem of timing the addition of listeners



## Post-Commit Listeners

- A post-commit listener just overrides `ResourceSetListenerImpl.resourceSetChanged()`
  - `DemultiplexingListener` implements this by dispatching the notifications one by one to the `handleNotification()` method



# Post-Commit Listeners

- Advantages of the `ResourceSetChangedEvent` include:
  - Listeners know that the changes are permanent
  - Notifications can be processed efficiently as an aggregate
    - Don't need to worry about dependency on "future" changes
  - No further changes can occur while the change event is being dispatched
    - Listeners are invoked in read-only transactions, so that they can safely read the resource set while analyzing the changes
- Listeners need to be aware that notifications are delayed relative to the timing of the changes
  - Notifications are only received after all changes are complete
  - Any given notification may not correspond to the current state of the resource set, depending on subsequent changes



## Pre-Commit Listeners

- Before a transaction closes, pre-commit listeners are notified of the changes performed
  - Listeners can provide additional changes, in the form of commands, to be appended to the transaction
    - As with post-commit listeners, the pre-commit listener is invoked in a read-only transaction, so it does not make changes “directly”
    - These commands implement proactive model integrity, as do triggers in RDBMS. Hence the term “trigger command”
- Trigger commands are executed in a nested transaction
  - This procedure is recursive: the nested transaction also invokes pre-commit listeners when it commits



# Pre-Commit Listeners



## Pre-Commit Listeners

- The TriggerListener class is convenient for processing notifications one by one, where appropriate



# Dynamic EMF

- Working with Ecore models with no generated code
  - Created at runtime
  - Loaded from a ecore resource
- Same behavior as generated code
  - Reflective EObject API
- Model created with dynamic EMF is same as one created with generated code



# Change Recording

- Track the changes made to instances in a model
  - Use the notification framework
- ChangeRecorder
  - Enables transaction capabilities
  - Can observe the changes to objects in a Resource or ResourceSet



# Automatically implemented Constraints

- Multiplicity constraints
  - Enforce the multiplicity modeled in the Ecore model
- Data type values
  - Ensure that the value of an attribute conforms to the rules of the data type



## Validator

- Invokes the invariants and constraints
- Invariants are defined by <<inv>> operations on a class
- Constraints are defined using a Validator



# EMF Utilities

- Copying
  - `EcoreUtil.copy`
- Equality
  - `EcoreUtil.equal`
- Cross-referencing, contents/container navigation, annotation, proxy resolution, adapter selection,  
...



# Summary

- In this module we explored



## EMF Transaction

- A framework for managing multiple readers and writers to EMF resources or sets of resources
- An editing domain object manages access to resources
  - Can be shared amongst applications
- A transaction object is the unit of work
- Support for rolling back changes
  - Support for workbench undo/redo infrastructure



## Read/Write Transactions

- Gives a thread exclusive access to a ResourceSet to modify its content
  - To change the Resources within the ResoureSet
- Prevent other threads from observing incomplete changes



## Read Transactions

- Reading a ResourceSet sometimes causes initialization to happen
  - e.g. proxy resolution
- Need to protect against concurrent initialization by simultaneous reads
- A read transaction protects the ResourceSet during simultaneous reads



# Change Events

- When a transaction is committed change notifications are sent to registered listeners
  - Includes a summary of changes
- Successful commits send notifications as a batch
  - Prevents listeners from being overwhelmed



# Workbench Integration



## Obtaining the Editing Domain

- Any editor that needs to access the registered editing domain simply gets it from the registry
  - It is lazily created, using the associated factory, on first access



# Creating Read/Write Transactions

- To do work in a read/write transaction, simply execute a Command on the TransactionalCommandStack
  - Just like using a regular EMF Editing Domain
- If the transaction needs to roll back, it will be undone automatically and will not be appended to the stack
  - In order to find out when rollback occurs, use the `TransactionalCommandStack::execute(Command, Map)` method, which throws `RollbackException`, instead of using `CommandStack::execute(Command)`
  - This method also accepts a map of options to configure the transaction that will be created (more on options, later)



# RecordingCommands

- The RecordingCommand class is a convenient command implementation for read/write transactions
  - Uses the change information recorded (for possible rollback) by the transaction to “automagically” provide undo/redo



# Transaction Options

- The `TransactionalCommandStack::execute()` method accepts a map of options defined by the `Transaction` interface that determine how changes occurring during the transaction are handled:
  - `OPTION_NO_NOTIFICATIONS`: changes are not included in post-commit change events
  - `OPTION_NO_TRIGGERS`: changes are not included in pre-commit change events
  - `OPTION_NO_VALIDATION`: changes are not validated
  - `OPTION_NO_UNDO`: changes are not recorded for undo/redo and rollback. Use with extreme caution!

# Transaction Options

- (continued from previous slide)
  - `OPTION_UNPROTECTED`: implies `OPTION_NO_UNDO`, `OPTION_NO_VALIDATION`, and `OPTION_NO_TRIGGER`. In addition, permits writing to the resource set even in an otherwise read-only context. Use with even more extreme caution!
- The `CommandStack::undo()` and `::redo()` methods use the following options for the undo/redo transaction:
  - `OPTION_NO_UNDO`: because we are undoing or redoing a previous recording, there is no need to record anew
  - `OPTION_NO_TRIGGER`: triggers performed during execution were recorded and are automatically undone; any additional changes would be inappropriate
  - `OPTION_NO_VALIDATION`: there is no need to validate a reversion to a previous state of the data



# Transaction Options

- The pre-defined options only apply to read/write transactions
  - Permitted on read-only transactions but have no effect
- Extensions of the transaction API can define custom options



## Creating Read-Only Transactions

- To read the contents of the resource set safely, use the `runExclusive()` API:

# Transaction Sharing

- Read-only transactions on multiple threads can be interleaved by cooperatively yielding their read lock
  - Recommended for long-running read operations
  - Call `TransactionalEditingDomain::yield()` to yield read access to other threads waiting for read-only transactions
  - Yielding thread waits until other threads return read access to it either by finishing their transactions or by yielding
  - Yielding is fair: read access is always passed to the thread that has waited longest
- Write access is not shared in this way
  - Readers cannot yield to writers
  - Writers cannot yield to any others



# Transaction Sharing



# Transaction Sharing

- A thread that owns a transaction can lend it to another thread using a `PrivilegedRunnable`
  - The privileged runnable takes over the transaction for the duration of its `run()` method
  - Can be used to share read-only *and* read-write transactions
- Ideal for runnables that need to access the resource set and the UI thread at the same time:
  - Pass the privileged runnable to `Display.syncExec(Runnable)` API to run on the UI thread
- Must only be used with synchronous inter-thread communication such as `syncExec`
  - The originator thread loses the transaction during the runnable



# Transaction Sharing



## Transaction Validation

- When a read/write transaction commits, all of the changes that it performed are checked using the Validation Framework's live validation capability
  - If problems of error severity or worse are detected, then the transaction rolls back
- Pre-commit listeners can also force the transaction to roll back by throwing a `RollbackException`
  - If a pre-commit listener cannot construct the command that it requires to maintain integrity, then it should roll back

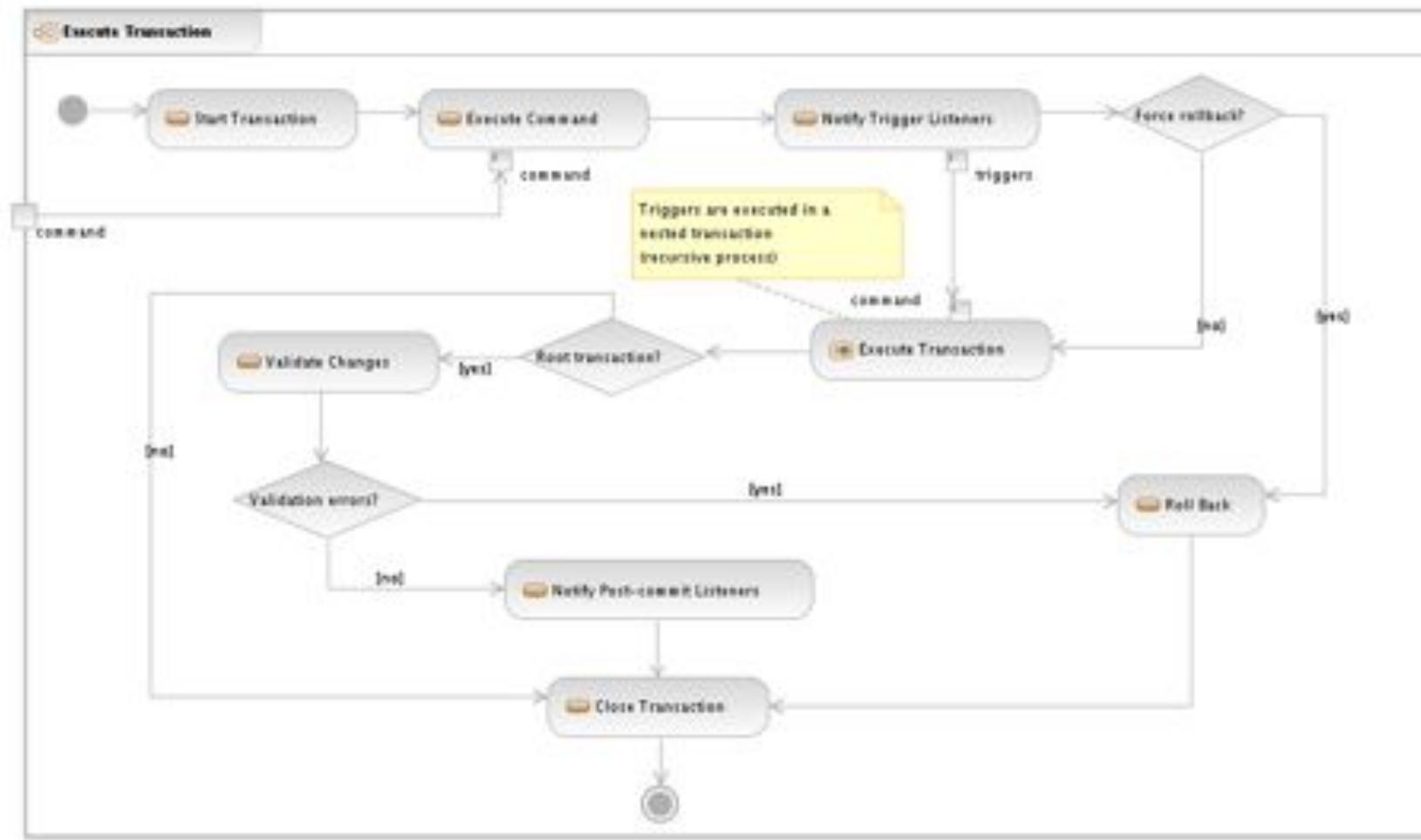
# Transaction Nesting

- Both read-only and read-write transactions can nest to any depth
- Post-commit events are sent only when the root transaction commits
  - Because even after a nested transaction has committed, it can be rolled back if its parent (or some ancestor) rolls back
- Pre-commit events are sent at every level of nesting
  - Because a parent transaction may assume data integrity conditions guaranteed by triggers when it resumes
- Validation is performed on all changes when a root transaction commits
  - Because triggers must be invoked first, in nested transactions



