



Eclipse Transformation Training



Introductions



Introduction - Instructors

- Workshop leader
 - Toby McClean
 - Principal Technology Specialist
 - Modeling, Domain Specific Modeling and Languages
 - Model transformations and code generation
 - Component based development and standards
- Facilitators
 - Mark Hermeling (week 1)
 - Director, Product Technology
 - Tim McGuire (week 2)
 - Senior Software Engineering Specialist

zeligsoft



Introductions - Participants

- Briefly, a little about you
 - Name
 - Department
 - What is your interest in Eclipse and the Rational Modeling Platform
- Experience
 - With Eclipse
 - With RSA, RSD or RSM
- How do you expect to apply what you learn this week

zeligsoft



Workshop Goals

- Workshop contents
 - Provide a high-level overview of Eclipse technologies
 - Related to modeling, domain specialization and transformation
 - Give you the chance to gain hands-on experience
 - With programmatically creating and transforming models
 - The second bullet requires significant detail
- What you will be able to do after the workshop
 - You will understand the possibilities of Eclipse and RSA-RTE
 - And have a good idea how to extend this environment for your day-to-day activities

zeligsoft

Toby McClean, Zeligsoft, 2008



Workshop Agenda

- Monday
 - Architecture overview
 - Eclipse modeling overview
- Tuesday
 - EMF in depth
 - UML in depth
- Wednesday
 - Transformation 1
- Thursday
 - Transformation 2
 - Validation

zeligsoft



Schedule

- 9:00-10:30
- 10:45-12:00
- 13:00-15:00
- 15:15-17:00

zeligsoft



Workshop Make-Up

- Half of this workshop will be slide material
- Half of it will be exercises and discussions
- Typical flow
 - Presentation of module
 - Discussion on how to apply to Ericsson's environment
 - Exercise introduction
 - Exercise
 - Wrap-up of topic
- This is a custom-built course for Ericsson
 - We encourage discussion and exploration

zeligsoft



Workshop Materials

- Workbook with slides
- Workbook with exercises
- VMWare image with Eclipse pre-installed
- Solution files

zeligsoft

Toby McClean, Zeligsoft, 2008



Administrative Topics

- Topics are presented in sequence
 - One module builds on the other
 - Exercises are important, but attendance is optional
 - That is, you will be able to continue if you miss an exercise
 - If you need to take of other business, this is a good time
- Schedule is approximate
 - We may need to shift timing and topics around
 - Depending on amount of discussion and your feedback

zeligsoft



The Eclipse Project

An introduction to the Eclipse Project



Overview

In this module we will explore the Eclipse Project including the different aspects of the Eclipse Workbench.

We will also explore how to develop and deploy plug-ins to extend the Eclipse Workbench.

Some content taken from
Developing Plug-ins for Eclipse (<http://www.eclipse.org/resources/>)
by Dwight Deugo and Nesa Matic

Toby McClean, Zeligsoft, 2008



Goals

- Following the completion of this module and its exercises you will
 - Have an understanding of the Eclipse Workbench
 - Know some of the terminology used within the Eclipse Workbench
 - Have an understanding of the Eclipse architecture
 - Have an understanding of how Eclipse is extended
 - Be able to develop and deploy a basic plug-in



Agenda

- What is Eclipse?
 - The Eclipse Workbench
 - The Eclipse Architecture
 - Extending the Workbench



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

- On topic
 - Business Intelligence and Reporting Tools (BIRT)
 - Device Software Development Platform
 - Eclipse Project
 - Eclipse Modeling Project
- Off topic
 - Data Tools Platform
 - Eclipse RT
 - SOA Tools
 - Eclipse Technology Project
 - Tools Project
 - Test and Performance Tools Platform Project
 - Eclipse Web Tools Platform Project

<http://www.eclipse.org/projects/listofprojects.php>



Brief History of Eclipse

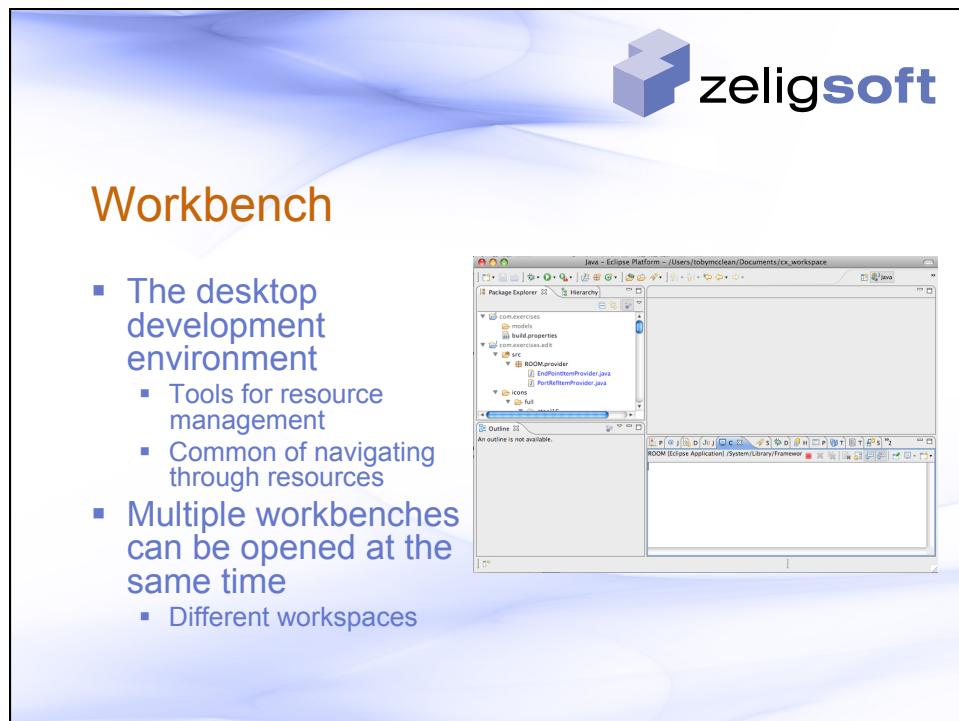
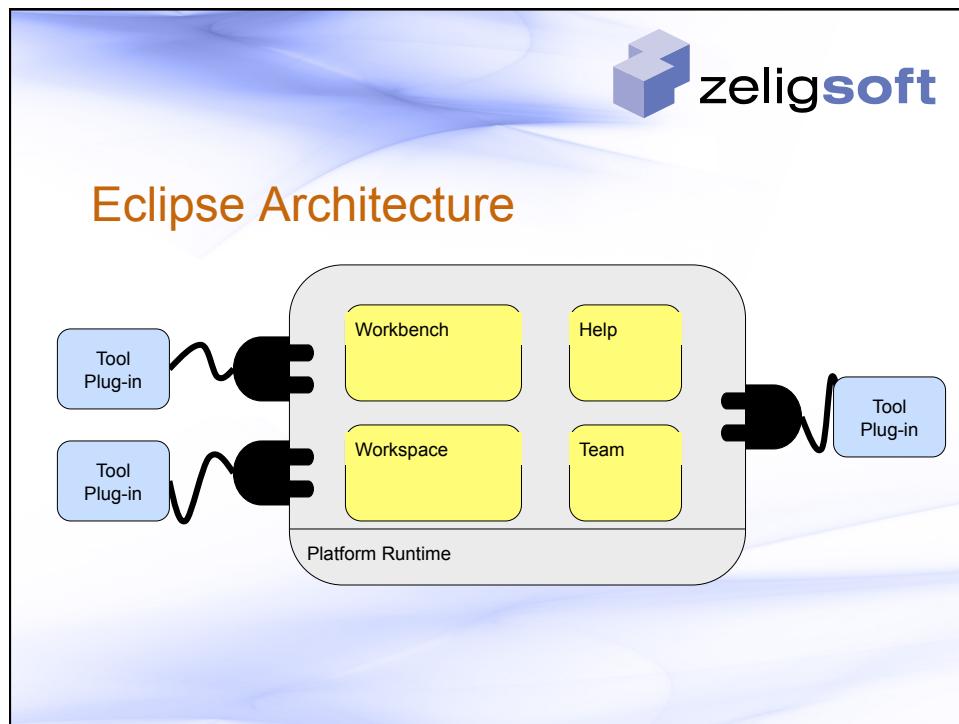
- 1994
 - VisualAge for Smalltalk
- 1996
 - VisualAge for Java
- 1996-2001
 - VisualAge Micro Edition
- 2001
 - Eclipse Project



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
- Facilitate the integration of tools
- Cross-platform support

Toby McClean, Zeligsoft, 2008



Toby McClean, Zeligsoft, 2008

zeligsoft

Workspace

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

zeligsoft

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



Agenda

- What is Eclipse?
- The Eclipse Workbench
- The Eclipse Architecture
- Extending the Workbench



The Eclipse Project

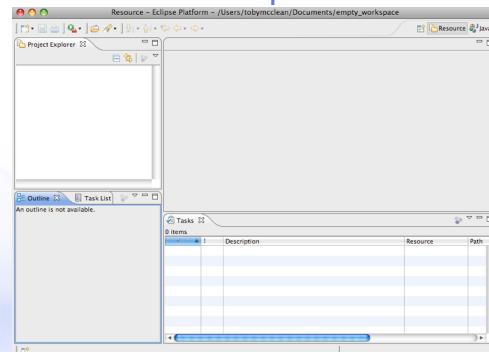
The Eclipse Workbench

Toby McClean, Zeligsoft, 2008



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 window for 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



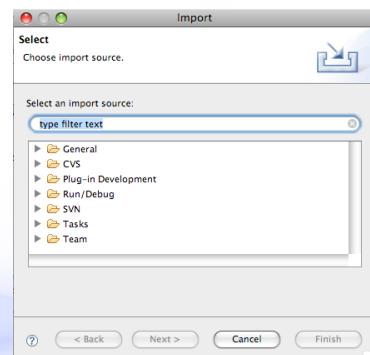
Resources in a Workbench

- 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 four different levels of resources:
 - Workspace root
 - 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



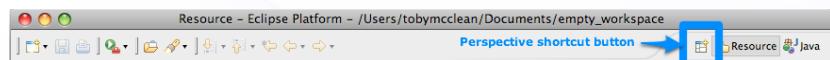
Perspectives

- 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



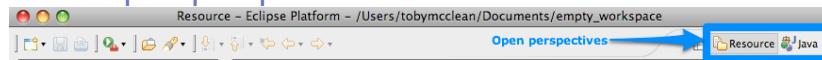
Customizing Perspectives

- 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



Agenda

- What is Eclipse?
- The Eclipse Workbench
- **The Eclipse Architecture**
- Extending the Workbench



The Eclipse Project

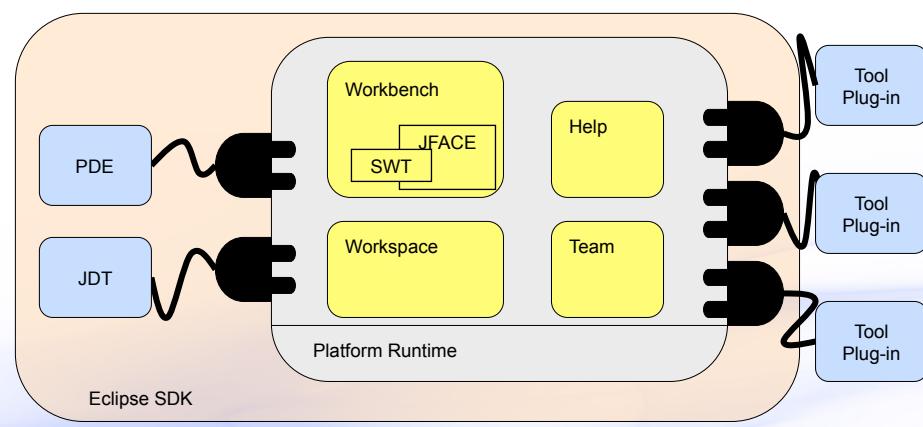
Eclipse Architecture

Toby McClean, Zeligsoft, 2008

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
 - Integrated 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
 - Defines plug-in extensions, and
 - Declares plug-in 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

zeligsoft

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

zeligsoft

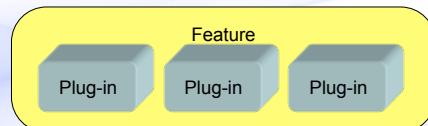
Installing Plug-ins

- 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



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



Agenda

- What is Eclipse?
- The Eclipse Workbench
- The Eclipse Architecture
- Extending the Workbench



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

Toby McClean, Zeligsoft, 2008



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”



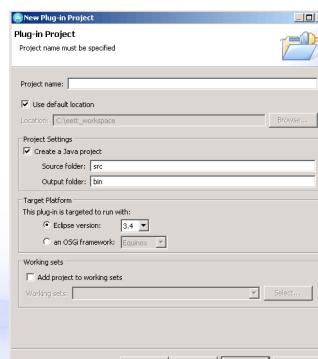
How to Extend the Workbench?