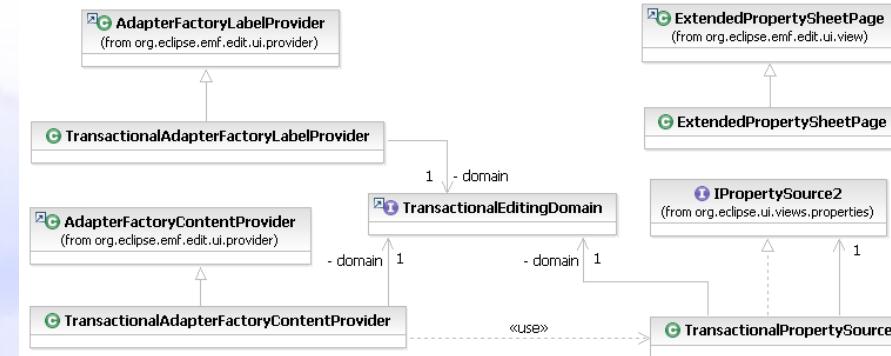
# Transaction Nesting

- Nested transactions inherit options from their parents
  - The standard options cannot be disinherited. e.g., if a parent transaction does not send post-commit notifications, then none of its descendants will, either, even if they explicitly specify `Boolean.FALSE` for that option
- Nested transactions can, however, apply more options than their parents
  - e.g., a child transaction can disable notifications. When its parent commits, the changes that it reports will simply exclude any that occurred during the execution of the child
- The inheritance of custom options in an extension of the transaction API is defined by that extension



# UI Utilities

- The Transaction API includes some utilities for building transactional editors
  - Use the editing domain to create read-only and/or read-write transactions as necessary
  - Substitute for the default EMF.Edit implementations





# EMF Query

- Framework for executing queries against any EMF based model
  - Java API
  - Customizable
- EMF model query
  - SQL like queries
  - `SELECT stament = new SELECT(new FROM(queryRoot), new WHERE(condition));`
- OCL support in query
  - Condition expressed in OCL
  - `self.member.oclTypeOf(uml::Property)`
    - Get all the Properties of a Classifier



# Query Statements

- SELECT statements filter the objects provided by a FROM clause according to the conditions specified in the WHERE clause
- SELECTs are IEOObjectSources, so they can be used in FROM clauses to nest queries
- UPDATE statements use a SET clause to update the objects provided by the FROM clause (filtered, of course, by the WHERE clause)



## The FROM Clause

- Uses EMF's tree iterators to walk the objects being queried
- Optionally specifies an `EObjectCondition` filter
- Search scope is encapsulated in an `IObjectSource`
  - provides objects via an iterator. The `FROM` descends into the contents of these objects if it is a hierarchical `IteratorKind`



## The WHERE Clause

- The WHERE clause specifies a single filter condition
- This filter condition can be arbitrarily complex
- Filters can be combined in innumerable ways using the common boolean operators
- The IN filter detects whether an object is an element of a supplied set (as in SQL)



## The WHERE Clause

- The framework provides a condition on the model type of objects
- Can also filter for objects that are identical to some target (`EObjectInstanceCondition`)



## The WHERE Clause

- The framework provides conditions that filter objects based on values of their features
- In particular, the `EObjectReferenceCondition` filters for cross-references to a particular object



# The WHERE Clause: Condition Policies

- For multi-valued structural features, the ConditionPolicy determines whether a Condition must match all values or just any value
  - Correspond to the relational  $\forall$  (for-all) and  $\exists$  (exists) quantifiers



# The WHERE Clause: Prune Handlers

- Just as with EMF's tree iterators, a query's iteration over its scope can be *pruned*
- Any `EObjectCondition` can have a `PruneHandler` that determines whether the search tree can be pruned
  - This allows the query to skip over entire sub-trees when a condition knows that it will never be satisfied for any of the contents of the current object
- The default prune handler in most cases is NEVER which, as the name implies, never prunes

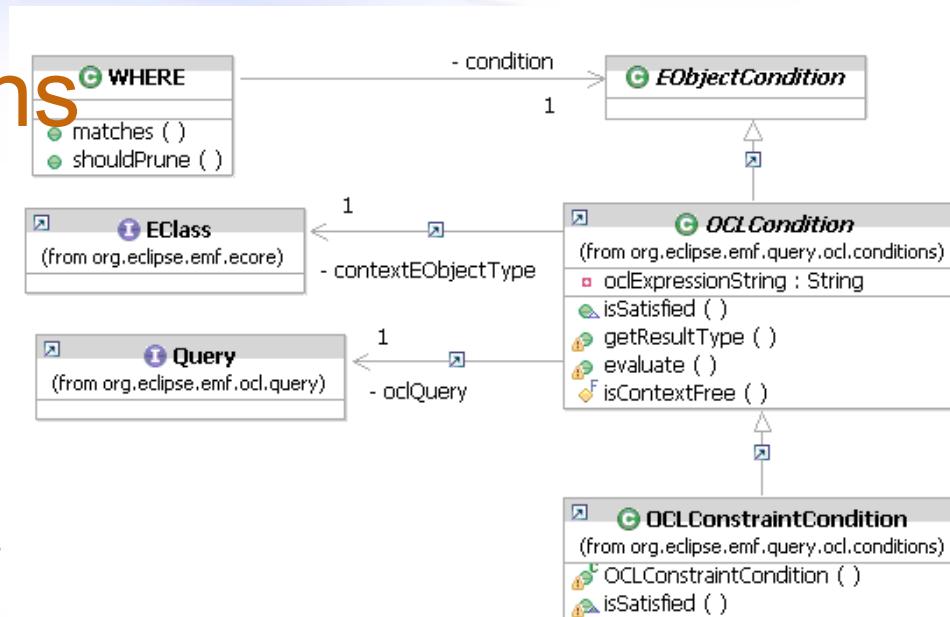


# The WHERE Clause: Other Conditions

- The framework includes a variety of conditions for working with primitive-valued EAttributes
  - Including strings, booleans, and numbers of all kinds
- Adapters convert inputs to the required data type
  - Default implementations simply cast, assuming that the values already conform
  - Can be customized to convert values by whatever means is appropriate

# OCL Conditions

- OCL can be used to specify WHERE clause conditions
- Only available when the OCL component of MDT is installed
- An OCLConstraintCondition specifies a boolean-valued expression (i.e., a constraint) that selects those elements for which the expression is true
- OCL expressions can be contextual or context-free





# The UPDATE Statement

- The UPDATE statement behaves much like a SELECT, except that it passes its result objects through the client-supplied SET clause
- The result of the UPDATE is the subset of the selected objects for which the SET clause returned true (indicating that they were, in fact, modified)



## SELECT Query in Action

- Queries can be used for anything from simple search functions to complex structural analysis
- They can even be used to implement validation constraints!



## SELECT Query with OCL Condition

- Context-free OCL conditions are applied to any element on which they can be parsed



# UPDATE Query in Action



# Summary

- In this module we explored





# Uniform Resource Identifier (URI)

- A formatted string that serves as an identifier for a resource
- Syntax:
  - Generic  
 $[scheme:]scheme-specific-part[#fragment]$
  - Hierarchical  
 $[scheme:][/authority][path][?query][#fragment]$
- Used in EMF to identify a resource, an object in a resource, or an Ecore package (namespace URI)



- The idea is to “decorate” the fragment portion of a URI with details that further describe the referenced object
  - The description can be used by a service to locate the object or to enrich an error message to be presented to the user
- Query data
  - Added to the fragment portion of the object’s URI
  - Must be at the end of the URI fragment portion
  - Is delimited by “?”
  - Example:  
`file:/c:/dir/library.xmi#//@books.0?Book.ISBN.0131425420?`



- If you are using a XMLResource to serialize your objects
  - Override the `getURIFragmentQuery(Resource, EObject)` method of `org.eclipse.emf.ecore.xmi.impl.XMLHelperImpl` to return the query that should be serialized for the given EObject
  - Override the `createXMLHelper()` method of `org.eclipse.emf.ecore.xmi.impl.XMLResourceImpl` to return an instance of your XMLHelper
- The `getURIFragmentQuery(...)` method returns the string that is used as the “query”
  - The string should not contain the delimiter character (?)
  - A null string indicates that there is no query
  - The characters must be valid URI fragment characters (as defined by the specification)



## Cross-Resource Containment

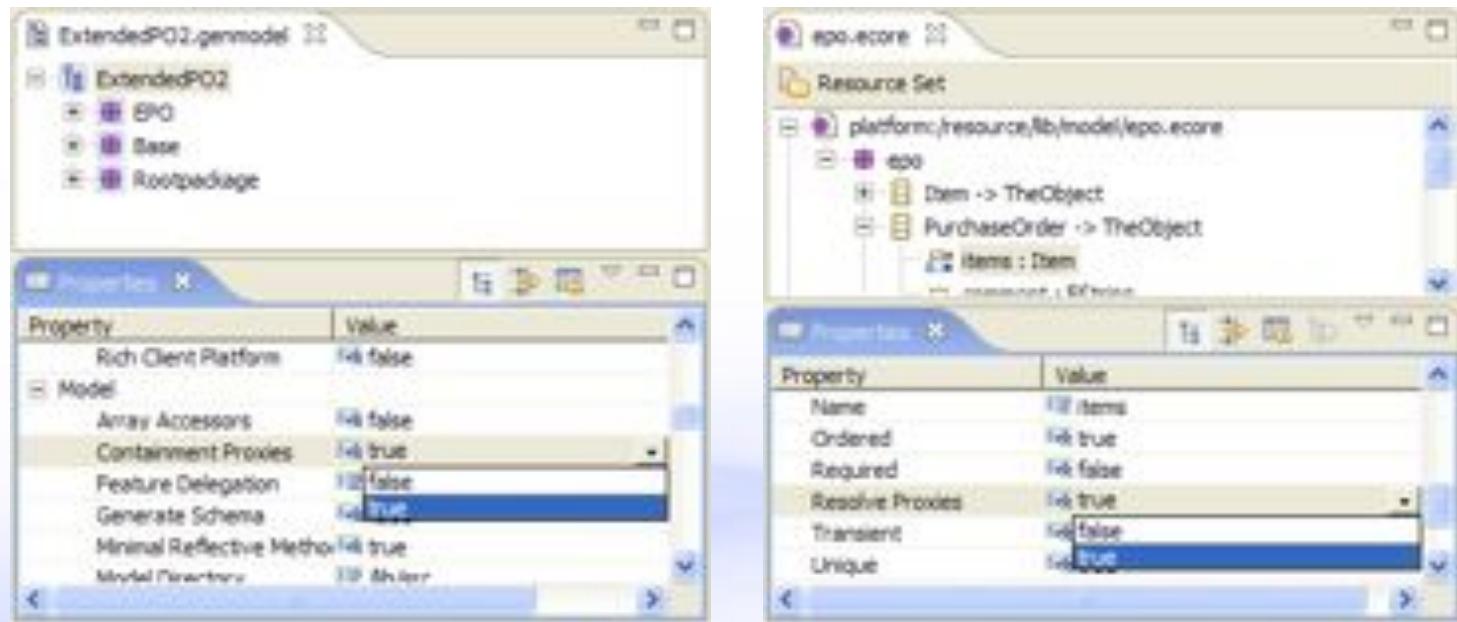
- Allows an object hierarchy to be persisted across multiple resources
  - `eObject.eResource()` may be different from `eObject.eContainer().eResource()`
- Must be explicitly enabled
  - The containment reference has to be set to resolve proxies
  - Also, on generated models, the value of the “Containment Proxies” generator model property has to be set to ‘true’

# Enabling Cross Document Containment

Dynamic Model

```
EReference houses = EcoreFactory.eINSTANCE.createEReference();
houses.setName("houses");
houses.setEType(house);
houses.setUpperBound(ETypedElement.UNBOUNDED_MULTIPLICITY);
houses.setContainment(true);
houses.setResolveProxies(true);
person.getEStructuralFeatures().add(houses);
```

Generated Model





# Resource Tips & Tricks

- Unloading a resource
  - `Resource.unload()`
  - EMF's default implementation performs the following steps:
    - Sets the *loaded* attribute to false;
    - Caches an iterator with all the proper contents of all objects held by the resource
    - Clears the resource's content, error and warning lists
    - Turns each object returned by the iterator into a proxy and clears its adapter list
    - Fires a notification
      - Feature ID = `Resource.RESOURCE_IS_LOADED`
  - You may also remove the resource from the resource set

# Resource Tips & Tricks

- Tracking modification
  - `Resource.setTrackingModification(boolean)`
  - When activated, EMF's default implementation performs the following steps:
    - Instantiates an Adapter and registers it on all proper objects
      - The adapter calls `Resource.setModified(boolean)` after receiving a notification
    - Registers the adapter on any object added to the resource and deregisters it from objects that are removed
  - When deactivated, the default implementation removes the adapter from the objects
  - You can manually define what is the `isModified` state of a resource



# Resource Tips & Tricks

- Am I loading something?
  - `Resource.Internal.isLoading()`
    - Every Resource is supposed to implement the `Resource.Internal` interface
  - When a resource is being loaded the `isLoading()` method returns true



# Going Deeper: Why is the Generator Model so important?

- The generator model acts as a decorator for an Ecore model
  - It provides details that would pollute the model, such as the
    - Qualified name of an EPackage
    - Prefix for package-related class names
    - Actual location of the model, edit, editor, and test source folders
- When a modeled domain is converted into an EMF model, the importer may be able to capture some generator model details and store them in a .genmodel file
  - The Java package of the annotated Java interfaces
  - Referenced XML Schemas that may or may not be already represented by Ecore models
- The generator model is useful to an exporter because
  - It can be used to persist details about the exported artifact
  - Some details may be important to properly describe the modeled domain





# Exploring UML in Eclipse



## What is the UML2 project?

- An EMF based implementation of the UML 2.x specification
- A base for modeling tools to build upon
- With support for UML Profiles



# EMF Implementation of UML2 Spec

- What many consider the reference implementation for the UML 2 specification
- Metamodel completely specified as an EMF model
- Support for many of the UML techniques
  - Redefinition
  - Subsetting



## Base for Modeling Tools

- Tools share a common foundation improving
  - Model interchange
  - Add-on tools, for example model analysis
  - Tool independent transformations
- Shared interpretation of the UML 2.x specification
- Models serialized to common XML format



# Support for UML Profiles

- Profiles are UML 2.x extensibility and customization mechanism
  - Use domain concepts
  - Refining semantics
  - Customize presentation
  - Tagging model elements
  - Add domain specific information
- Enables the definition of domain specific languages
  - For example Rose Real-Time in RSA

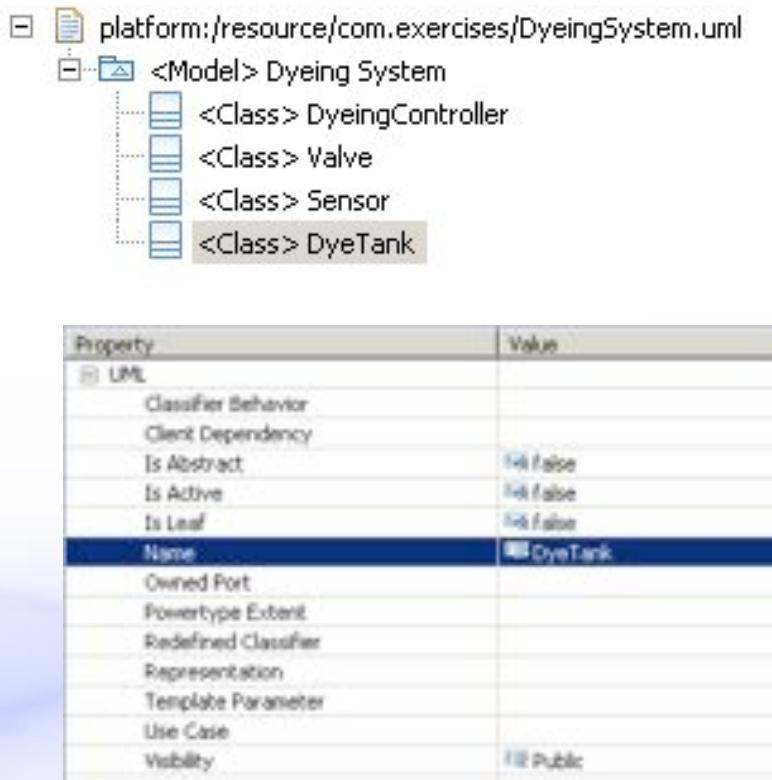


# Creating a UML model

- Default editor
  - Tree based editor
- Rational Modeling Platform
  - Visual modeling
- Programmatically
  - Use the Factory for the UML 2 model

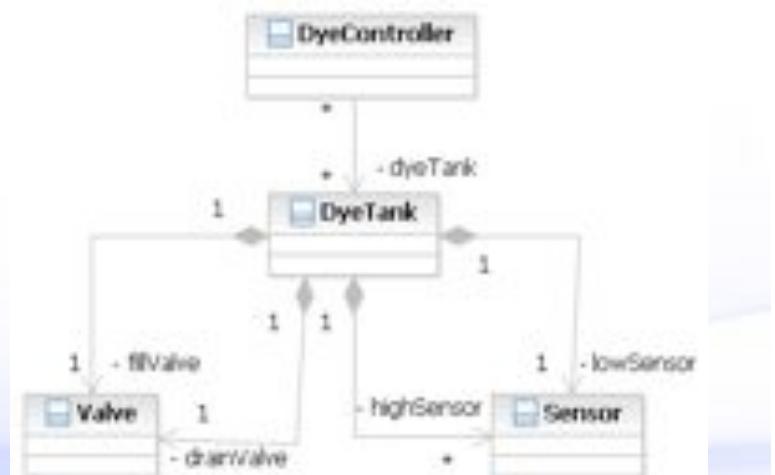
# UML Model Editor

- **File → New...** from the main menu
- Select UML Model
  - Under Example EMF Model Creation Wizards
- Provide a name
- Set Model Object to Model
- Open with UML Model Editor
- Use Create Child in the context menu to create model elements



# Rational Modeling Platform

- Create a UML 2 model graphically
- How to
  - File → New... from the main menu
  - UML Model
    - Under Modeling
    - Set the model name
- Create elements using diagrams and project explorer
- Created model can be opened in the UML Model Editor



# Creating Models in Code

- Programmatically create UML models and elements
- Use of the standard EMF generated API
- EMF reflective capabilities available

```
// Create resource to add model to
ResourceSet resourceSet = new ResourceSetImpl();
Resource modelResource
    = resourceSet.createResource(URI.createURI("dyeingSystem.uml", true));

// Create UML model and set name
Model dyeingSystemModel = UMLFactory.eINSTANCE.createModel();
dyeingSystemModel.setName("DyeingSystem");

// Add model to the resource
modelResource.getContents().add(dyeingSystemModel);
```



## More Creating Models in Code

- To create UML model elements use
  - UMLFactory, or
  - For Packages and Models
    - createdOwnedClass
    - createNestedPackage
    - createdOwnedPrimitiveType
    - createdOwnedEnumeration
    - ...