- Steps for adding a plug-in
 - Define Plug-in project
 - Plug-in nature
 - Java nature (optional)
 - Write the plug-in code
 - 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

zeligsoft

Creating Plug-in Project

- Plug-in project are created with the New Project wizard
 - File → New → Project...
- Project will contain
 - source code for the plug-in
 - manifest and property files

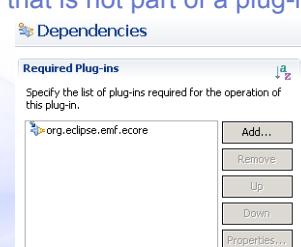


Implementation details

zeligsoft

Updating Project's Dependencies

- To make classes required for the plug-in visible for the project, update its dependencies
 - You add a dependency on another plug-in
 - You can add a jar that is not part of a plug-in to the build path



Implementation details

Toby McClean, Zeligsoft, 2008



Creating a Class

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

[Implementation details](#)



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

[Implementation details](#)



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

Implementation details



Defining Manifest

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



Implementation details

Toby McClean, Zeligsoft, 2008

zeligsoft

Defining plugin.xml

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

Implementation details

zeligsoft

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

Implementation details



Getting Ready to Export Plug-in

- Create a build.properties file in your project

```
source.. = src/  
output.. = bin/  
bin.includes = META-INF, \  
.,\  
plugin.xml, \  
plugin.properties
```

- Build properties editor available

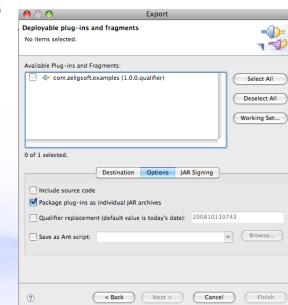
- Open with → Build Properties Editor

Implementation details



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



Implementation details

Toby McClean, Zeligsoft, 2008



Plug-in Registry

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

Implementation details



Summary

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



Questions



Toby McClean, Zeligsoft, 2008



Eclipse and Rational Modeling Tools



Overview

- In this module we explore the eclipse modeling tools in more detail
- Look at the different projects and how they relate



Goals

- After this module you will
 - Have an understanding of Eclipse modeling projects
 - How they extend each-other
 - How they can be transformed into models and text
 - You will also have a roadmap for the rest of the training



Agenda

- Modeling in Eclipse
 - EMF, GMF, TMF, TCS
- Tooling in Eclipse
 - MDT, EMFT,
- Transformations
 - M2M, M2T

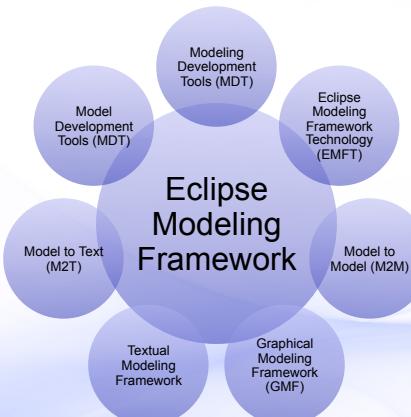


Eclipse Modeling Project

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



Ecosystem of the Modeling Project



Toby McClean, Zeligsoft, 2008



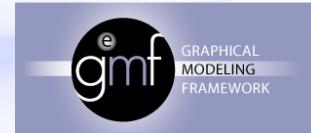
Eclipse Modeling Framework (EMF)

- 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



Toby McClean, Zeligsoft, 2008



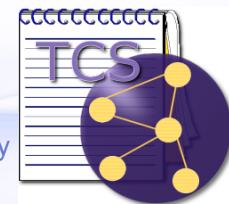
Textual Modeling Framework (TMF)

- 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
- Non-standards based language for annotating an abstract syntax model with textual syntax
- Able to generate
 - Grammar
 - Editor
 - Model to text transformation with traceability





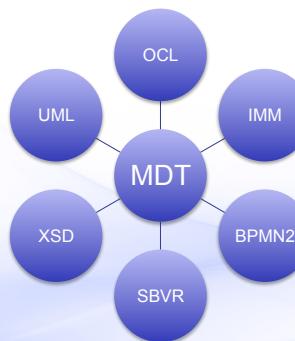
TMF - xtext

- 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 stand models



Toby McClean, Zeligsoft, 2008



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



Toby McClean, Zeligsoft, 2008



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 JSP 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 non-standards based syntax
- Editor for developing templates/transformations
- Execution engine for interpreting templates
- An extended version is used by GMF to generate code



Summary

- The projects are nicely layered
- Good effort to keep things separated
- Ability for users to pick and choose what they want
- Lots of content to go through this week



Questions



Toby McClean, Zeligsoft, 2008



Exploring EMF

What is the Eclipse Modeling Framework?



Overview

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



Agenda

- **EMF models**
 - Meta-model
 - Creating models
 - Testing models
- Generating code from EMF models
 - Model manipulation
 - Model Editors
- EMF architecture and run-time
 - Transactions
 - Queries



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

Toby McClean, Zeligsoft, 2008

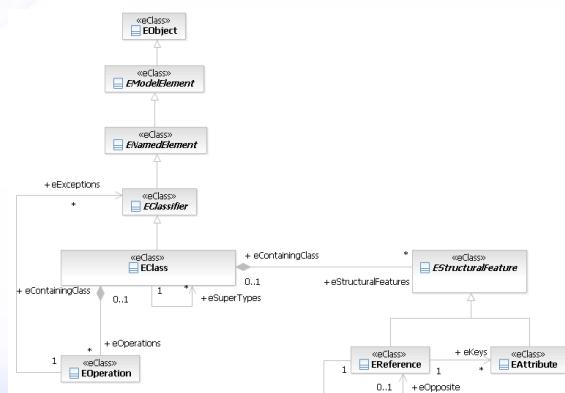


What is an EMF Model?

- 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



Toby McClean, Zeligsoft, 2008



Ecore Metamodel Concrete Types

- **EPackage**
 - A named grouping of concepts, the domain
- **EClass**
 - A concept or class of object in a domain
- **EAttribute**
 - An attribute used to describe or define a concept
- **EReference**
 - Relationship with another concept
- **EOperation**
 - Behavior of a concept
- **EEnum**
 - A type whose possibilities are defined by a list of literals
- **EDataType**
 - Defines the types of attributes



EPackage

- **name**
 - Friendly label that need not be unique
- **nsURI**
 - Used to uniquely identify the package,
 - Used by serialization as the XML namespace
- **nsPrefix**
 - The namespace prefix that corresponds to the XML namespace in the serialized instance models
- **eClassifiers**
 - Contains a set of EClass and EDataTypes
- **eSubPackages**
 - May contain a set of nested EPackages



EClass

Property	Value	
Abstract	<input checked="" type="checkbox"/> false	False means that instances be created
Default Value	<input type="checkbox"/>	
ESuper Types	<input type="checkbox"/> NamedElement, PackageableElement	
Instance Type Name	<input type="checkbox"/>	The Java type that is modeled by this
Interface	<input checked="" type="checkbox"/> false	If true no implementation is generated
Name	<input type="checkbox"/> Actor	

EAttribute

Property	Value	
Changeable	<input checked="" type="checkbox"/> true	Can the value be changed
Default Value Literal	<input type="checkbox"/> false	
Derived	<input checked="" type="checkbox"/> false	Is the value derived from other ref or attrs
EAttribute Type	<input type="checkbox"/> EBoolean [boolean]	
EType	<input type="checkbox"/> EBoolean [boolean]	
ID	<input checked="" type="checkbox"/> false	
Lower Bound	<input type="checkbox"/> 0	
Name	<input type="checkbox"/> conjugated	
Ordered	<input checked="" type="checkbox"/> true	Does the order of the elements matter
Transient	<input checked="" type="checkbox"/> false	False means it's value is serialized
Unique	<input checked="" type="checkbox"/> true	
Unsettable	<input checked="" type="checkbox"/> false	Does the value space include the unset state
Upper Bound	<input type="checkbox"/> 1	
Volatile	<input checked="" type="checkbox"/> false	False means a field is generated for the attrib.



EReference

Property	Value
Changeable	<input checked="" type="checkbox"/> true Can the value be changed
Container	<input checked="" type="checkbox"/> false
Containment	<input checked="" type="checkbox"/> false Is it a composite relationship
Default Value Literal	<input type="checkbox"/>
Derived	<input checked="" type="checkbox"/> false Is the value derived from other references or attribute
EKeys	
EOpposite	
EType	<input type="checkbox"/> Protocol -> NamedElement, PackageableElement
Lower Bound	<input type="checkbox"/> 0
Name	<input type="checkbox"/> protocol
Ordered	<input checked="" type="checkbox"/> true
Resolve Proxies	<input checked="" type="checkbox"/> true Resolve the proxies automatically?
Transient	<input checked="" type="checkbox"/> false False means it's value is serialized
Unique	<input checked="" type="checkbox"/> true
Unsettable	<input checked="" type="checkbox"/> false Does the value space include the unset state
Upper Bound	<input type="checkbox"/> 1
Volatile	<input checked="" type="checkbox"/> false False means a field generated for the reference

EOperation

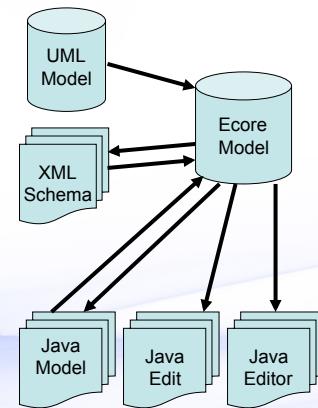
Property	Value
EExceptions	exceptions thrown by the operation
EType	return type
Lower Bound	<input type="checkbox"/> 0 lower bound on the return type
Name	<input type="checkbox"/> getAllPorts
Ordered	<input checked="" type="checkbox"/> true
Unique	<input checked="" type="checkbox"/> true
Upper Bound	<input type="checkbox"/> -1 upper bound on the return type

Toby McClean, Zeligsoft, 2008



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
- Supports both
 - EMFizing a legacy data model
 - Greenfields development



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
 */
public interface Port extends NamedElement {
    /**
     * @model
     */
    Protocol getProtocol();

    /**
     * @model
     */
    boolean isConjugated();
}
```



More EMF from Java

- Create or use existing Java interfaces for data model or 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



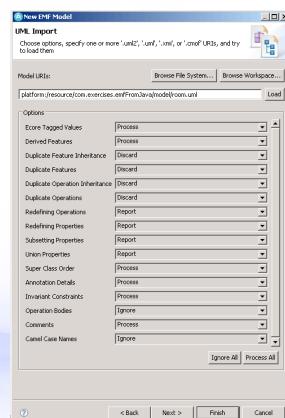
EMF from UML Class Model

- 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...



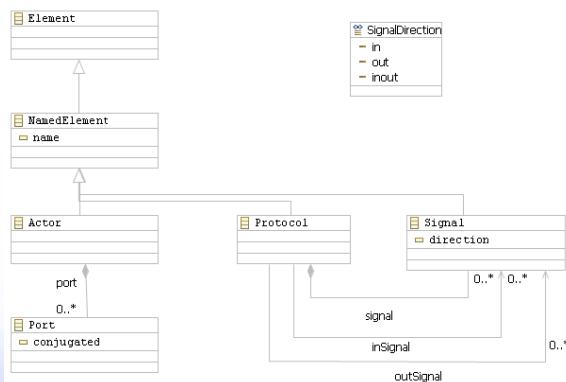
EMF with Ecore Diagram Editor

- 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

Toby McClean, Zeligsoft, 2008



More EMF with Ecore Diagram Editor



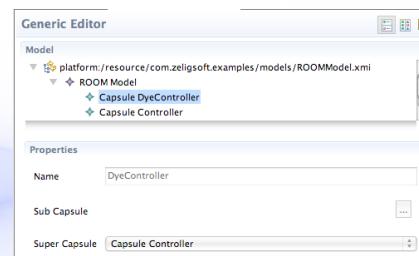
How do I Test my Model?

- Manually review the model and ensure that it looks like it will work!
 - Error prone
- Build the generation model now generate the default model editor
 - Time consuming when frequently changing
- Use the dynamic instance creation capabilities
 - Part of the Sample Ecore Model Editor
- Use the generated test suite from EMF
 - Generates a JUnit test suite, to test the user supplied code in operations and derived features



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



Do this with Code?

- Register the ecore model as a dynamic package
- Use the reflective API of EMF
 - Can create instances and persist them



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(nsURI)

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



Using the 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



Dynamic example

```
private static final String MODELS_FOLDER
    = "platform:/resource/com.eett.exercises.emf.pluginlets/models/";
...
// Retrieved the EPackage for the Room metamodel that we registered
// It is retrieved using the string we specified as the URI when we
// registered it in the dynamic_package extension point
EPackage room
    = EPackage.Registry.INSTANCE
        .getEPackage("http://www.eett.com/room/2008");

// Create a ResourceSet so that we can create Resources
// to store the model elements we create in
ResourceSet resourceSet = new ResourceSetImpl();

// Create the Resources to store the objects that we create in
Resource controllerResource
    = resourceSet.createResource(URI.createURI(
        MODELS_FOLDER + "dyeingController.xmi"));
Resource abstractControllerResource
    = resourceSet.createResource(URI.createURI(
        MODELS_FOLDER + "abstractDyeingController.xmi"));

Implementation details
```



Dynamic Example (2)

```
// In order to be able to create an Actor object we need
// have its EClass which we can get from the EPackage
EClass actorEClass = (EClass) room.getEClassifier("Actor");

// Similarly, to set the features of an Actor we
// need the EStructuralFeature objects for them which we
// can get from the EClass
EStructuralFeature actorNameAttribute
    = actorEClass.getEStructuralFeature("name");
EStructuralFeature actorSuperclass
    = actorEClass.getEStructuralFeature("actorSuperclass");

// Create the AbstractDyeingController Actor and add it
// to the Resource we created for it
EObject abstractController =
    EcoreUtil.create(actorEClass);
abstractController
    .eSet(actorNameAttribute, "AbstractDyeingController");
abstractControllerResource.getContents().add(abstractController);

Implementation details
```

Toby McClean, Zeligsoft, 2008



Dynamic Example (3)