# Persistence and Serialization

- Models are persisted in an XMI compliant format
- Tools built on top of UML2 project should be able to load the model
  - Likely won't maintain diagrams

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
    xmlns:uml="http://www.eclipse.org/uml2/2.1.0/UML" xmi:id="_AjSRsJutEd2e_u-cIXQ8Q" name="Dyeing System">
    <packagedElement xmi:type="uml:Class" xmi:id="_P7Ds8JutEd2e_u-cIXQ8Q" name="DyeingController"/>
    <packagedElement xmi:type="uml:Class" xmi:id="_UaWiwJutEd2e_u-cIXQ8Q" name="Valve"/>
    <packagedElement xmi:type="uml:Class" xmi:id="_XswusJutEd2e_u-cIXQ8Q" name="Sensor"/>
    <packagedElement xmi:type="uml:Class" xmi:id="_cTgQUJutEd2e_u-cIXQ8Q" name="DyeTank"/>
</uml:Model>
```

# Summary

- In this module we explored
  - What the UML2 project is
  - How to create UML2 models
    - UML Model Editor
    - Rational Modeling Platform
    - Through code
  - Persistence

# Extending UML

- UML 2 is a general purpose modeling language
  - Large and expressive
- Often specific domains need to extend it
  - Additional concepts
  - Restricting metamodel
  - Defining domain specific semantics for elements
- Different approaches
  - Feather weight - Tagging a model
  - Light weight - UML Profile
  - Heavy weight - Extending the metamodel



# Tagging a model with keywords

- Lightweight approach to adding data to a model
  - Control code generators
  - Categorizing
- Create by
  - Adding Annotation with source set to UML
  - Add details to the Annotation with key being the keyword
- Simple API to retrieve keywords
  - addKeyword, removeKeyword, and hasKeyword



## Extending UML with a Profile

- UML profiles provide a lightweight approach to extending the UML metamodel
- UML profiles can be created in the same way as UML models
  - UML Model Editor
  - Rational Modeling Platform
  - Programmatically
- Profiles can be published or registered
  - Makes them accessible to others
  - Approach used by RSA RTE



## UML Profile

- Primary construct in a profile is a stereotype
  - Extends one or more metaclasses from the UML
  - May add information and/or constraints
  - May add/or constrain semantics
  - May change graphical display
- Can be **applied** to one or more UML models or packages
  - Stereotypes applied to model or package and its contents

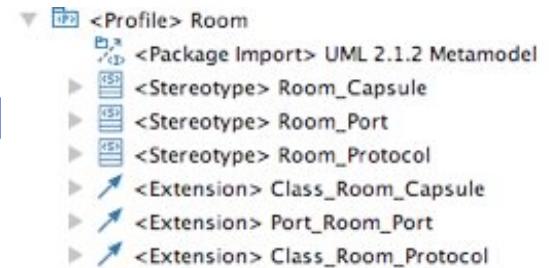
# Creating UML Profile

- Using UML Model Editor
- Select **File → New...**
- Choose UML Model and provide a name
  - Under Example EMF Model Creation Wizards
- Set Model Object to Profile
- Open with UML Model Editor
- Select **UML Editor → Profile → Reference Metamodel...** from the main menu
- Choose the UML metamodel

```
▼ platform:/resource/com.zeligsoft.training.exercises.uml/UMLRT.profile.uml
  ► pathmap://<Profile> Room
  ► pathmap://UML_METAMODELS/UML.metamodel.uml
  ► pathmap://UML_PROFILES/Ecore.profile.uml
  ► pathmap://UML_PROFILES/Standard.profile.uml
  ► pathmap://UML_METAMODELS/Ecore.metamodel.uml
  ► pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml
```

# Creating Stereotype

- Open profile in UML Model Editor
- Select the profile object and **New Child**  
→ **Owned Stereotype** → **Stereotype**  
from the context menu
- Select the stereotype and **UML Editor**  
→ **Stereotype** → **Create Extension...**  
from the main menu
- Select the UML metaclasses to extend
- Creates Extension object in the profile





# UML Profile Static vs. Dynamic

- **Dynamic Profile**
  - The profile is defined in a model whose model object is a Profile
  - No code is generated
  - Profile is deployed in a plug-in and registered as a dynamic package
- **Static Profile**
  - The profile is defined in a model whose model object is a Profile
  - An API for the profile is generated to make it easier to work with the profile in code
  - Profile code is deployed in a plug-in and registered as a static package



# Deploying a Dynamic Profile

- Define the profile
  - Converts the profile elements into Ecore representation
  - Select the profile element in the model
  - Select **UML Editor** → **Profile** → **Define** from main menu
  - This stores Ecore representation as an annotation in the profile
- Make the project a plug-in project and make sure the profile model is in the build
- Register the profile
  - `org.eclipse.uml2.uml.dynamic_package`



## Deploying a Static Profile

- Generate the profile code
- Make the project a plug-in project and make sure the src folder with the generated code is in the build
- Register the profile
  - `org.eclipse.uml2.uml.generated_package`



# Generating Profile Code

- Apply Ecore profile to the profile object
- Apply ePackage stereotype to the profile object
  - Set the following ePackage properties
    - NS URI
    - NS Prefix
    - Base Package
- Create an EMF model from the profile using the UML model importer
- Configure generator settings
- Generate Model code for the EMF model



# Summary

- In this module we explored

# Applying a UML profile

- A Profile is applied to a Model or Package
- Open model in UML Model Editor
- Select model or package object and **UML Editor → Package → Apply Profile...** from the main menu
- Choose the profiles to apply
- Creates a Profile Application object in the package





## Applying a UML Profile - Code

```
// Load the resource containing the profile
Resource roomProfileResource
    = resourceSet.getResource(ROOM_PROFILE_URI, true);
Profile roomProfile =
    (Profile) roomProfileResource.getContents().get(0);

// Apply the profile to a model
dyeingSystemModel.applyProfile(roomProfile);
```

# Applying a Stereotype

- Stereotypes are applied to elements in a model
  - More than one can be applied
  - Applicable elements defined by Stereotypes metaclass extensions
  - Attributes of Stereotypes can be set in the properties view of the UML Model Editor
- Select the element in the editor and **UML Editor** ⇒ **Element** ⇒ **Apply Stereotype...** from the main menu

|                       |  |
|-----------------------|--|
| RT Port               |  |
| Is Cordugate          | <input type="checkbox"/> False                         |
| Is Notification       | <input type="checkbox"/> False                         |
| Is Publish            | <input type="checkbox"/> False                         |
| Is Wired              | <input type="checkbox"/> False                         |
| Registration          | <input type="checkbox"/> Automatic                     |
| Registration Override | <input type="checkbox"/>                               |
|                       |  |
| UML                   |  |
| Aggregation           | <input checked="" type="checkbox"/> Composite          |
| Association           | <input type="checkbox"/>                               |
| Class                 | <input type="checkbox"/> <<HelloWorld>> <<HelloWorld>> |
| Client Dependency     | <input type="checkbox"/>                               |
| Default               | <input type="checkbox"/>                               |
| End                   | <input type="checkbox"/>                               |
| Is Behavior           | <input checked="" type="checkbox"/> true               |
| Is Derived            | <input checked="" type="checkbox"/> false              |
| Is Derived Union      | <input checked="" type="checkbox"/> false              |
| Is Leaf               | <input checked="" type="checkbox"/> false              |
| Is Ordered            | <input checked="" type="checkbox"/> false              |
| Is Read Only          | <input checked="" type="checkbox"/> false              |
| Is Service            | <input checked="" type="checkbox"/> false              |
| Is Static             | <input checked="" type="checkbox"/> false              |
| Is Unique             | <input checked="" type="checkbox"/> true               |
| Lower                 | <input type="checkbox"/> 1                             |
| Name                  | <input type="checkbox"/> log                           |
| Protocol              |  |



## Applying a Stereotype - Code

```
// Get the Capsule stereotype object from the profile
// and apply it to a class object
Stereotype capsuleStereotype
    = roomProfile.getOwnedStereotype("Room_Capsule");

org.eclipse.uml2.uml.Class dyeTank =
    dyeingSystemModel.createOwnedClass("DyeTank", false);

dyeTank.applyStereotype(capsuleStereotype);
```



# Working Stereotype Properties

Accessing a stereotype value

```
drainValvePort.getValue(portStereotype, "conjugated");
```

Setting a stereotype value

```
drainValvePort.setValue(portStereotype, "conjugated", false);
```

Adding to a stereotype list property

```
((List) dyeSystemComponent.getValue(componentStereotype, "includes"))
    .add("MyInclude.h");
```



# Summary

- In this module we explored
  - Applying a profile
  - Applying stereotypes to model elements
  - Working with profiles in code
    - Applying profile
    - Applying stereotype
    - Getting/Setting stereotype values



# Keywords in RSA-RTE

- [TM TODO]



# Profiles in RSA-RTE

- Profiles are used extensively in RSA-RTE
  - Extending/constraining the UML to UML-RT semantics and notation
    - UMLRealTime profile
    - UMLRealTime model libraries for things like Frame, and Log
  - Adding information to model elements to generate code
    - C++ support through CPPPropertySets profile
    - C++ model libraries for
- When a RSA-RTE model is created
  - UMLRealTime profile is applied
  - CPPPropertySets profile is applied
  - RTClasses, RTComponents and CPPPrimitiveDatatypes libraries



# Creating Profiles in RSA-RTE

- Model the domain
  - Concepts, attributes and relationships in a Profile
  - Complimentary model libraries
  - Constraints - OCL or Java
- Publish the Profile
  - Helps RSA-RTE with versioning and migration
- Register the Profile in a plug-in
  - Make it available for RSA-RTE to apply it
- Can export to open source UML 2

# Profile

- **File → New → Other...**
- Select UML Profile or UML Profile Project...
  - Under Modeling → UML Extensibility
- Set name and import the UML Primitive Types library
- Creates a Profile object that already imports the UML metamodel
- Add stereotypes, classes and relationships

# Releasing a Profile

- RSA-RTE provides a release capability for profiles
  - A released profile is to be additive otherwise backwards compatibility may not work
- To release a profile
  - Select Release in the context menu of the profile object
  - Associate a label with the release
  - Be careful how often a profile is released





# Publishing a Profile

- Publishing a profile makes it available for others to install and use in their RSA-RTE models
- To publish
  - Make the project a plug-in project
  - Add the profile file to the build
  - Extend “com.ibm.xtools.uml.msl.UMLProfiles”
    - Enables RSM to find the profile
  - Publish the plug-in

```
<extension point="com.ibm.xtools.uml.msl.UMLProfiles">
    <UMLProfile id="com.zeligsoft.exercises.profiles.room"
        name="ROOM"
        path="pathmap://EXERCISE_PROFILES/Room.epx" required="false" visible="true" />
</extension>
```

# Model Libraries

- Model libraries provide a mechanism for publishing a set of model elements for reuse
  - e.g. Data types, reusable components
- It is a model with the `modelLibrary` stereotype applied
  - `modelLibrary` is part of the Standard profile
- The elements will be read-only in the model that imports the library



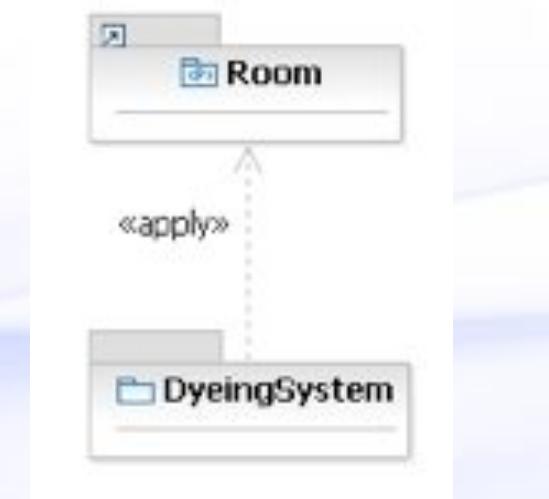
# Publishing a Model Library

- Publishing a model library makes it available for others to import and use in their RSA-RTE models
- To publish
  - Make the project a plug-in project
  - Add the model file to the build
  - Extends com.ibm.xtools.uml.msl.UMLLibraries
    - Enables RSM to find the profile
  - Publish the plug-in

```
<extension point="com.ibm.xtools.uml.msl.UMLLibraries">
    <UMLLibrary
        name="RoomServices"
        path="pathmap://EXERCISE_MODEL_LIBRARIES/RoomServices.emx">
    </UMLLibrary>
</extension>
```

# Applying Profiles in RSA-RTE

- On a Model or Package “Add Profile...”
- Apply Stereotype to Elements in the model
- Set attributes of the Stereotype
  - Advanced tab
  - Stereotype tab





# Summary

- In this module we explored



## Transactions with RSA-RTE

- Recall that transactions ensure that a command on the model is executed exclusively
- And are useful for integrating with the workbench's undo/redo infrastructure
- To get the editing domain used by RSA-RTE
  - Use the UMLModeler object  
`UMLModeler.getEditingDomain()`





# Eclipse Transformation Technologies



# Overview

Version 0.1, Toby McClean, Zeligsoft, 2008



# Agenda



# Model Transformation

- Model transformation is the creation of one or more target artifacts from one or more source models
- Model transformations are used for
  - Tool integrations
  - Model refinement, abstraction and refactoring
  - Code generation
  - Documentation generation and reporting



# Model Transformation

- Typically we talk about two forms
  - Model to model transformations (M2M)
  - Model to text transformations (M2T)



# M2M Transformations

- An M2M transformation creates one or more **target models** from one or more **source models**
- Classifying M2M transformations
  - Horizontal
  - Vertical
  - Bi-directional
  - In place



# More M2M Transformations

- Eclipse Technologies
  - Java
  - M2M Project - **QVT**, xTend and ATL
- RSA-RTE Technologies
  - Rational Transformation Engine



# Model to Text Transformations

- A M2T transformation creates one or more text based artifacts from one or more source model
- Typically template based
  - Create a template from an example of the desired artifact
- M2T consideration
  - Often it is best to use M2T with a source model that is dedicated to the transformation
- Eclipse technologies
  - M2T Project - JET2 and xPand



# Summary

Version 0.1, Toby McClean, Zeligsoft, 2008



# Eclipse Transformation Technologies

Model to Model Transformations

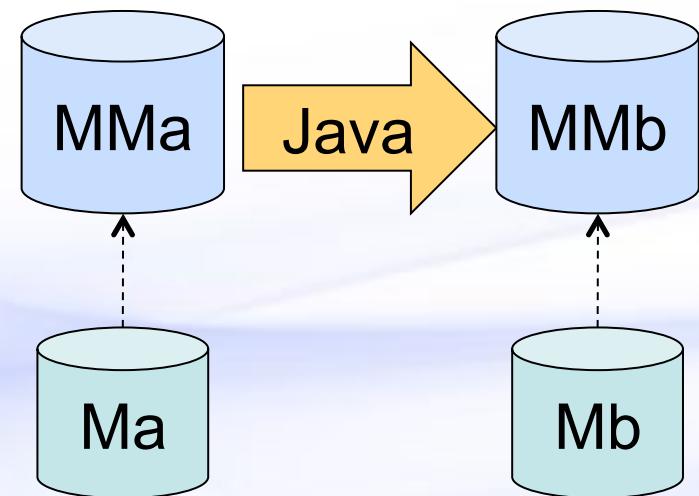


# Overview

- In this section we will explore the M2M technologies available in Eclipse
- The specific technologies are
  - M2M with Java
  - M2M with QVT

## M2M with Java

- The most basic approach to M2M is to use straight Java with the EMF generated API
- Transformation writer has full power or a 3GL
  - Development tools
  - Debugging facilities



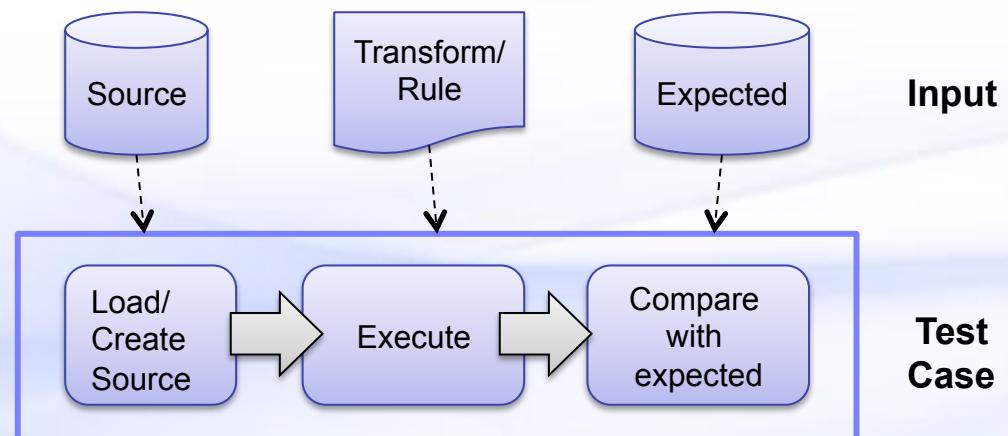


# Developing a Transformation

- Develop the transformation as if you were writing a Java application
- Model navigation
  - EMF Reflective API
  - Metamodel specific API generated by EMF
- Transformation rules
  - Rule constraints
  - Target element construction

# Testing Transformations

- Traditional Java test techniques
  - Hand crafted test framework
  - JUnit
- Using JUnit
  - Automatable
  - Repeatable





# Executing the Transformation

- Use workbench Java execution capabilities
  - We can use the Run As → Java Application
  - We can debug using Debug As → Java Application
  - To pass transformation parameter values
    - Use command line
    - Build a user interface
- Integrate with workbench
  - Provide an **action** to invoke from editor and/or view
  - Provide a transformation **resource with action** to execute

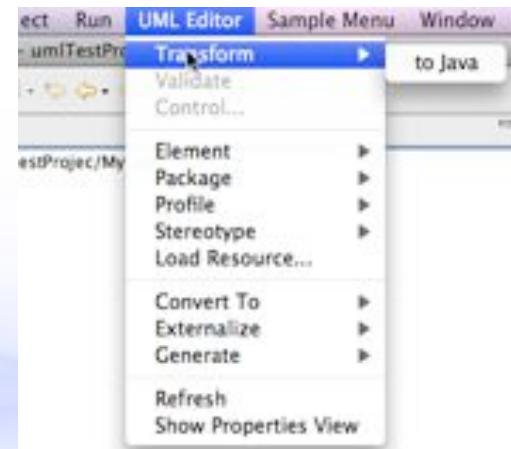


# Integrating with the Workbench

- Lets look at adding an action to the UML Model Editor to transform a UML element into a Java abstract syntax model
- Contribute an action to the editor's menu
- Implement behavior for the action
- Test
  - In runtime workbench
- Deploy
  - Publish the plug-in

# Contribute Action to the Editor

- UML Model Editor ID
  - org.eclipse.uml2.uml.editor.presentation.UMLEditorID
- Menu bar path
  - org.eclipse.uml2.umlMenuID
    - settings
    - actions
    - additions
    - additions-end





# Integrating with RSA-RTE

- Contribute an action
  - to the Project Explorer context menu
  - to the Diagram Editor context menu
- Implement action handler
  - Implement `IEditorActionDelegate`
- Test
  - In runtime workbench
- Deploy
  - Publish the plug-in



# Java Transformation Example

Version 0.1, Toby McClean, Zeligsoft, 2008



# Java Transformation Considerations

- Transformation Architecture
  - How flexible is it
  - Rules – classes vs. methods
  - Extensibility
- Model merge
- Transactions
- Traceability must be managed by transformation developer

# Summary

- In this module we explored
  - M2M transformations with Java
  - Integrating Java M2M transformations with
    - UML Model Editor
    - RSA-RTE
  - Considerations when implementing M2M with Java

# Query/View/Transformation (QVT)

- Features
  - What is it?
  - Where does it come from? E.g. OMG Standard
- Editor
  - Content Assist
- Executing
  - Manually
  - With ANT
- Integrating with RSx



# Query/View/Transformation (QVT)

- An OMG specification for transforming querying and transforming models
- Two languages
  - Operational Mapping Language (OML)
    - Procedural/Imperative language to mapping and query definition
    - Supported in M2M project
  - Relational Language
    - Declarative language for mapping definition
    - Being developed as part of the M2M project



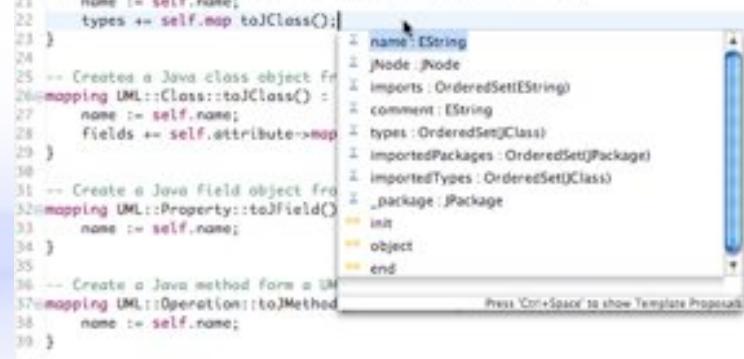
# Operational QVT Project (QVTO)

- The QVT OML is a sub-project of the M2M project
  - <http://www.eclipse.org/m2m/>
- Its goal is to provide an implementation of the MOF 2.0 Query/View/Transformation Specification
  - <http://www.omg.org/cgi-bin/doc?ptc/2007-07-07>
- Works out of the box for transforming EMF based models

# Developing QVTO Transformations

- Editor
  - Syntax highlighting
  - Code completion
- Model navigation
  - OCL based syntax for model navigation
  - Support for OCL collection operators
- Focus on transformation logic
  - Execution infrastructure left to the QVTO runtime