```
// Create the DyeingController Actor and add it
// to the Resource we created for it
EObject controller = EcoreUtil.create(actorEClass);
controller
    .eSet(actorNameAttribute, "DyeingController");
controllerResource.getContents().add(controller);

// Set the actorSuperclass feature of the dyeingController
// to be the abstractController
controller.eSet(actorSuperclass, abstractController);
```

Implementation details



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



Agenda

- EMF models
 - Meta-model
 - Creating models
 - Testing models
- Generating code from EMF models
 - Model manipulation
 - Model Editors
- EMF architecture and run-time
- Transactions
- Queries



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
 - Volatile attributes
 - Operations

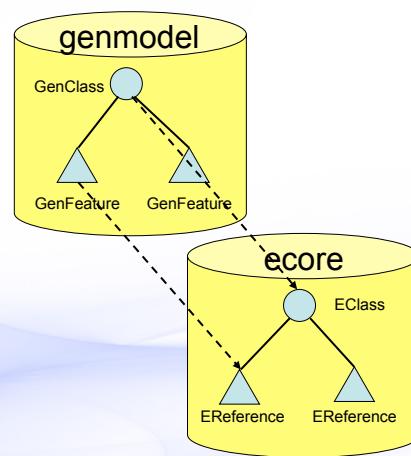
Toby McClean, Zeligsoft, 2008

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
- *NOTE THAT BOTH ANT AND COMMAND LINE STILL REQUIRE ECLIPSE*

The genmodel

- Wraps the Ecore model and automatically kept in sync
 - Only true for Ecore
 - Others like Java require explicit reload
- 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 is generated
 - Model
 - Edit
 - Editor
 - Tests

Property	Value
Bundle Manifest	<input checked="" type="checkbox"/> true
Compliance Level	<input checked="" type="checkbox"/> 5.0
Copyright Fields	<input checked="" type="checkbox"/>
Copyright Text	<input checked="" type="checkbox"/>
Language	<input checked="" type="checkbox"/>
Model Name	<input checked="" type="checkbox"/> Room
Model Plug-in ID	<input checked="" type="checkbox"/> com.zeligsoft.examples
Non-NLS Markers	<input checked="" type="checkbox"/> false
Runtime Compatibility	<input checked="" type="checkbox"/> false
Runtime Jar	<input checked="" type="checkbox"/> false
Runtime Version	<input checked="" type="checkbox"/> 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



Model Code Detail

Unit	Description	File name	Subpkg.	Opt.
Model	Plug-in Class			Y
	OSGi Manifest	META-INF /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

Unit	Description	File name	Subpkg.	Opt.
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

Toby McClean, Zeligsoft, 2008



More Model Edit Code

Unit	Description	File Name
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
 - Choice between workbench integration or a rich client
- 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

Unit	Description	File Name
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>Advisor.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
 - This is the best practice

Toby McClean, Zeligsoft, 2008



Summary

- In this module we explored
 - The EMF code generation
 - What is generated?
 - Model
 - Edit
 - Editor
 - How do we generate it?

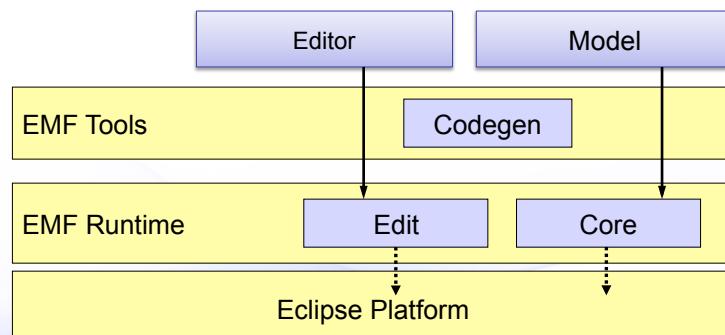


Agenda

- EMF models
 - Meta-model
 - Creating models
 - Testing models
- Generating code from EMF models
 - Model manipulation
 - Model Editors
- **EMF architecture and run-time**
- Transactions
- Queries



EMF Architecture



EMF Runtime - Core

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

Toby McClean, Zeligsoft, 2008



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
- Resources are identified by their their URI



Notification

- Every EObject is a notifier
 - Whenever it changes it notifies interested parties
- In EMF terminology observers are adapters
 - A way to extend or change the behavior
 - No subclassing is required
- AdapterFactory provides the means to add an adapters
 - `adapterFactory.adapt(object, ITreelItemContentProvider.Class)`
- Notification is automatically generated

Toby McClean, Zeligsoft, 2008



Notification Example

```
/**  
 * <!-- begin-user-doc -->  
 * <!-- end-user-doc -->  
 * @generated  
 */  
public void setActorSuperclass(Actor newActorSuperclass) {  
    Actor oldActorSuperclass = actorSuperclass;  
    actorSuperclass = newActorSuperclass;  
    if (eNotificationRequired())  
        eNotify(new ENotificationImpl(this, Notification.SET,  
                                      ROOMPackage.ACTOR__ACTOR_SUPERCLASS,  
                                      oldActorSuperclass, actorSuperclass));  
}
```

Implementation details



Notification Example

```
public void setActor(Actor newActor) {  
    if (newActor != eInternalContainer()  
        || (eContainerFeatureID != ROOMPackage.BINDING_ACTOR && newActor != null)){  
        if (EcoreUtil.isAncestor(this, newActor))  
            throw new IllegalArgumentException("Recursive containment not allowed for "  
                                              + toString());  
        NotificationChain msgs = null;  
        if (eInternalContainer() != null)  
            msgs = eBasicRemoveFromContainer(msgs);  
        if (newActor != null)  
            msgs = ((InternalEObject)newActor)  
                  .eInverseAdd(this, ROOMPackage.ACTOR_BINDING, Actor.class, msgs);  
        msgs = basicSetActor(newActor, msgs);  
        if (msgs != null) msgs.dispatch();  
    }  
    else if (eNotificationRequired())  
        eNotify(new  
                ENotificationImpl(this,  
                                  Notification.SET, ROOMPackage.BINDING_ACTOR, newActor, newActor));  
}
```

Implementation details

Toby McClean, Zeligsoft, 2008



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



Dynamic EMF

- We looked at this before in this module
- 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
- Supports delegation of reflective API to static API and vice-versa

Toby McClean, Zeligsoft, 2008



Automatically implemented Constraints

- Some constraints are automatically implemented
 - Derived directly from the model
- 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



Validation

- Infrastructure for providing rich invariants and constraints on the model
 - Will invoke the invariants and constraints
 - Requires the user to write the invariants and constraints
- Invariants
 - Defined by operations on a class with signature
 - (EDiagnosticChain, EMap) : EBoolean
- Constraints
 - Defined using a Validator
- More on this in a later module



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 architecture
 - EMF runtime aspects
 - Notification
 - Persistence
 - Change recording
 - Validation and constraints



Agenda

- EMF models
 - Meta-model
 - Creating models
 - Testing models
- Generating code from EMF models
 - Model manipulation
 - Model Editors
- EMF architecture and run-time
 - Transactions
 - Queries



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 ResourceSet
- Prevent other threads from observing incomplete changes
 - Classic dirty read “phenomenon”



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
 - Has its quirks, too. Most notable is its asynchronous nature: the object that sent a notification may not be in either the old state nor the new state, so processing changes takes some getting used to



Creating Read/Write Transactions

- To create a read/write transaction
 - Execute a command on the TransactionalCommandStack
 - TransactionalCommandStack::execute(Command, Map)
- Transaction can be rolled back and it will be undone automatically
- Supports undo and redo functionality
 - TransactionalCommandStack::undo
 - TransactionalCommandStack::redo



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



RecordingCommand Example

```
TransactionalCommandStack ts = ....  
ts.execute(new RecordingCommand() {  
    protected void doExecute() {  
        Iterator iter = resource.getAllContents();  
        while (iter.hasNext()) { // changes are determined on-the-fly  
            Object next = iter.next();  
            if (next instanceof Library) {  
                ((Library) next).getBooks().add(  
                    LibraryFactory.eINSTANCE.createBook());  
            }  
        }  
    }, Collections.EMPTY_MAP);
```

Implementation details

Toby McClean, Zeligsoft, 2008



Creating Read-Only Transactions

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

[Implementation details](#)



Read-only Transaction Example

```
TransactionalCommandStack ts = ....  
  
ts.runExclusive(new Runnable() {  
    public void run() {  
        while (moreToRead()) {  
            // ... do a bunch of reading ...  
            readSomeStuff();  
  
            // checking the progress monitor is a good opportunity to  
            // yield to other readers  
            if (monitor.isCancelled()) {  
                forgetIt();  
                break;  
            }  
        }  
    }  
});
```

[Implementation details](#)



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
- Post-commit listeners



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



ResourceSetListener

- Avoid reacting to changes before they have been committed
- If a transaction rolls back, no event is sent
 - Except for changes that are not undoable, such as proxy resolution and resource loading. In general, these are the changes that are compatible with a read-only transaction



Summary

- In this module we have discussed
 - EMF transactions and how to respond to their changes



Agenda

- EMF models
 - Meta-model
 - Creating models
 - Testing models
- Generating code from EMF models
 - Model manipulation
 - Model Editors
- EMF architecture and run-time
- Transactions
- **Queries**



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

Toby McClean, Zeligsoft, 2008



Query Statements

- SELECT statements filter the objects provided by a FROM clause according to the conditions specified in the WHERE clause
- `SELECT(
 <FROM>,
 <WHERE>);`

Implementation details



The FROM Clause

- Uses EMF's tree iterators to walk the objects being queried
 - Traverses the hierarchy of the elements
- Optionally specifies an EObjectCondition filter
 - Filter the source objects

Implementation details



The WHERE Clause

- The WHERE clause specifies a single filter condition
- Filters can be combined in innumerable ways using the common boolean operators
 - Not, ENOT, And, Implies, and Equivalent

[Implementation details](#)



Collection & Type Conditions

- **EObjectInstanceCondition**
 - Tests whether an object is of a particular EClass
- **IN**
 - Tests whether an object is an element of a collection

[Implementation details](#)

Toby McClean, Zeligsoft, 2008



EAttributes Conditions

- The framework includes a variety of conditions for working with primitive-valued EAttributes
 - StringLength, StringRegularExpressionValue, StringValue, SubStringValue
 - NumberCondition.* with NumberCondition.RelationalOperator
 - BooleanCondition
- EAttributeValueCondition
- Adapters convert inputs to the required data type

[Implementation details](#)



EReference Conditions

- EObjectContainmentCondition
 - Tests for the containing feature to see if it is the same as a specific EReference
- EObjectReferencerCondition
 - Tests if an EObject references another EObject
- EObjectReferenceValueCondition
 - Test the value of an EReference

[Implementation details](#)



OCL Conditions

- OCL can be used to specify WHERE clause conditions
 - OCLConstraintCondition
- Specifies a boolean-valued expression (i.e., a constraint) that selects those elements for which the expression is satisfied
- Requires that the MDT OCL component

[Implementation details](#)



The UPDATE Statement

- Passes the SELECT objects to the SET clause
- The result is the subset of the SELECT objects that were modified by the SET clause

[Implementation details](#)

Toby McClean, Zeligsoft, 2008



SELECT Query Example

```
new SELECT(  
    new FROM(objects),  
    new WHERE(new EObjectReferenceValueCondition(  
        ROOMPackage.Literals.ACTOR__ACTORSUPERCLASS,  
        EObjectInstanceCondition.IS_NULL))).execute();
```

[Implementation details](#)



SELECT Query with OCL Condition

```
new SELECT(  
    new FROM(objects),  
    new WHERE(new OCLConstraint(  
        "self.ports->isEmpty()",  
        ROOMPackage.Literals.ACTOR))).execute();
```

[Implementation details](#)



Summary

- In this module we explored
 - How to query EMF models



Questions



Toby McClean, Zeligsoft, 2008



Exploring UML in Eclipse

The what and how of the UML2 project?



Overview

- Understand profiles in UML
 - How to create and apply them
- Understand how RSA-RTE uses profiles



Goals

- After these modules you will understand how RSA-RTE extends UML for real-time and embedded modeling
- Discuss what type of extensions you could define



Agenda

- UML2 models
- Extending UML2 models
 - Keywords, profiles, stereotypes
- RSA-RTE and profiles



What is the UML2 project?

- An UML2 project 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 Eclipse UML2 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 format
 - Can be the OMG XMI 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

Toby McClean, Zeligsoft, 2008



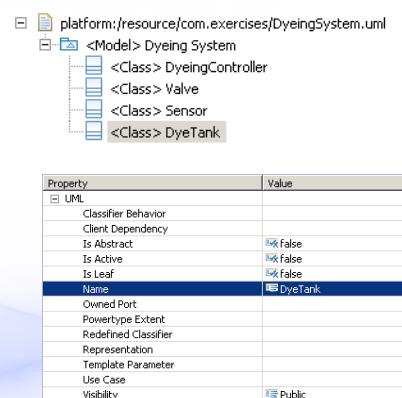
Creating a UML model

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



UML Model Editor

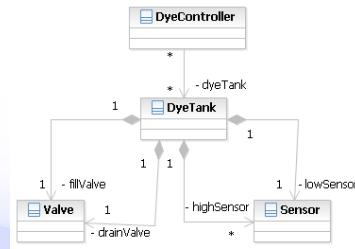
- **File → New...** from the main menu
- Select UML Model
 - Under Example EMF Model Creation Wizards
- Provide a name
- Set Model Object to Model
- Open with UML Model Editor
- Use Create Child in the context menu to create model elements





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);
```