```
1modeltype UML "strict" uses "http://www.eclipse.org/uml2/2.1.0/UML";
2modeltype JAVA "strict" uses "http://www.eclipse.org/emf/2002/Java";
3
4transformation Room2Java(in source:UML, out target:JAVA);
5
6main() {
7    source.rootObjects()(UML::Model) -> map toJModel();
8}
9
10-- Create a sequence of java compilation units from
11-- a UML model by mapping each class in the UML model
12-- to a java class
13mapping UML::Model::toJModel() : Sequence(JAVA::JCompilationUnit) {
14    init {
15        result += self.packagedElement[UML::Class] -> map toJCompilationUnit();
16    }
17}
18
19-- Create a compilation unit from a UML class
20mapping UML::Class::toJCompilationUnit() : JCompilationUnit {
21    name := self.name;
22    types += self.map toJClass();
23}
24
25-- Creates a Java class object From
26mapping UML::Class::toJClass() :
27    name := self.name;
28    Fields += self.attribute->map
29}
30
31-- Create a Java Field object From
32mapping UML::Property::toJField() {
33    name := self.name;
34}
35
36-- Create a Java method form a UML
37mapping UML::Operation::toJMethod() {
38    name := self.name;
39}
```



The screenshot shows a code editor window displaying QVTO transformation code. A template proposal dropdown menu is open over the code, listing various UML elements like 'name: EString', 'INode: INode', etc. The menu also includes 'init', 'object', and 'end' options. A tooltip at the bottom right of the menu says 'Press 'Ctrl+Space' to show Template Proposals'.



# Testing QVTO Transformations

- Same approach as Java transformations
- Traditional Java test techniques
  - Hand crafted test framework
  - JUnit
- Using JUnit
  - Automatable
  - Repeatable



# Executing QVTO Transformations

- The QVTO project integrates with the workbench Run framework
- Select the transformation resource in the project explorer
- Choose Run As → Run Configurations... from the context menu
- Create a new configuration under Operational QVT Interpreter
  - The transformation parameters is populated from the transformation module that is specified
  - The option to generate a trace file for debugging
  - The option to save the configuration for sharing



# Integrating with the Workbench



# Integrating with RSA-RTE

Version 0.1, Toby McClean, Zeligsoft, 2008



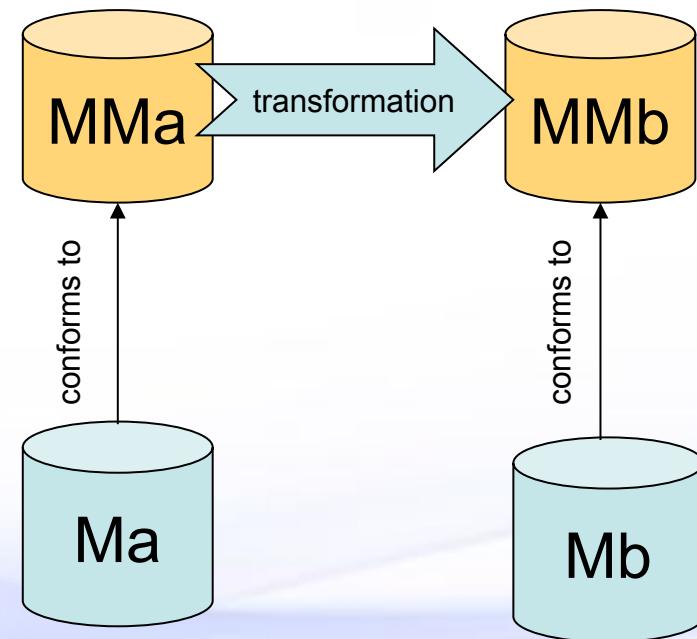
# QVTO - Transformation

- Referring to metamodels
  - modeltype keyword
  - Use namespace URI used to register metamodel  
modeltype UML uses ‘<http://www.eclipse.org/uml2/2.1.0/UML>’
- Transformation defines source and target metamodels
  - in keyword indicates source
  - out keyword indicates target

transformation Design2Implementation(in uml : UML, out UML);

# QVTO - Transformation

- Source model Ma that conforms to metamodel MMa
- Target model Mb that conforms to metamodel MMb
- Possible to have multiple sources and targets



**transformation mMa2MMb(in ma : Mma, out mb : MMb);**



# QVTO - Metamodels

- Define the parameter types for transformations
- Can be explicitly referenced by their namespace URI
  - UML - <http://www.eclipse.org/uml2/2.1.0/UML>
  - Ecore - <http://www.eclipse.org/emf/2002/Ecore>
- Use Metamodel Explorer view to find URI's
  - Window → Show View → Metamodel Explorer
- Syntax

```
modeltype <local name> uses '<ns uri>';  
modeltype UML uses 'http://www.eclipse.org/uml2/2.1.0/UML';
```



## QVTO - Entry Point

- The entry point is explicit and identified by a parameterless mapping with the name main
  - One entry point per transformation
  - `main() {<body>}`
- Invoked when the transformation is invoked
- Abstract transformations do not have an entry point

# QVTO - Mapping Rules

- A mapping rule
  - Applies to specific metaclass
  - Has a name that it is referred to by
  - May have additional in/out/inout parameters
  - Creates/modifies/returns one or more specific metaclasses
    - Maybe a collection
- Syntax

```
mapping (<context type>::)?<name>(<parameters>?)(:<result parameters>?) {<body>}
```
- Example

```
mapping UML::Class::capsuleToClass() : UML::Class { ... }
```



# QVTO - Mapping Parameters

- Mapping parameters allow additional data/elements to be passed into and out of the mapping
- Implicit parameters
  - self - context of mapping
  - result - target of mapping
- Direction
  - in - object passed in with read-only access
  - out - value set by mapping
  - inout - object passed in with readwrite access
- Syntax
  - <direction> <name> : <type>



# QVTO - Invoking Mapping

- A mapping is invoked on an object whose type complies with the context type of the mapping
- Special operation on object whose parameter is a mapping
  - `<object>.map <mapping with context type>()`
  - `capsule.map capsuleToClass();`
    - Assuming that capsule is UML::Class
- Values can be passed into the mapping
  - `capsule.map capsuleToClass(true);`



# QVTO - Constraining Mappings

- It is possible to restrict **when** a mapping will execute
  - when clause constrains the input parameters that are accepted for the mapping to execute
  - ... (:<result parameters>)?(**when** { <constraint> })?
  - **when** { self.isStereotypedBy('Capsule') }
- Two modes of invocation
  - Standard .map if the context doesn't satisfy the **when** clause then the mapping is not executed and control is returned to the caller
    - **when** clause acts like a guard
  - Strict .xmap if the context doesn't satisfy the **when** clause then an exception is thrown
    - **when** clause acts like a pre-condition



# QVTO - Post Conditions



# QVTO - Implementing Mappings

- Four sections to the body of a mapping
  - init
    - variable assignments
    - out parameter assignments
  - instantiation
    - implicitly instantiates out parameters that are null
  - population
    - updating output parameters
  - end
    - mapping invocations
    - logging, assertions, etc.



# QVTO - Expressions

- Object creation/construction



# QVTO - Object Construction

- To explicitly create an object of a specific type
- object keyword
  - object <identifier> : <type> { <update slots> }
- If the variable (referred to by identifier) is null
  - Creates the object
- If the variable is not null
  - Slots are updated
- Trace is created immediately upon object construction
- Can be part of an assignment statement



# QVTO – Object Construction

- No population
  - object jClass : JClass{}
- With population
  - object jClass : JClass{ name:= uClass.name; }
- As part of an assignment
  - features += object JOperation{ name:= 'log'; }



## QVTO - Constructors

- Special type of operation that creates instances of a specific type
  - Parameters used to populate the object
- Useful for simplifying transformation logic
- Invoked with the new keyword
- Syntax  
**constructor <type>::<name>(<parameters>){<body>}**

# QVTO - Constructors

## ■ Constructor example

- `constructor JavaMM::JClass::JClass(uName:String, attributes : Sequence(UML::Property)) {  
 name := uName;  
 fields += attNames.new(an) JField(an.name);  
}`

## ■ Using a constructor

- `types +=  
 packagedElement[UML::Class].new(uClass) JClass(uClass.name, uClass.attribute);`

# QVTO - Helpers

- A special type of operation that calculates a result from one or more source objects
  - Similar to an operation in Java
  - May have side effects on parameters
  - Requires explicit return
- Use for simplifying transformations
  - Encapsulate complex navigations
- Query helper
  - A helper that has no side effects on the parameters
  - Can be defined on primitive types to extend their capabilities

# QVTO - Helpers

## ■ Helper example

```
▪ helper umlPrimitiveToJava(UML::PrimitiveType uType) : String {  
    return if uType.name = 'String' then 'String' else  
        if uType.name = 'Boolean' then 'boolean' else  
            if uType.name = 'Integer' then 'int' else  
                uType.name  
            endif  
        endif  
    endif  
}
```

## ■ Query example

```
▪ query UML::Element::isTaggedWith(in tag: String) : Boolean {  
    return self.hasKeyword(tag);  
}
```

# QVTO - Intermediate Data

- Able to define classes and properties within a transformation
- Intermediate class
  - Local to the transformation it is defined in
  - Helps defined data to be stored during a transformation
  - Currently not supported
- Intermediate property
  - Local to the transformation it is defined in
  - Instance of metaclass or intermediate class
  - Use to extend metamodel
    - Can be attached to a specific type, appears as though it is a property of that type

# QVTO – Intermediate Data

- Intermediate class syntax
  - **intermediate class** <name> {<attributes>}
- Intermediate class example
  - **intermediate class** [TODO TM]
- Intermediate property syntax
  - **intermediate property** <name> : <type>;
- Intermediate property example
  - **intermediate property** UMLClass::allAttributes : Sequence(UML::Property);



# QVTO - Resolving Objects

- Transformations often perform multiple passes in order to resolve cross references
  - Referenced objects may not exist yet
- Facilities are provided to reduce the number of passes, by using the trace records that QVT creates
  - Resolve target from source and source from target
  - Resolve using a specific mapping rule
  - Specify number of objects to resolve
  - Defer resolution to end of the transformation
  - Filter the scope of objects to resolve



# QVTO – Resolving Objects

- Deferred resolution example
- Filtered resolution example
- Resolve single object example
- Resolve multiple objects example

# QVTO – Transformation reuse

- Composition
  - Explicit instantiation and invocation
  - transformation ROOM2JavaExt(in room : ROOM, out java : JAVA)  
access transformation ROOM2Java(in ROOM, out JAVA)