Toby McClean, Zeligsoft, 2008



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 xml: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">
    <pakkagedElement xmi:type="uml:Class" xmi:id="_P7Dw8JutEd2e_u-cIXQ8Q" name="DyeingController"/>
    <pakkagedElement xmi:type="uml:Class" xmi:id="_UaWiwJutEd2e_u-cIXQ8Q" name="Valve"/>
    <pakkagedElement xmi:type="uml:Class" xmi:id="_XswusJutEd2e_u-cIXQ8Q" name="Sensor"/>
    <pakkagedElement 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
 - Middle weight
 - Extending the metamodel through metaclass specialization

Toby McClean, Zeligsoft, 2008



Tagging a model with keywords

- Feather weight 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
- An Eclipse UML2 project construct
 - Not part of the OMG specification



Adding Keywords

- Add an EAnnotation to the element
 - UML Editor → New Child → EAnnotations → EAnnotation
- Set the EAnnotation source attribute
 - UML
- Add a Details Entry
 - UML Editor → New Child → Details Entry
- Set the Key attribute
 - This is your keyword
- Additional keywords are added by create new Details Entry elements



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
  ► <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 a 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

```
▼ <Profile> Room
  ► <Package Import> UML 2.1.2 Metamodel
  ► <Stereotype> Room_Capsule
  ► <Stereotype> Room_Port
  ► <Stereotype> Room_Proto
  ► <Extension> Class_Room_Capsule
  ► <Extension> Port_Room_Port
  ► <Extension> Class_Room_Proto
```



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`

Toby McClean, Zeligsoft, 2008



Deploying a Static Profile

- Generate the profile code
 - Requires creating an EMF representation of the profile
- Make the project a plug-in project
 - Include the generated code and the profile model
- 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



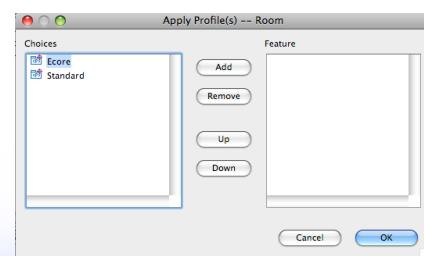
Discussion

- Examples from your experience?
 - Additional information?
 - Additional concepts?
- Does this impact modeling, validation or generation?



Applying a UML profile

- A Profile is applied to a Model, Profile 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



Toby McClean, Zeligsoft, 2008



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);
```

Implementation details

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 Conjugate	<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 type="checkbox"/> Composite
Association	<input type="checkbox"/> <>HelloWorld
Class	
Client Dependency	<input type="checkbox"/>
Default	<input type="checkbox"/>
End	
Is Behavior	<input checked="" type="checkbox"/> true
Is Derived Union	<input type="checkbox"/> false
Is Leaf	<input type="checkbox"/> false
Is Ordered	<input type="checkbox"/> false
Is Read Only	<input type="checkbox"/> false
Is Service	<input type="checkbox"/> false
Is Static	<input type="checkbox"/> false
Is Unique	<input type="checkbox"/> true
Lower	<input type="checkbox"/> 1
Name	<input type="checkbox"/> log
Protocol	

Toby McClean, Zeligsoft, 2008



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);
```

Implementation details



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");
```

Implementation details

Toby McClean, Zeligsoft, 2008



Summary

- In this module we explored
 - Extending a model with additional information
 - Through keywords and profiles
 - How to achieve this through model and code



Keywords in RSA-RTE

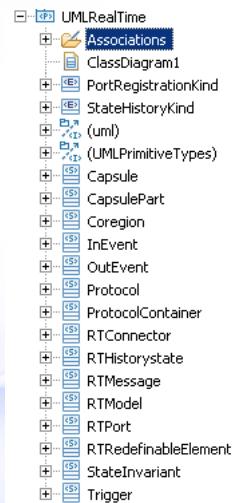
General	Keywords: <input type="text"/>		
Profiles	Applied Stereotypes:		
Stereotypes			
Documentation	Stereotype	Profile	Required
Constraints	modellibr...	Standard	False
Capabilities	RTModel	UMLRealTime	True
Relationships			
	Apply Stereotypes...		Unapply Stereotypes

Toby McClean, Zeligsoft, 2008



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

Toby McClean, Zeligsoft, 2008



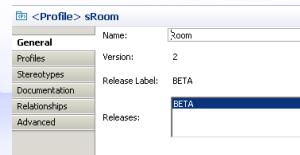
Creating Profiles in RSA-RTE

- 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

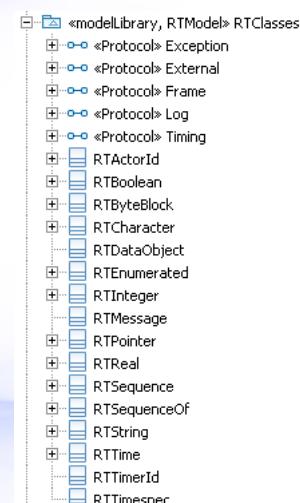
- Publishing a profile makes it available for others to install and use in their RSA-RTE models
- To publish
 - Make the project a plug-in project
 - Add the profile file to the build
 - Extend “com.ibm.xtools.uml.msl.UMLProfiles”
 - Enables RSM to find the profile
 - Publish the plug-in

```
<extension point="com.ibm.xtools.uml.msl.UMLProfiles">
    <UMLProfile id="com.zeligsoft.exercises.profiles.room"
        name="ROOM"
        path="pathmap://EXERCISE_PROFILE/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



Toby McClean, Zeligsoft, 2008



Publishing a Model Library

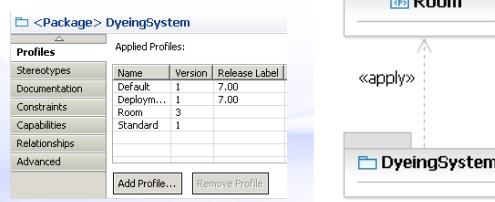
- Publishing a model library makes it available for others to import and use in their RSA-RTE models
- To publish
 - Make the project a plug-in project
 - Add the model file to the build
 - Extends com.ibm.xtools.uml.msl.UMLLibraries
 - Enables RSM to find the profile
 - Publish the plug-in

```
<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
 - Profiles and RSA-RTE
 - How to create, release and publish
 - Model Libraries



Discussion

- Any RSA-RTE concepts you would add/remove?

Toby McClean, Zeligsoft, 2008



Questions



Toby McClean, Zeligsoft, 2008



Eclipse Transformation Technologies

Exploring your model transformation options in Eclipse?



Overview

- Introduction to the eclipse transformation techniques
- Exercises to get you started



Goals

- After this module you will have
 - An understanding of the main transformation technologies
 - Understand their applicability
 - Be able to put simple transformations together



Agenda

- Model transformation introduction
- Model to model
 - Java
 - QVT
- Model to text
 - xPand
 - xTend
 - JET
- oAW

Toby McClean, Zeligsoft, 2008



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 interchange through transformation)
 - 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



Eclipse Transformation Technology

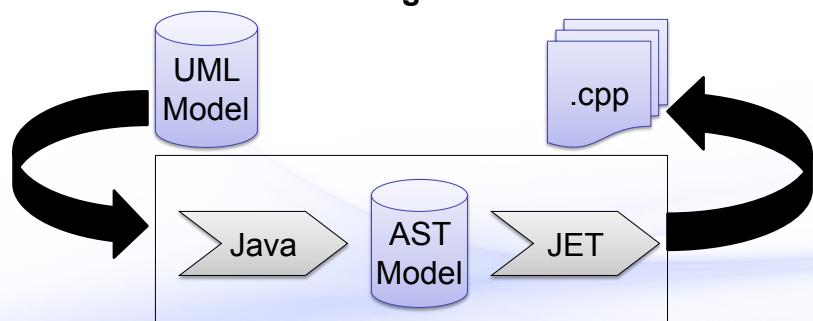
- Eclipse Technologies
 - Java
 - M2M Project - **QVT**, xTend and ATL
- RSA-RTE Technologies
 - Rational Transformation Engine

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**

RSA RTE Transformation Workflow

Lets put this into context, RSA-RTE uses a M2M then M2T to generate code





Summary

- Introduction to main transformation technologies
- Model-to-model
- Model-to-text
- Usually first do model-to-model, then model-to-text



Discussion

- What type of transformations could you imagine?
 - Some examples
 - Structural, in place
 - Capsule-to-class
 - Proxies
 - Bi-directional
 - Linking between system and design
 - Analysis
 - Walk the model and generate reports
 - Dependency, packaging violations



Agenda

- Model transformation introduction
- Model to model
 - Java
 - QVT
- Model to text
 - xPand
 - xTend
 - JET
- oAW



Eclipse Transformation Technologies

Model to Model Transformations

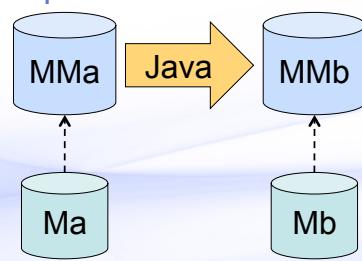
Toby McClean, Zeligsoft, 2008

Overview

- In this section we will explore the M2M technologies available in Eclipse
- The specific technologies are
 - Java
 - 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
- Transformation written against meta-model
 - Executed against instance model

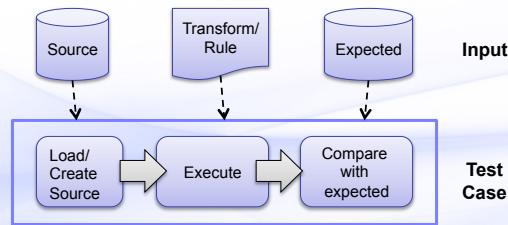


Developing a Transformation

- 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



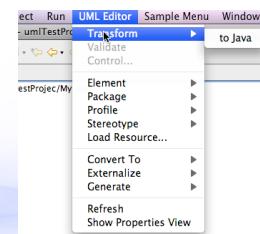
Integrating with the Workbench

- 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 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



Agenda

- Model transformation introduction
- Model to model
 - Java
 - QVT
- Model to text
 - xPand
 - xTend
 - JET
- oAW



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)

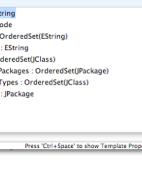
- The QVT OML is a sub-project of the M2M project
 - <http://www.eclipse.org/m2m/>
- Its goal is to provide an implementation of the MOF 2.0 Query/View/Transformation Specification
 - <http://www.omg.org/cgi-bin/doc?ptc/2007-07-07>
- Works out of the box for transforming EMF based models



Developing QVTO Transformations

- Editor
 - Syntax highlighting
 - Code completion
- Model navigation
 - Superset of EssentialOcl (OCL adaptation for EMOF)
 - Support for OCL collection operators
- Focus on transformation logic
 - Execution infrastructure left to the QVTO runtime

```
1:modeltype UML "strict" uses "http://www.eclipse.org/uml2/2.1.0/UML";
2:modelext JAVA "strict" uses "http://www.eclipse.org/emf/2002/Java";
3:
4:transformation Room2Java<in source:UML, out target:JAVA>;
5:
6:map<in> {
7:    source.rootObjects()>[UML::Model]>map toModel();
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
13:mapping (UML::Model)>toJModel< : Sequence<JAVA::CompilationUnit> {
14    int i;
15    result += self.packagedElements[UML::Class]>map toJCompilationUnit();
16}
17}
18
19-- Create a compilation unit from a UML class
20:mapping (UML::Class)>toJCompilationUnit< : JCompilationUnit {
21    none := self.name;
22    types += self.map toJClass();
23}
24
25-- Create a Java class object from a UML class
26:mapping (UML::Class)>toJClass< : {
27    none := self.name;
28    fields = self.attributes>map
29}
30
31-- Create a Java field object from a UML property
32:mapping (UML::Property)>toJField< : {
33    none := self.name;
34}
35
36-- Create a Java method from a UML operation
37:mapping (UML::Operation)>toMethod< : {
38    none := self.name;
39}
```





Toby McClean, Zeligsoft, 2008

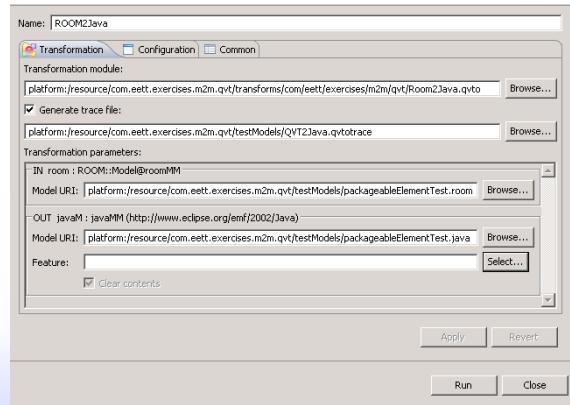


Executing QVTO Transformations

- 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



Executing QVTO Transformations



Toby McClean, Zeligsoft, 2008



Integrating with the Workbench

- Add an action to the workbench
- Create a context
- Use an interpreted QVT transformation
- Grab the input model element from a Resource or editor
- Execute the transformation
- Display or persist results and trace

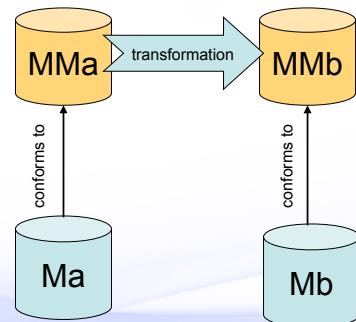


QVTO - Transformation

- 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

- Entry point is explicit and identified by a possibly 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
- Example
 - `mapping main(in rModel:Model, out jModel:JModel)`

Implementation details



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 { ... }`

Implementation details



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 read/write access
- Syntax
 - <direction> <name> : <type>
- Example
 - in prefix : String
 - out elements : Sequence(ModelElement)

Implementation details



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);

Implementation details

Toby McClean, Zeligsoft, 2008



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> })?
- Example
 - ```
mapping uml::Class::class2JClass() : JClass
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

Implementation details



## 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 result parameters
  - end
    - mapping invocations
    - logging, assertions, etc.

Implementation details



## 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

[Implementation details](#)



## 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'; }`

[Implementation details](#)

Toby McClean, Zeligsoft, 2008



## 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>}`

Implementation details



## 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);
```

Implementation details



## 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

Implementation details



## QVTO - Helpers

### ▪ Helper example

```
* helper UML::PrimitiveTypeToJava(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);
}
```

Implementation details

Toby McClean, Zeligsoft, 2008



## 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

Implementation details



## QVTO – Intermediate Data

- Intermediate class syntax
  - `intermediate class <name> {<attributes>}`
- Intermediate class example
  - `intermediate class LeafAttribute  
  { name : String; kind:String; attr:Attribute }`
- Intermediate property syntax
  - `intermediate property <name> : <type>;`
- Intermediate property example
  - `intermediate property UMLClass::allAttributes :  
    Sequence (UML::Property);`

Implementation details



## 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

Implementation details



## QVTO – Resolving Objects

- Deferred resolution example
  - `protocol.late resolveoneIn(JClass);`
- Specific mapping
  - `protocol.resolveoneIn(Protocol::protocol2JClass, JClass);`
- Filtered resolution example
  - `protocol.resolveone (name = 'Control');`
  - `protocol.resolveone (p : JClass | p.name = 'Control');`
- Resolve multiple objects example
  - `protocol.resolve (JClass);`

Implementation details

Toby McClean, Zeligsoft, 2008



## 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)`

Implementation details



## QVTO – Mapping Reuse

- **Inheritance**
  - Inherited mapping is executed after the init section
  - `mapping A::AtoSubB() : SubTypeofB inherits A::AtoB {...}`
    - Executes init of AtoSubB then AtoB then the rest of AtoSubB
- **Merge**
  - List of mappings executed in sequence after end section
  - `mapping A::AtoB() : B`  
`merges A::toSuperB1, A::toSuperB2 {}`
    - Executes AtoB then toSuperB1, then toSuperB2
- Parameters of inherited/merged mappings must match

Implementation details



## 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

```
mapping PackageableElement::roomElement2JCompilationUnit()
: JCompilationUnit
 disjuncts Actor::actor2JCompilationUnit,
 Protocol::protocol2JCompilationUnit {
}
```

[Implementation details](#)



## 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>}

[Implementation details](#)

Toby McClean, Zeligsoft, 2008



## QVTO – IF Example

```
init {
 result := object JClass{};
 if(self.isStereotypedBy('Capsule') then {
 result.name := self.name + 'C';
 } else {
 result.name := self.name + 'P';
 }endif;
}
```



## QVTO – FOREACH Example

```
init {
 ...
 self.property->foreach(p | p.isStereotypedBy('Port')) {
 result.member += object JField{ name:= p.name; };
 }

 range(1, 5)->forEach(i) {...}
}
```



## 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> {}

Implementation details



## 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

Implementation details

Toby McClean, Zeligsoft, 2008



## 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;**

[Implementation details](#)



## 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

[Implementation details](#)



## 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;
}
```

Implementation details



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

Implementation details

Toby McClean, Zeligsoft, 2008



QVTO - Programatically Invoking

```
URI transformation =
    URI.createURI("platform:/resource/com.eett.exercises.m2m.qvt/transforms/
Room2Java.qvto");
IFile qvtFile = getIFile(transformation);
IContext qvtContext = new Context();
QvtInterpretedTransformation trans =
    new QvtInterpretedTransformation(qvtFile);
EObject inputModel =
    getInput(
        URI.createURI("platform:/resource/com.eett.exercises.m2m.qvt/testModels/
packageableElementTest.room"));
EObject[] inputs = {inputModel};
TransformationRunner.In input =
    new TransformationRunner.In(inputs, qvtContext);
TransformationRunner.Out output = null;

try {
    output = trans.run(input);
} catch (MdaException e) {
    out.println("Error running transformation!");
    out.println(e.getMessage());
}
```



Summary

- In this module we explored
 - QVT mappings
 - Invoking QVT
 - Details on QVT usage



Agenda

- Model transformation introduction
- Model to model
 - Java
 - QVT
- Model to text
 - xPand
 - xTend
 - JET
- oAW



Eclipse Transformation Technologies

M2T with xPand

Toby McClean, Zeligsoft, 2008



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 non-standardized 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



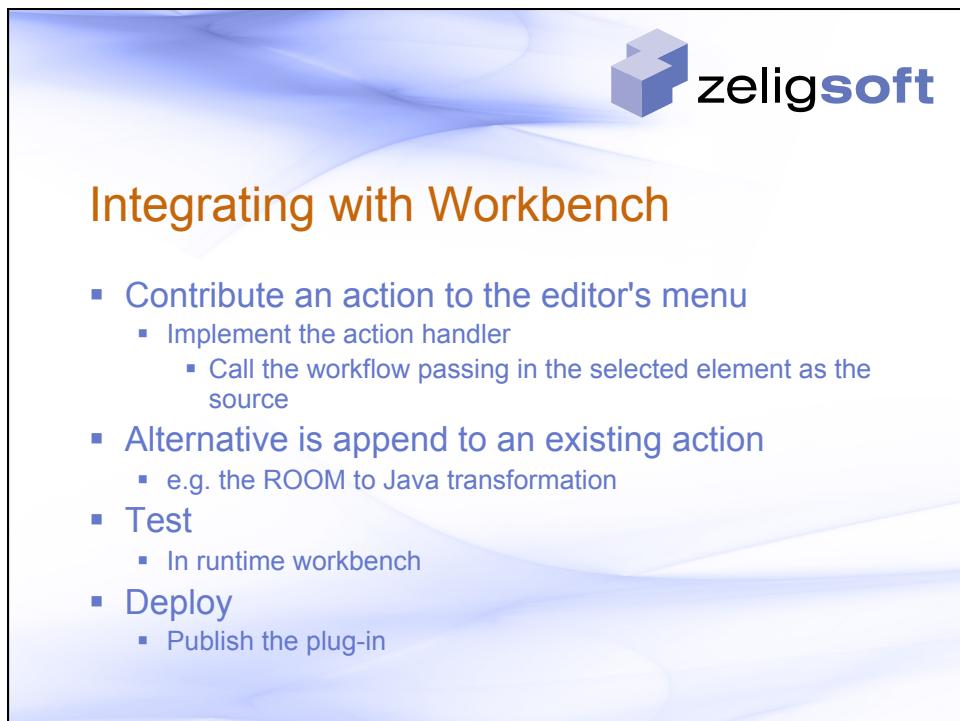
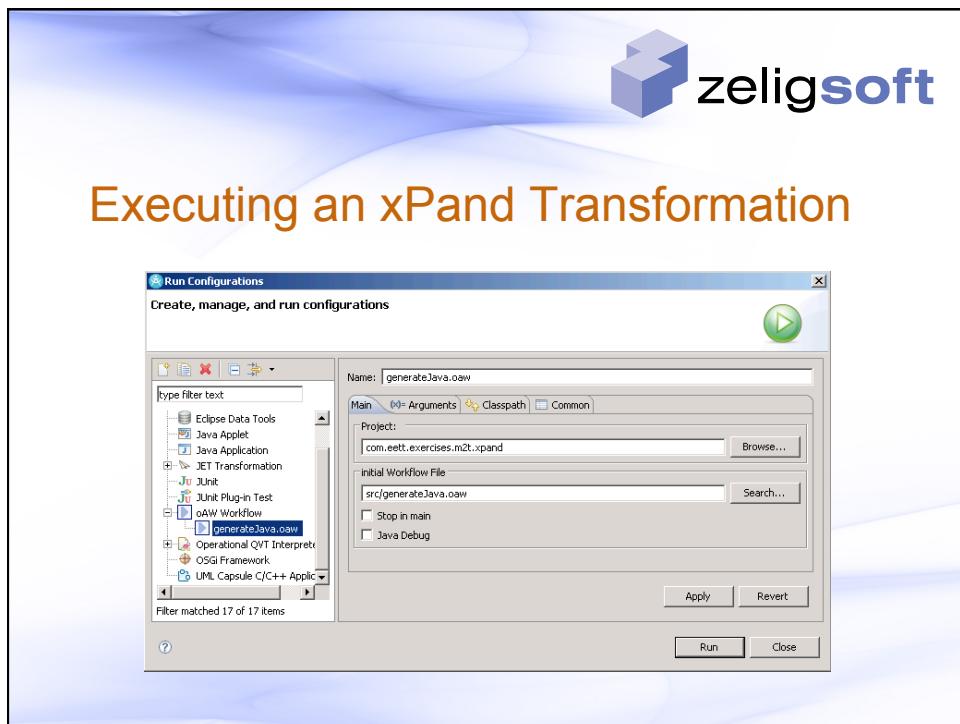
Developing xPand Transformations

- Editor
 - Syntax highlighting
 - Code completion
- Model navigation
 - Java based syntax for model navigation
 - Has operators for working with collections
- Focus on transformation logic
 - Execution infrastructure left to the xPand runtime

A screenshot of a computer screen displaying an xPand transformation script named "Generate.java.xpt". The code uses Java-like syntax for defining main methods, imports, and loops over elements. It includes sections for writing compilation units and handling imports.

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 XpandFacade class
 - Use the Run as... or Debug as... Java Application





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 - 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»

Toby McClean, Zeligsoft, 2008



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 – Templates Example

```
<<DEFINE writeJCompilationUnit FOR JCompilationUnit>>
<<IF this.name != null>>
  <<FILE this.name + ".java">>

<<REM>>
  The imports required by this compilation unit that
  may be packages, types or freeform
<<ENDREM>>
<<FOREACH this.importedPackages AS i>>
  import <<i.name>>;
<<ENDFOREACH>>
<<FOREACH this.importedTypes AS i>>
  import <<i.name>>;
<<ENDFOREACH>>
<<FOREACH this.imports AS i>>
  import <<i>>;
<<ENDFOREACH>>
```

Implementation details



xPand – Template Example (2)

```
<<REM>>
  Write out the source for each of the classes in the
  compilation uni
<<ENDREM>>
<<EXPAND writeJClass FOREACH this.types>>
<<ENDIF>>
<<ENDDEFINE>>
```

Implementation details

Toby McClean, Zeligsoft, 2008



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 a workflow and referenced
 - Workflow
 - <outlet path='main/src-gen'> **-- default**
 - <outlet name='TO_SRC' path='main/src' overwrite='false'>
 - «FILE 'test/note.txt'»# this goes to the default outlet«ENDFILE»
 - «FILE 'test/note.txt' TO_SRC»# this goes to the TO_SRC outlet«ENDFILE»

Implementation details



xPand - Invoking a Template

- The xPand language uses EXPAND to invoke a template
 - «EXPAND definitionName [(parameterList)] [FOR expression | FOREACH expression [SEPARATOR expression]]»
- FOR
 - Invokes the template on the specified element
- FOREACH
 - Invokes the template on each element in the collection
- SEPERATOR
 - Specifies an optional delimiter output between each invocation
- Omitting FOR\FOREACH invokes with FOR this

Implementation details



xPand – Invoking a Template

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

Implementation details



xPand – Invoking Example

- Implicit element

```
<<DEFINE writeJCompilationUnit FOR JCompilationUnit>>
  <<EXPAND writePackage>>
<<ENDDEFINE>>
```

- Iterate over elements

```
<<DEFINE writeJField FOR JField>>
  <<EXPAND writeFieldType FOR this.type>>
<<ENDDEFINE>>
```

- Iterate over elements

```
<<DEFINE main FOR JModel>>
  <<EXPAND writeJCompilationUnit FOREACH this.elements>>
<<ENDDEFINE>>
```

Implementation details

Toby McClean, Zeligsoft, 2008



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
- **FOREACH**
 - execute the contents for each element in a collection
- **ERROR**
 - aborts evaluation with the specified message

[Implementation details](#)



xPand - LET

```
«LET jClass.package.name + '.' + jClass.name AS qualifiedName»  
/**  
 * The fully qualified name of the class is «qualifiedName»  
 **/  
«ENDLET»
```

[Implementation details](#)



xPand – IF, ELSEIF, ELSE

```
«REM»Output the return type for the operation
«ENDREM»
«IF jOperation.type != null»
    «jOperation.type.name»
«ELSE»
    void
«ENDIF»
```

Implementation details



xPand - FOREACH

```
«FOREACH jClass.member AS jMember»
    «IF jMember.metatype == JField»
        //Output for a Java field
    «ELSEIF jMember.metatype == JOperation»
        //Output for a Java operation
    «ENDIF»
«ENDFOREACH»
```

Implementation details

Toby McClean, Zeligsoft, 2008



Summary

- In this module we explored
 - xPand transformations
 - Invoking xPand
 - xPandsyntax



Agenda

- Model transformation introduction
- Model to model
 - Java
 - QVT
- Model to text
 - xPand
 - **xTend**
 - JET
- oAW