```
main() {  
    var base := new ROOM2Java(room, java);  
    base.transform();  
}
```

- Extension
  - Implicit instantiation
  - Ability to override a mapping in the extended transformation, which will be used in place of the mapping in the extended transformation
  - transformation ROOM2JavaExt(in room : ROOM, out java : JAVA)  
extends transformation ROOM2Java(in ROOM, out JAVA)



# QVTO – Mapping Reuse

- Inheritance
  - Inherited mapping is executed after the init section
- Merge
  - List of mappings executed in sequence after end section
- Parameters of inherited/merged mappings must match



# QVTO - Disjuncts

- An ordered list of mappings
  - First mapping in the list whose guard (type and when clause) is satisfied is executed
  - Null is returned if no mapping in the list is executed
- Example
  - [TODO]



# QVTO - OCL and MOF Type System

- QVT type system is an extension to the OCL and MOF type systems
  - Adds mutable list (List), mutable dictionary (Dict), ordered tuple (Tuple)
- [TODO – Move to advanced concepts slide]

# QVTO - Control

- **while loop**
  - execute a block until a condition is false
- **foreach**
  - iterate over a block
  - can add filter to the iterator
- **while and foreach support**
  - break
  - continue
- **if-then-else**
  - `if(<cond>){<block>}`
  - `elif(<cond>){<block>}`
  - `else{<block>}`



# QVTO - More than 1 Target Model

- When multiple target models are specified, need to be able to indicate which one a model is instantiated in
  - object <type>@<target model>{}
  - mapping <type>::<name> : <type>@<target model> {}

# QVTO - Libraries

- A QVTO library
  - contains definitions of specific types
  - contains queries, constructors, and mappings
- A library must explicitly included
  - By extending
  - By accessing
- Blackboxing
  - Defining a library in a language other than QVT



# QVTO - Configuration Properties

- Provided the ability to pass additional information into the transformation
- Accessed in the transformation logic as if they are variables
- Syntax
  - **configuration property <name> : <type>;**
- Example
  - **configuration property useGenerics : Boolean;**

# QVTO - Logging

- Transformations can log messages to the execution environment
- `log(<message>, <data>, <log level>);`
  - Message - the message to the users
  - Data - an optional parameter that is the model element to be associated with the message
  - Log level - an integer indicating logging level that can be used to filter message significance



# QVTO - Working with Profiles

- QVTO provides no special operators for working with UML2 profiles
- Transformations and mappings use the API defined in the UML2 metamodel
- A library can be built to simplify the UML2 stereotype API
- Example
  - ```
query UML::Element::isStereotypedBy(in qualifiedName : String) : Boolean {
    return self.getAppliedStereotype(qualifiedName) <> null;
}
```

# QVTO - Extensions

- Blackbox libraries are defined through the `org.eclipse.m2m.qvt.oml.ocl.libraries` extension point
- Must have a static class `Metainfo`
  - specifies the parameters of the transformation
- Enables QVTO to leverage the power of a 3GL like Java



# QVTO - Programmatically Invoking

- [TODO Decide if this is beyond the scope or can be moved to exercises]



# Summary

- In this module we explored



# Eclipse Transformation Technologies

M2T with xPand



# xPand Overview

- The M2T language from the openArchitectureWare toolkit
- xPand has been adapted and used by the GMF project for its generators
- Graduated to become a sub-project of the M2T project with MDT
  - Integrating some of changes made by GMF
- It is a proprietary declarative template based language



## xPand Highlights

- Supports template polymorphism
- Extensible with the xTend language
- Support for aspect oriented techniques
- Editor with syntax highlighting and code completion support
  - Metamodel aware
  - Extension aware
- Debugger



# Developing xPand Transformations

- Editor
  - Syntax highlighting
  - Code completion
- Model navigation
  - OCL based syntax for model navigation
  - Support for subset of OCL collection operators
- Focus on transformation logic
  - Execution infrastructure left to the xPand runtime



# Testing xPand Transformations

Version 0.1, Toby McClean, Zeligsoft, 2008



# Executing an xPand Transformation

- Using an oAW workflow
  - More about workflows later
  - Can use integration with the Run as... or Debug as... oAW workflow
- Write a Java application
  - Call the transformation explicitly in code
  - Use the Run as... or Debug as... Java Application

# Integrating with Workbench

- Contribute an action to the editor's menu
  - Implement the action handler
    - Call the workflow passing in the selected element as the source
- Alternative is append to an existing action
  - e.g. the ROOM to Java transformation
- Test
  - In runtime workbench
- Deploy
  - Publish the plug-in



# Integrating with RSA-RTE

- Contribute an action
  - to the Project Explorer context menu
  - to the Diagram Editor context menu
- Implement action handler
  - Class that implements IEditorActionDelegate
- Test
  - In runtime workbench
- Deploy
  - Publish the plug-in



# xPand Transformation Example

Version 0.1, Toby McClean, Zeligsoft, 2008

# xPand - Metamodels

- Define the types used in transformation
  - Use fully qualified names
  - Use unqualified names for imported metamodels
- Referenced through
  - namespace
- Syntax
  - «IMPORT <namespace of metamodel>»
- Example
  - «DEFINE write FOR java::JClass»
  - «IMPORT java»
  - ...
  - «DEFINE write FOR JClass»

# xPand - Extensions

- xPand has a supporting language xTend for specifying extensions
  - Additional features for metamodel types
  - Additional helper functionality
- Extensions with xTend
  - xTend language which can define blackbox functions that are implemented in Java
- Found on the classpath of the xPand template
  - Imported by «EXTENSION com::zeligsoft::exercises::room::xpand::RoomUtils»
- Appear as though they are part of the meta type

# xPand - Templates

- An xPand template consists of
  - Referenced metamodels
  - Imported extensions
  - Set of DEFINE blocks
- DEFINE block
  - name
  - metamodel class for which template is defined
  - comma separated parameter list
- Syntax
  - «DEFINE templateName(formalParameterList) FOR MetaClass»  
    a sequence of statements  
«ENDDEFINE»



# xPand - Output

- The output control structure in xPand is FILE which defines a target file to write the contents of the block to
- Outlets can be defined in the workflow and referenced
  - <outlet path='main/src-gen' /> -- default
  - <outlet name='TO\_SRC' path='main/src' overwrite='false' />
  - «FILE 'test/note.txt' TO\_SRC»  
# this goes to the TO\_SRC outlet  
«ENDFILE»



# xPand - Invoking a Template

- The xPand language uses EXPAND to invoke a template
  - «EXPAND definitionName [(parameterList)]  
[FOR expression | FOREACH expression [SEPARATOR expression] ]»
- Omitting FOR or FOREACH invokes with FOR this
- FOREACH invokes for each element in the collection
- SEPEARATOR specifies an optional delimiter output between each invocation in FOREACH



# xPand – Invoking a Template

- Invocation finds a template than matches the name specified and picks the most specific type match
  - Polymorphic behaviour

# xPand - Control

- **LET block**
  - the value of the expression is bound to the specified variable
  - only available inside the block
- **IF, ELSEIF, ELSE block**
  - traditional if construct from programming languages
- **ERROR**
  - aborts evaluation with the specified message



# xPand - Advanced Features

- **PROTECT**
  - used to mark sections in the generated code that shall not be overridden again by the subsequent generation



# Summary

- In this module we explored



# xTend Overview

- A language to
  - Define libraries of independent operations
  - Define non-invasive metamodel extensions
- Operations and extensions can be defined
  - Using xTend expressions
  - Using Java (blackboxing)
- Can be called
  - Directly from a workflow
  - From within an xPand transformation



# JET Overview

- Original Eclipse M2T technology
  - Language for specifying templates to output text based artifacts
- Works with XML content and EMF based models
  - Including UML2 models
- A declarative language
  - XML based syntax
  - Extensive use of XPath for model navigation
- JET templates are automatically compiled to Java
- Used by
  - Eclipse EMF code generation
  - RSA-RTE intermediate language model to text transformation



# Developing a JET Transformation

- JET transformations can be created by
  - Adding a JET transformation to an existing project
  - Creating a JET transformation project
- Editor
  - Syntax highlighting
- Model navigation
  - XPath
- Focus on transformation logic
  - Execution infrastructure left to the compiled JET code



# Executing a JET Transformation

- The JET project integrates with the workbench Run framework
- Select the transformation resource in the project explorer
- Choose Run As → Run Configurations... from the context menu
- Create a new configuration under JET transformation
  - The input model is specified
  - The option to save the configuration for sharing

# Integrating with Workbench

- Contribute an action to the editor's menu
  - Implement the action handler
    - Call the workflow passing in the selected element as the source
- Alternative is append to an existing action
  - e.g. the ROOM to Java transformation
- Test
  - In runtime workbench
- Deploy
  - Publish the plug-in



# Integrating with RSA-RTE

- Contribute an action
  - to the Project Explorer context menu
  - to the Diagram Editor context menu
- Implement action handler
  - Class that implements IEditorActionDelegate
- Test
  - In runtime workbench
- Deploy
  - Publish the plug-in



# JET Concepts Overview

- Comments
  - <%-- ... --%>
- Directives
  - Provide guidance to JET
  - <%@ ... %>
- Declarations
  - Declare Java methods or fields
  - <%! ... %>
- Expressions
  - Valid Java expression, no semicolon
  - <%= ... %>
- Scriptlets
  - Valid Java statements or blocks (complete or partial)
  - <% ... %>



## JET - Metamodels

- By default JET is setup to work with XML files
- To work with EMF models
  - In the transform extension specify model loader as `org.eclipse.jet.emf.modelLoader`
- To work with a specific metamodel add its Java package to the imports list of the template



# JET – Loading Models

- There are two tags in JET used to load content to be used in the transformation
- c:load
  - Loads the referenced model into a specified variable
  - Can be referenced through the variable after that
- c:loadContent
  - Load the content of the tag into the specified variable as XML



## JET - Extensions

- JET is extremely extensible and since it is much like JSP you are able to insert Java almost anywhere in a template
- The other means of extension are
  - Custom model loaders
  - New tag libraries
  - New XPath functions
  - Custom model inspectors

# JET - Templates

- A template in JET is defined in a file
  - There is a 1 to 1 mapping between file and template
- Directives configure the template
  - @jet
    - affects the code created by the JET compiler
    - package
    - class
    - imports
    - startTag
    - endTag
  - @taglib
    - imports a tag library for use in the template and assigns it a namespace prefix

# JET - Output

- Out of the box JET provides several tags that help with output produced by the transformation
- f:ident
  - Indent the contents the specified number of times
- f:lc, f:uc
  - Convert the contents to lowercase/uppercase
- f:replaceAll
  - Replace all instances of a value within the contents to a new value
- f>xpath
  - Evaluate an XPath expression and writes it result



# JET - Invoking a Template

- To invoke another template from the current template use the c:invokeTransform
  - passes the current transformation's source and context variables
- Syntax
- Example



## JET - Control

- The control structures available in JET include the following tags
  - c:choose
    - A group of mutually exclusive choices
  - c:if
    - Only process the contents if a test condition is satisfied
  - c:iterate
    - Process the contents for each element specified by an XPath expression
    - If the XPath expression evaluates to a number that it iterates that number of times



## JET - Reuse

- There are two forms of reuse in JET
  - c:include tag which processes the referenced template and includes its output
    - Variables from the including template are passed to the included template (can specify which ones)
  - Overriding a transformation
    - In the transform extension point declare that the transformation overrides templates in the overridden transformation

# JET - Expressions

- JET has several tags for working model elements
- c:get
  - Evaluate an XPath expression and write the result
- c:set
  - Select an object with a XPath expression and set an attribute on it to the specified value
  - Can be used to dynamically add to an object at runtime
- c:setVariable
  - Create a variable and sets its value by specifying an XPath expression



## JET - Working with Profiles

- Since JET is a generic solution there is no special support for UML Profiles
- Use the UML API to access stereotype information
- Create templates that encapsulate the handling of specific stereotypes

# JET - Logging

- JET has several tags to support logging in the transformation
- c:log
  - write a message to the transformation log
  - optional severity attribute
- c:dump
  - dump the contents of the node passed
- c:marker
  - create an Eclipse task marker to the text in the tag
  - optional description for the marker



# JET - Advanced

Version 0.1, Toby McClean, Zeligsoft, 2008

# oAW Workflow

- An XML based language for describing the sequence of steps in a transformation
  - For example
    - load UML model,
    - transform to Java mode,
    - manipulate Java model,
    - write Java model to source, and
    - format generated Java code
- In the process of becoming an Eclipse project called Model Workflow Engine (MWE)



# oAW Workflow Project?

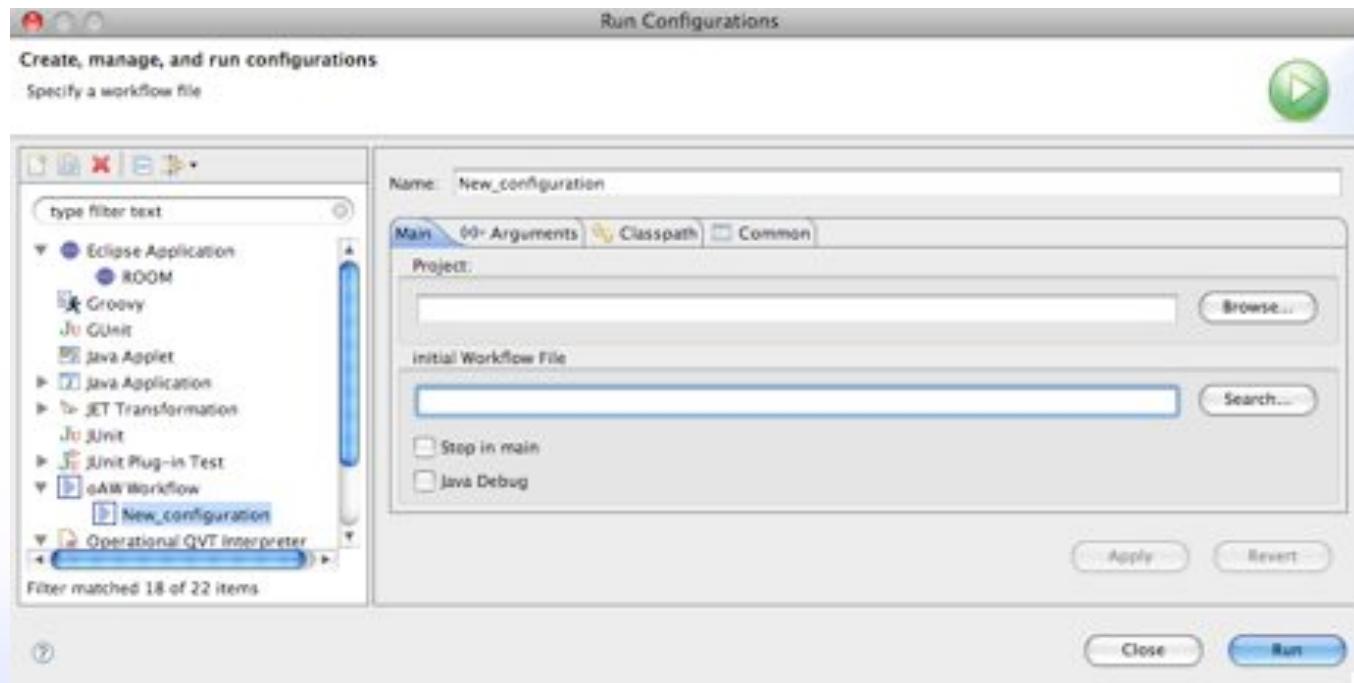
- Out of the box the oAW Workflow Project consists of
  - A workflow execution engine
  - Workflow components for reading and writing EMF models
  - API for integration with oAW Workflow
  - Workbench integration
    - Editor, Run as..., Debug as..., and ANT

# More oAW Workflow Project?

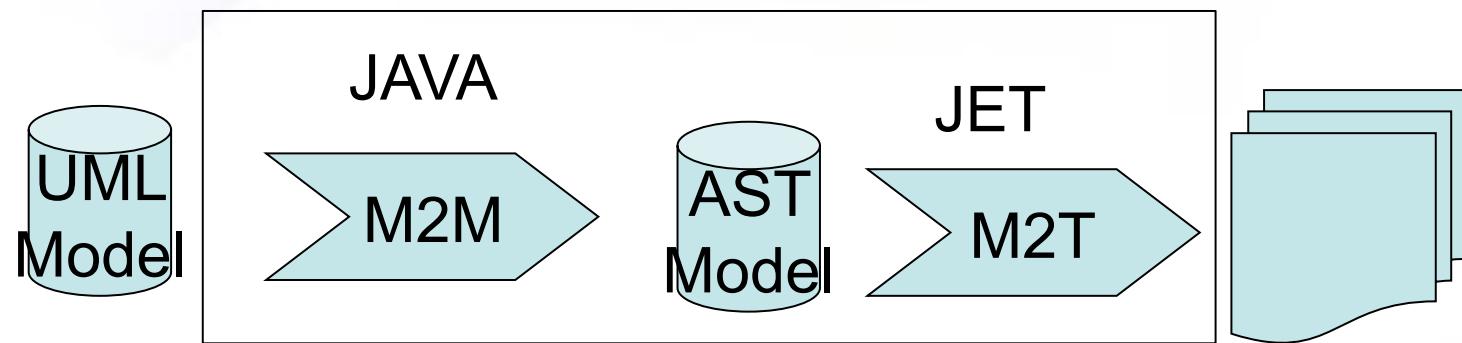
- API allows for custom workflow components
  - e.g. QVT transformation execution
- Configuration properties
  - Properties passed into the workflow
  - <property name='targetDir' value='src-gen'/>
- Slots or variables
  - Simple syntax, variables are referred to by name
  - No declaration
  - <component id="generator" class="oaw.xtend.XtendComponent">  
    ...  
    <outputSlot value="javaModel" />  
  </component>

# Executing an oAW Workflow

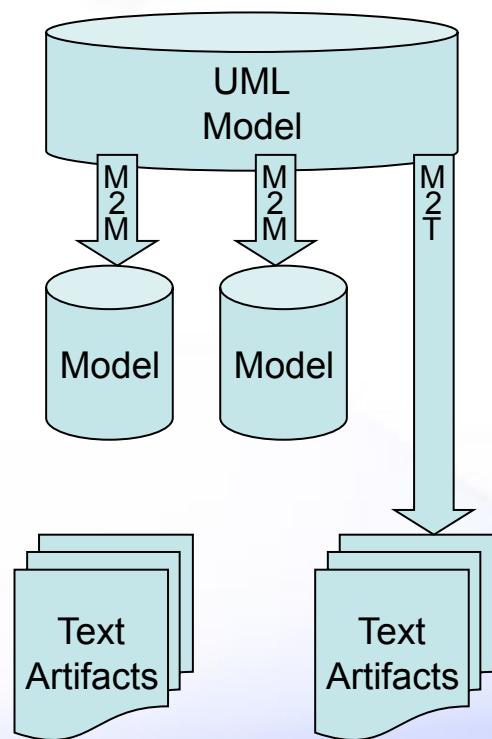
- The oAW Workflow Engine is integrated with Run as...



# RSA RTE Transformation Workflow



# Transforming UML Models in Eclipse





# Summary

- In this module we explored
  - Plug-ins
  - MANIFEST
  - Plug-in descriptors
  - Testing plug-ins
  - Exporting plug-ins