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



xTend Example

```
import java;
import org::eclipse::emf::java;

String qualifiedName(JClass jClass) :
    jClass.package.name + "." + jClass.name;

String javaFileName(JCompilationUnit unit) :
    JAVA com.eett.exercises.m2t.GenerateJava.javaFileName
        (org.eclipse.emf.java.JCompilationUnit);
```

Implementation details

Toby McClean, Zeligsoft, 2008



Summary

- In this module we have covered
 - A short introduction to xTend



Agenda

- Model transformation introduction
- Model to model
 - Java
 - QVT
- Model to text
 - xPand
 - xTend
 - JET
- oAW



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
 - JSP 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

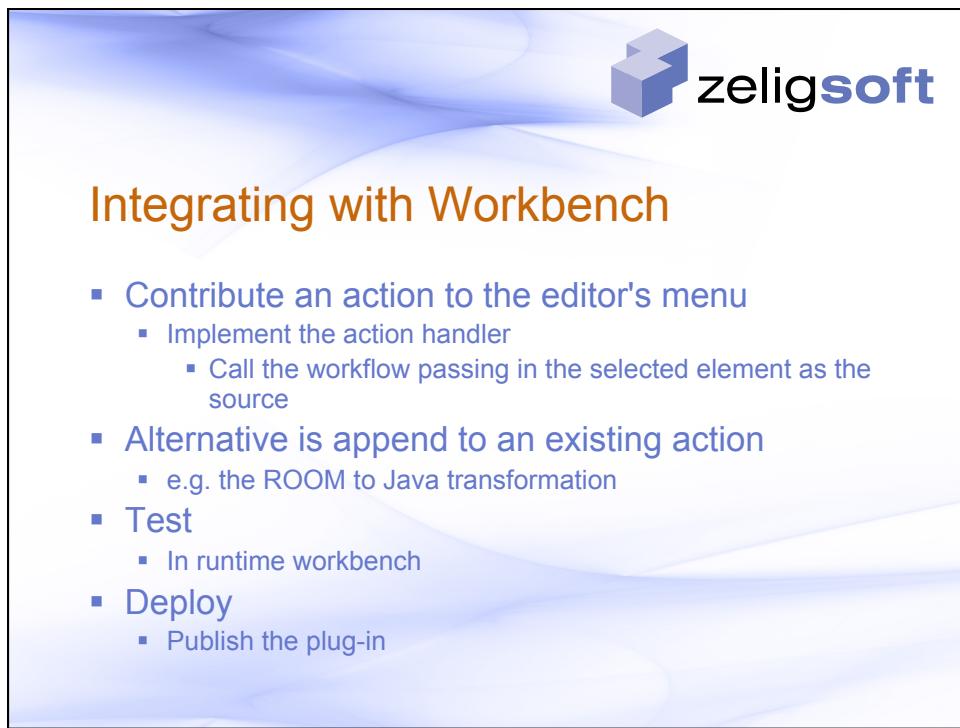
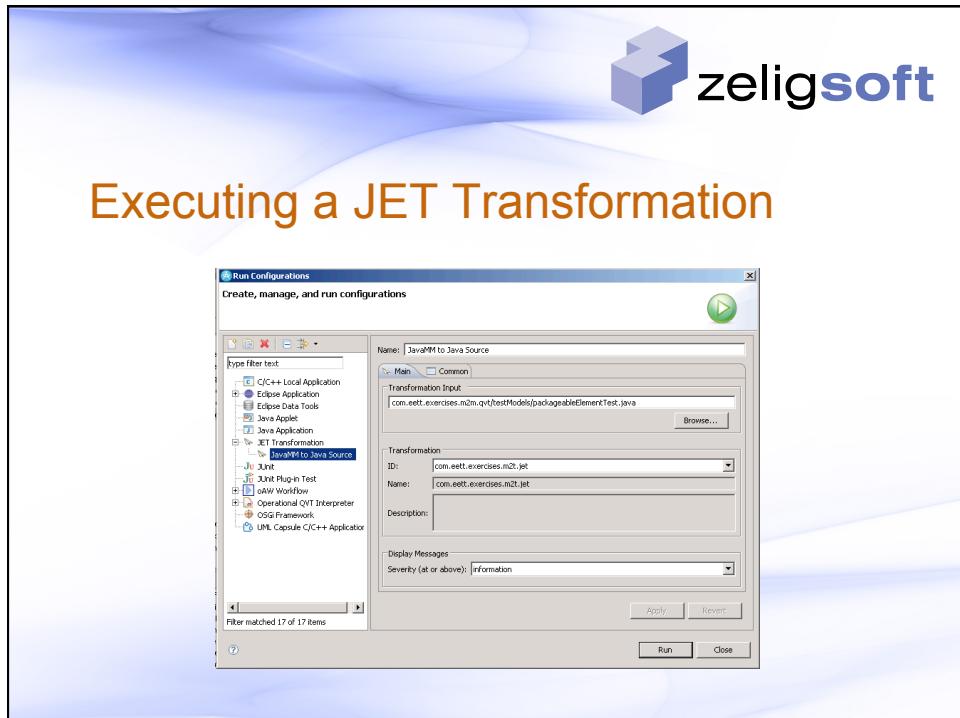


Developing a JET Transformation

- 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

Toby McClean, Zeligsoft, 2008





Toby McClean, Zeligsoft, 2008



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)
 - <% ... %>

Implementation details



JET – Directives

- **@jet**
 - Control or affect the Java code that is created by the JET compiler
 - <%@jet package="" class="" imports="" startTag="" endTag="" />
- **@taglib**
 - Import a tag library that is used in the template and assign it to a namespace
 - Control tags – org.eclipse.jet.controlTags
 - Workspace tags – org.eclipse.jet.workspaceTags
 - Java tags – org.eclipse.jet.javaTags
 - Format tags – org.eclipse.jet.java.formatTags

[Implementation details](#)



JET - Declarations

- Used to declare Java methods and fields that are part of the class generated by the template
- Any syntactically correct method or field declaration is valid
- **<%! declaration %>**
- **Example**
 - <%! private String qualifiedName; %>

[Implementation details](#)

Toby McClean, Zeligsoft, 2008



JET – Expressions

- A valid Java expression which will be evaluated and emitted
 - They do not include a ";" at the end
- Has access to any Java element in scope
 - Including implicit objects context and out
- <%= **expression** %>
- Example
 - <%= 3 + 4 %>

Implementation details



JET – Scriptlets

- One or more Java statements
 - <% **statement+** %>
- Has access to any Java element in scope
 - Including implicit objects context and out
- A block can be split between Scriptlets
 - <% if(jCompilationUnit.getName() != null) { %>
 - ...
 <% } // end if %>

Implementation details



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 imports="org.eclipse.emf.java.*" %>`
 - Can use the schema for an EMF model to control inputs

```
<transform  
    modelLoader="org.eclipse.jet.emf"  
    modelSchema="emf.java.xsd"  
    startTemplate="templates/main.jet"  
    templateLoaderClass="com.eett.exercises.m2t.jet.co  
<description></description>  
<tagLibraries>
```

Implementation details



JET – Loading Models

- Tags in JET used to load content
 - Load the content into a variable so it can be referenced
- `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

Implementation details

Toby McClean, Zeligsoft, 2008



JET - Extensions

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

[Implementation details](#)



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

[Implementation details](#)



JET – A template of a Template

- **Template directives**
 - Configure the output of the JET compiler
 - Reference the tag libraries to be used
- **Compute derived attributes by traversing model**
 - Consider this the annotated model
- **Perform transformations on annotated model**
 - Creating projects, folders and files
- **Post transformation actions**
 - Template specific actions outside transformation logic

[Implementation details](#)



JET - Output

- **The output control in M2T is critical**
 - Out of the box JET provides several tags that help with output produced by the transformation
 - Found in the formatting tag library
- **f:indent**
 - Indent the contents the specified number of times
- **f:lcl, 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

[Implementation details](#)

Toby McClean, Zeligsoft, 2008



JET - Control

- The control tag library provides capabilities for putting control logic into your template
- **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

[Implementation details](#)



JET – Control (c:choose)

▪ Syntax

- `<c:choose select="[xpath]">[content]</c:choose>`
- Select when used specifies the value to be used in the when tags test attribute

▪ Example

```
<c:choose>
  <c:when test="$jCompilationUnit/@name != ''">
    <ws:file path="{concat($jCompilationUnit/@name, '.java')}"
             template="templates/JCompilationUnit.java.jet"/>
  </c:when>
  <c:otherwise></c:otherwise>
</c:choose>
```

[Implementation details](#)



JET – Control (c:if)

- Syntax

- <c:if test="[condition]" var="[var name]"></c:if>
- var is optional and if specifies stores the result of evaluating the condition before it is converted to a boolean

- Example

```
<c:if test="$jCompilationUnit/@name != ''">
    <ws:file path="{concat($jCompilationUnit/@name, '.java')}"
              template="templates/JCompilationUnit.java.jet"/>
</c:if>
```

Implementation details



JET – Control (c:iterate)

- Syntax

- <c:iterate select="" var="" delimiter=""></c:iterate>
- select – xpath that returns node set or number
- var – the variable referencing the current iterator object
- delimiter – string written to output between iterations but not the last

- Example

```
<c:iterate select="$jCompilationUnit/types" var="type">
    class <c:get select="$type/@name /> {
    }
</c:iterate>
```

Implementation details

Toby McClean, Zeligsoft, 2008



JET - Expressions

- JET has several tags for working model elements and variables in the logic of the template
- **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

[Implementation details](#)



JET – Expressions (c:get)

- **Syntax**
 - `<c:get select="[xpath]" default="[value]" />`
 - select is the xpath to evaluate, if it selects nothing than an error *may* occur
 - default (optional) is the value to use if the XPath expression returns nothing, prevents error
- **Example**

```
<c:get select="$jClass/@name" default=" " />
```

[Implementation details](#)



JET – Expressions (c:set)

- **Syntax**

- `<c:set select="[XPath]" name="[name]">[value]</c:set>`
- select is XPath expression selecting the element to add or set the attribute specified by the value of name on
- creates attribute if none exists

- **Example**

```
<c:set select="$jCompilationUnit" name="fileName">
  <c:get select="concat($jCompilationUnit/@name, '.java')"/>
</c:set>
```

[Implementation details](#)



JET – Expressions (c:setVariable)

- **Syntax**

- `<c:setVariable select="[xpath]" var="[name]" />`
- Assign the result of evaluating select to the variable specified by var

- **Example**

```
<c:setVariable select="/" var="jModel"/>
```

[Implementation details](#)

Toby McClean, Zeligsoft, 2008



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)
 - Example
 - <c:include template="templates/header.jet.inc" />
 - Overriding a transformation
 - In the transform extension point declare that the transformation overrides templates in the overridden transformation

Implementation details



JET - Invoking a Transform

- To invoke another template from the current template use the c:invokeTransform
 - passes the current transformation's source and context variables
- Syntax
 - <c:invokeTransform transformId="" passVariables="" />
 - passVariables is optional
- Example
 - <c:invokeTransform transformId="com.eett.exercises.m2t.jet.writejava" />

Implementation details



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
 - Make use of declarations, expressions and scriptlets
- Create templates that encapsulate the handling of specific stereotypes
 - Use the c:include to execute the template in place

[Implementation details](#)



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

[Implementation details](#)

Toby McClean, Zeligsoft, 2008



JET – Logging (c:log)

- Examples
- <c:if test="\$jClass/@name = "">
 <c:log severity="error">
 Can not write a class that has no name
 </c:log>
</c:if>
- <c:log>
 Writing <c:get select="\$jClass/@name" />
</c:log>

Implementation details



JET – Logging (c:dump)

- Examples
- <c:if test="\$jClass/@name = "">
 <c:log severity="error">
 Can not write a class that has no name
 </c:log>
 <c:dump select="\$jClass" />
</c:if>

Implementation details



JET – Logging (c:marker)

- Example
 - <c:marker description="concat(\$op/@name, ' needs an implementation')>
 throw UnsupportedOperationException();
</c:marker>
- A task marker will be created in the Eclipse task view identifying to the user that something needs to be done

[Implementation details](#)



Summary

- We have explored JET
 - What is JET
 - How I develop a model to text transformation with JET
 - How is my transformation integrated with Eclipse
 - The features and syntax of JET
- You should now be able to
 - Do the JET exercises
 - Navigate around the JET documentation and other resources

Toby McClean, Zeligsoft, 2008



Agenda

- Model transformation introduction
- Model to model
 - Java
 - QVT
- Model to text
 - xPand
 - xTend
 - JET
- oAW



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?

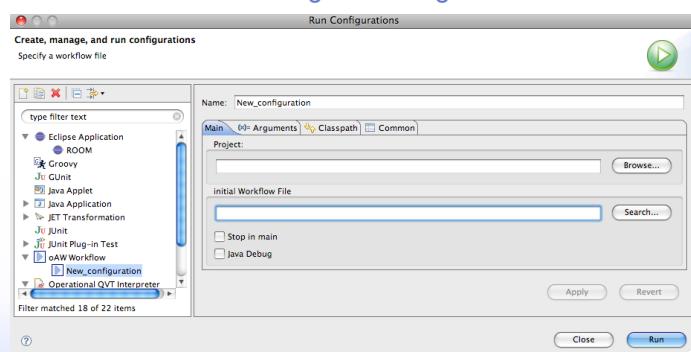
- API allows for custom workflow components
 - e.g. QVT transformation execution
- Configuration properties
 - Properties passed into the workflow
 - <property name='targetDir' value='src-gen/'>
- Slots or variables
 - Simple syntax, variables are referred to by name
 - No declaration
 - <component id="generator"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

Toby McClean, Zeligsoft, 2008



Executing an oAW Workflow

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



Discussion

- We can combine the previous module and this module
 - Extend the model with additional information
 - Extend generation to use this information
 - May require RTS customization
- Examples?
 - Deadlines in messages
 - Port patterns



Questions



Toby McClean, Zeligsoft, 2008



Validating Eclipse Models

How can I ensure valid models?



Overview

- Explore the validation service
- Explore the different types of constraints



Goals

- After this module you understand how validation is performed in the Eclipse environment
- You will be able to add your own constraints



Agenda

- Overview
- Static and dynamic constraint providers
- Constraints
- Validation service
- Creating constraints

Toby McClean, Zeligsoft, 2008



What can I Validate?

- The validation framework provides a way to describe constraints to go with models
 - To protect model sanity
 - To validate domain rules
 - To validate your specific rules



Validation Framework Overview

- Constraints are organized through categories and bindings
- Constraint providers contribute constraints
- Constraint parsers implement implementation languages
- Traversal strategies walk the model
- Validation listeners are notified when validation occurs
- Notification generators for custom (esp. higher-order) notifications for processing in live validation



What does the Validation Framework Provide?

- Invocation (Triggers)
 - “Batch” validation: initiated by the user or by the system on some important event, validates all or a selected subset of a model
 - “Live” validation: initiated automatically by the system to validate a set of changes performed during some transaction. The semantics of “transaction” are defined by the client
 - Constraints can specify which particular changes trigger them (by feature and event type)
- Support for OCL constraints
 - EMFT Validation uses the OCL component of the MDT project to provide out-of-box support for specifying constraints using OCL
 - API supports UML binding and more flexibility in working with OCL constraints embedded in metamodels



Constraint Providers

- Constraint providers are of two flavours: static and dynamic
 - Static providers declare their constraints in the plugin.xml of a client plug-in of the constrained model
 - Dynamic providers obtain constraints from an arbitrary source at run-time
- Both kinds of providers can declare constraint categories and include their constraints in categories defined by any provider
 - Categories are hierarchical namespaces for constraints
 - Constraints are grouped by category in the preference page

Toby McClean, Zeligsoft, 2008



Static Constraint Provider

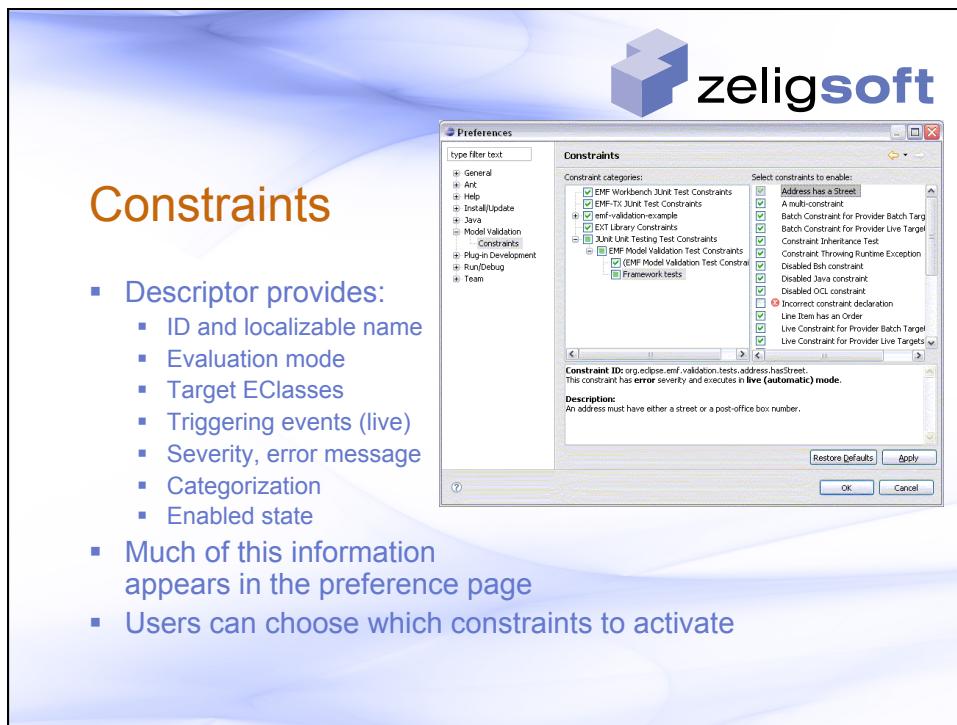
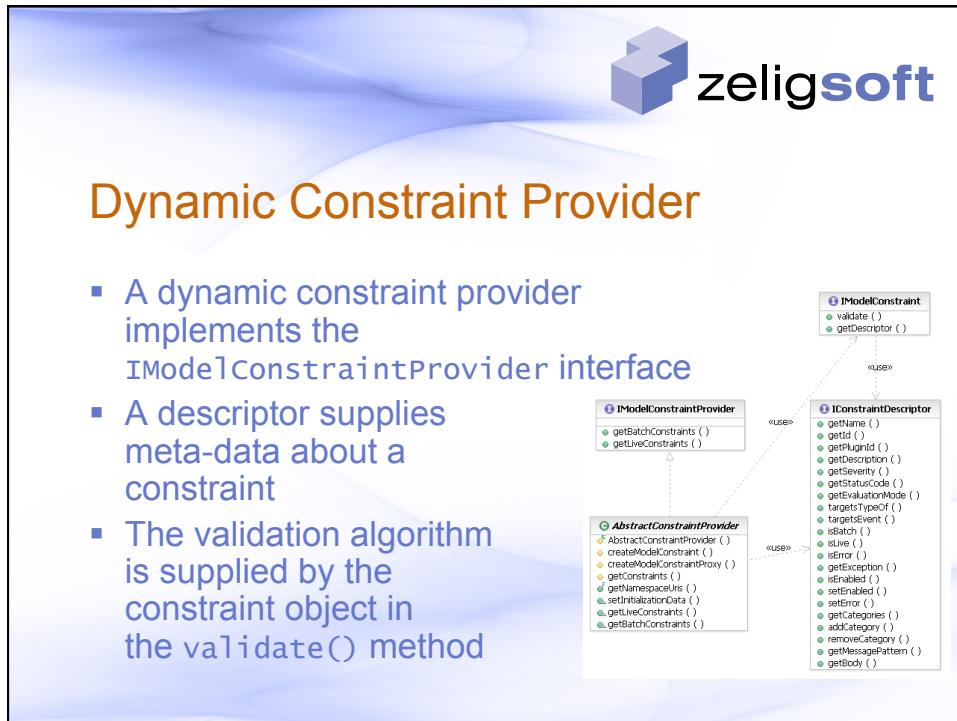
```
<extension point="org.eclipse.emf.validation.constraintProviders">
  <category name="Library Constraints" id="com.example.library">
    <constraintProvider>
      <package namespaceUri="http://www.eclipse.org/Library/1.0.0"/>
      <constraints categories="com.example.library">
        <constraint
          lang="Java"
          class="com.example.constraints.UniqueLibraryName"
          severity="WARNING"
          mode="Batch"
          name="Library Must have a Unique Name"
          id="com.example.library.LibraryNameIsUnique"
          statusCode="1">
          <description>Libraries have unique names.</description>
          <message>{0} has the same name as another library.</message>
          <target class="Library"/>
        </constraint>
      </constraints>
    </constraintProvider>
  </extension>
```



Dynamic Constraint Provider

- Registered by name of a class implementing the `IModelConstraintProvider` interface
- Indicate the packages for which they supply constraints
 - Biggest difference is the absence of a `<constraints>` element
- System can optionally cache the provided constraints

```
<extension point="org.eclipse.emf.validation.constraintProviders">
  <category name="Library Constraints" id="com.example.library">
    <constraintProvider
      class="com.example.MyConstraintProvider"
      cache="false">
      <package namespaceuri="http://www.eclipse.org/Library/1.0.0"/>
    </constraintProvider>
  </extension>
```



Toby McClean, Zeligsoft, 2008

Evaluation Modes

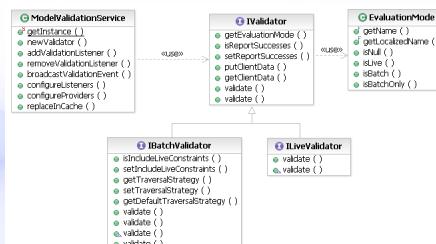
- Batch validation is usually explicitly requested by the user, via a menu action or toolbar button
- Live validation is performed automatically as changes are made to EMF resources
 - Live constraints indicate which specific changes trigger them

```

<constraint
    mode="Live"
    ...
    >
    <description>Libraries have unique names.</description>
    <message>{0} has the same name as: {1}.</message>
    <target class="Library">
        <event name="Set">
            <feature name="name"/>
        </event>
    </target>
</constraint>
  
```

Validation Service

- Evaluation of constraints is performed via the Validation Service
- The service provides validators corresponding to the available evaluation modes
- By default, batch validation includes live constraints, also, for completeness





Validation Service

- To validate one or more elements, in batch mode, simply create a new validator and ask it to validate:

```
List objects = myResource.getContents(); // objects to validate

// create a validator
IValidator validator = ModelvalidationService.getInstance()
    .newValidator(EvaluationMode.BATCH);

// use it!
IStatus results = validator.validate(objects);

if (!results.isOK()) {
    ErrorDialog.openError(null, "validation", "Validation Failed",
        results);
}
```



Validation Service

- Live validation does not validate objects, but rather notifications indicating changes to objects

```
List<Notification> notifications = ... ; // some changes that we observed

// create a validator
IValidator validator = ModelvalidationService.getInstance()
    .newValidator(EvaluationMode.LIVE);

// use it!
IStatus results = validator.validate(notifications);

if (!results.isOK()) {
    ErrorDialog.openError(null, "validation", "Validation Failed",
        results);
}
```



Creating Constraints

- Specifying a constraint doesn't necessarily require any code
- Requires the OCL component of the MDT project

```
<constraint
    lang="OCL"
    mode="Batch"
    ...
    >
<description>Libraries have unique names.</description>
<message>{0} has the same name as: {1}.</message>
<target class="Library"/>

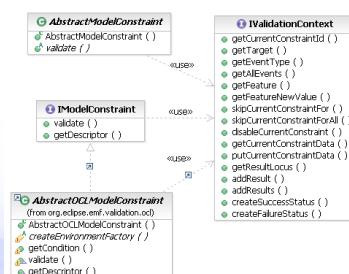
<![CDATA[
Library.allInstances()->forAll(l |
    l <> self implies l.name <> self.name)
]]>

</constraint>
```



Creating Constraints

- Sometimes the easiest or best way to formulate a constraint is in Java
- Java constraints extend the `AbstractModelConstraint` class
- The validation context provides the `validate()` method with information about the validation operation





Validation Context

- Provides the element being validated (the target)
- Indicates the current evaluation mode (live or batch)
 - In case of live invocation, provides the changed feature and the event type
- Allows constraints to cache arbitrary data for the duration of the current validation operation
- Provides convenient methods for reporting results
- Provides the ID of the constraint that is being invoked

IValidationContext
● getCurrentConstraintId ()
● getTarget ()
● getEventType ()
● getAllEvents ()
● getFeature ()
● getFeatureNewValue ()
● skipCurrentConstraintFor ()
● skipCurrentConstraintForAll ()
● disableCurrentConstraint ()
● getCurrentConstraintData ()
● putCurrentConstraintData ()
● getResultLocus ()
● addResult ()
● addResults ()
● createSuccessStatus ()
● createFailureStatus ()

Creating Constraints

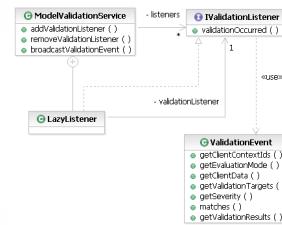
```
public class LibraryNameIsUnique extends AbstractConstraint {  
    public IStatus validate(IValidationContext ctx) {  
        Library target = (Library) ctx.getTarget(); // object to validate  
  
        // does this library have a unique name?  
        Set<Library> libs = findLibrariesWithName(target.getName());  
        if (libs.size() > 1) {  
            // report this problem against all like-named libraries  
            ctx.addResults(libs);  
  
            // don't need to validate these other libraries  
            libs.remove(target);  
            ctx.skipCurrentConstraintFor(libs);  
  
            return ctx.createFailureStatus(new Object[] {  
                target, libs});  
        }  
  
        return ctx.createSuccessStatus();  
    }  
}
```

Toby McClean, Zeligsoft, 2008

Listening for Validation Events

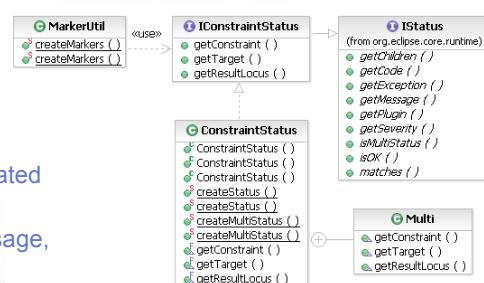
- Every validation operation executed by the Validation Service generates an event
- The event indicates the kind of validation performed, on what objects, and what were the results (incl. severity)
- Listeners registered on the extension point are lazily initialized when their selection criteria are met

```
<extension point="org.eclipse.emf.validation.validationListeners">
    <listener class="com.example.validation.ProblemsReporter">
        <clientContext id="com.example.MyClientContext"/>
    </listener>
</extension>
```



Reporting Problems

- Problems are reported as **IConstraintStatus** objects. A status knows:
 - The constraint that was violated
 - The objects that violated it
 - The severity and error message, as usual for an **IStatus**
- The marker utility encodes all of this information in a problem marker on the appropriate resource, to show in the Problems view





Summary

- We have discussed validation in Eclipse
- Static, dynamic, live and batch
- You have learned how to create a constraint



RSA-RTE Validation

Validating models in RSA-RTE

Toby McClean, Zeligsoft, 2008



Overview

- Adding constraints to models in RSA-RTE
 - Authoring
 - Executing
- Adding constraints to profiles in RSA-RTE
 - Authoring
 - Executing



Authoring Model Constraint

- Constraints can be added to model elements
 - Add UML → Constraint from the elements context menu
 - Defined by
 - Name
 - Constrained elements
 - Modeling level
 - Language
 - Mode
 - Message

A screenshot of the RSA-RTE interface. At the top, there's a 'Class Rule' dialog box with fields for 'Name' (DyeingController), 'Owner' (DyeingController), 'Type' (Rule), 'Modeling Level' (Model), 'Language' (OCL), and 'Value'. Below it is a 'MetaConstraint' dialog box with sections for 'General' (Evaluation Mode: Batch), 'Validation' (Message:), and 'Constrained Elements' (Severity: Error).



Executing Model Constraint

- Batch constraints are evaluated by
 - Validate in the context menu of the element
- Live constraints are evaluated by
 - Validate in the context menu
 - i.e. they are all Batch
- You will see the results if any
 - In the problems view



Authoring Profile Constraint

- Profile authored in the same way as model constraints
- They apply to elements that the stereotype is applied to
- Nothing special is needed to have them work in a deployed profile
- This is specific to RSx you would have to create your own with Eclipse UML2



Executing Profile Constraint

- Batch mode constraints evaluate
 - When the model or model element is validated
 - They are reported in the Problems view
- Live mode constraints evaluate
 - When the model change is made
 - They are reported in a dialog and in the Problem view



Summary

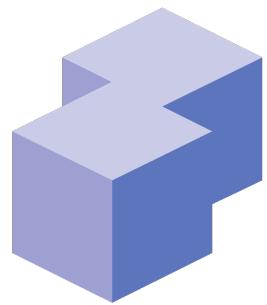
- We have discussed validation specific to RSx
- How to author model specific constraints
- How to author constraints in profiles



Questions



Toby McClean, Zeligsoft, 2008



zeligsoft

Eclipse Transformation Training
Exercise Workbook

1 Plug-in Exercises

1.1 What this exercise is about

In this exercise you will extend the Workbench

At the end of this exercise you should be able to

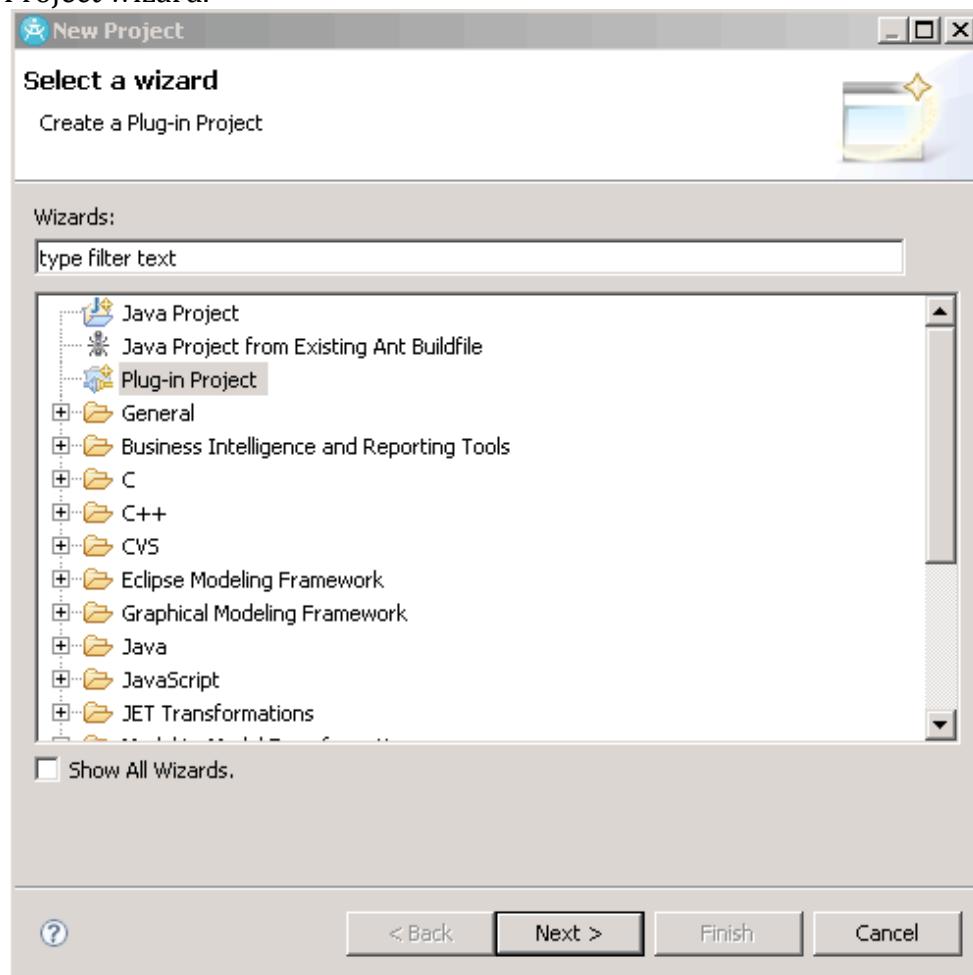
- Create Eclipse Plug-in Projects
- Add menus to the Workbench

1.2 Create a new project

The first step in creating our plug-in is to create a plug-in project using the Eclipse New Project wizard. We will create an empty Eclipse Plug-in project that contributes to the Workbench UI.

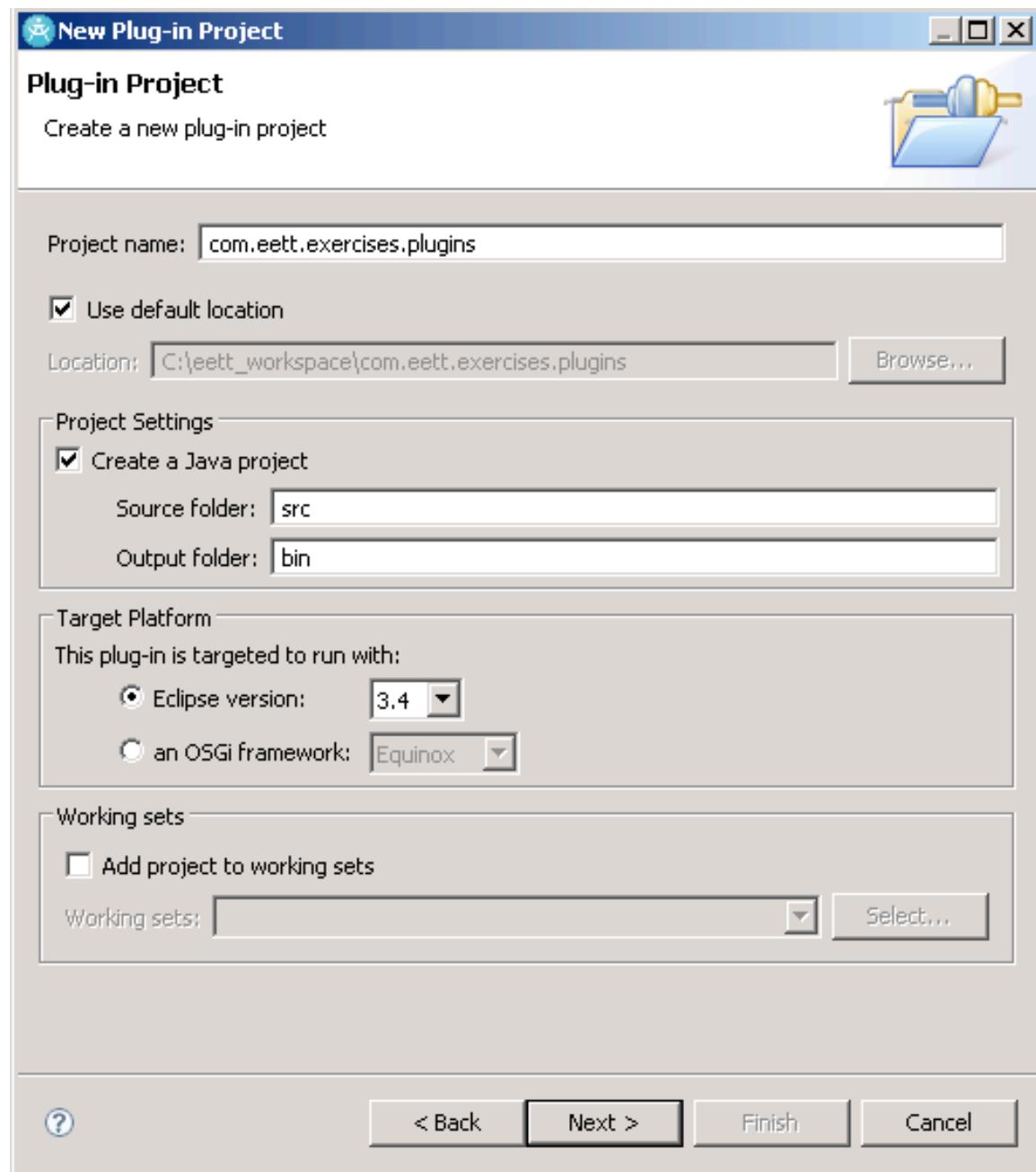
From the File menu New → Project...

Select Plug-in Project from the list of possible projects to create in the New Project wizard.

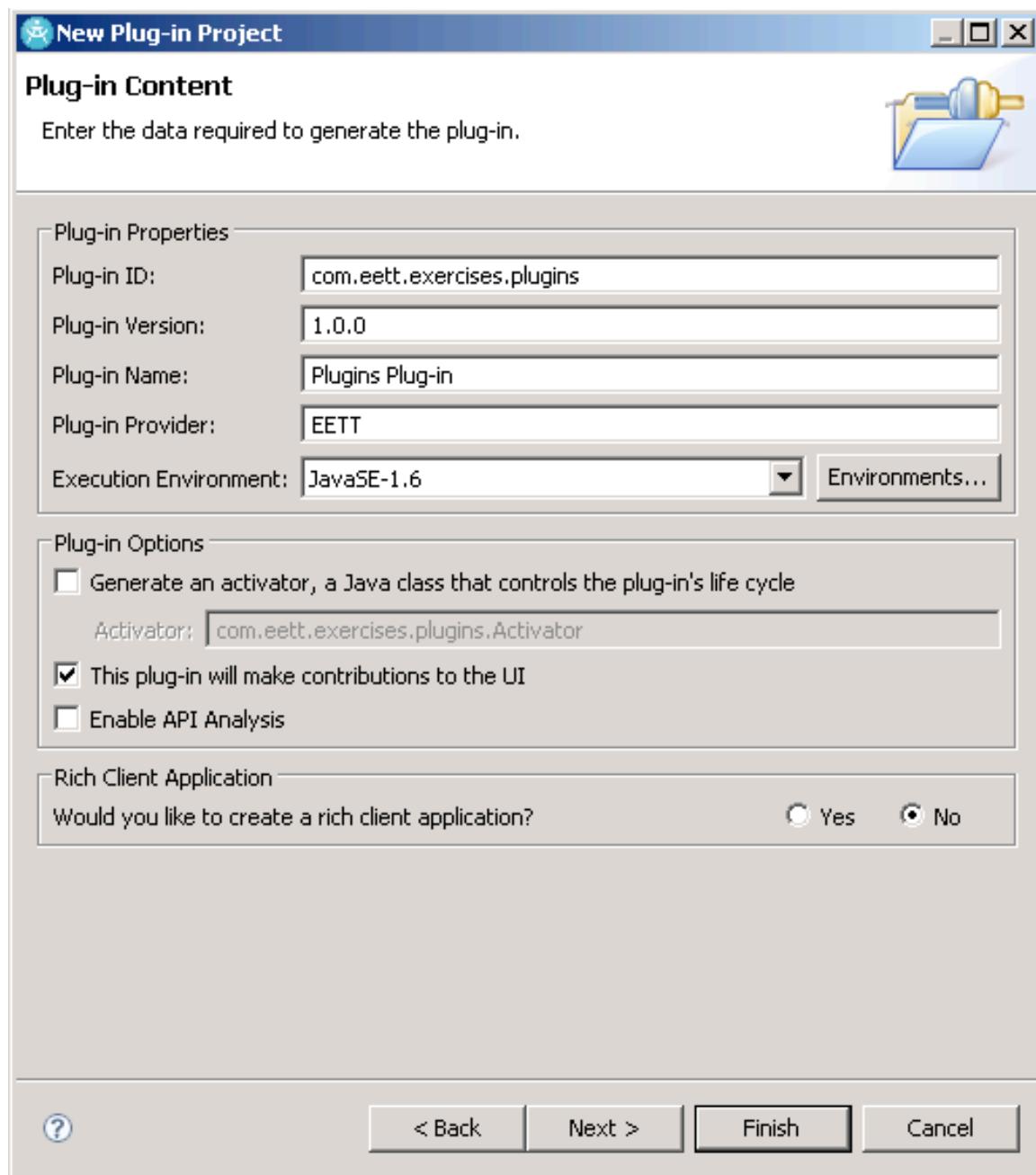


Click the Next button.

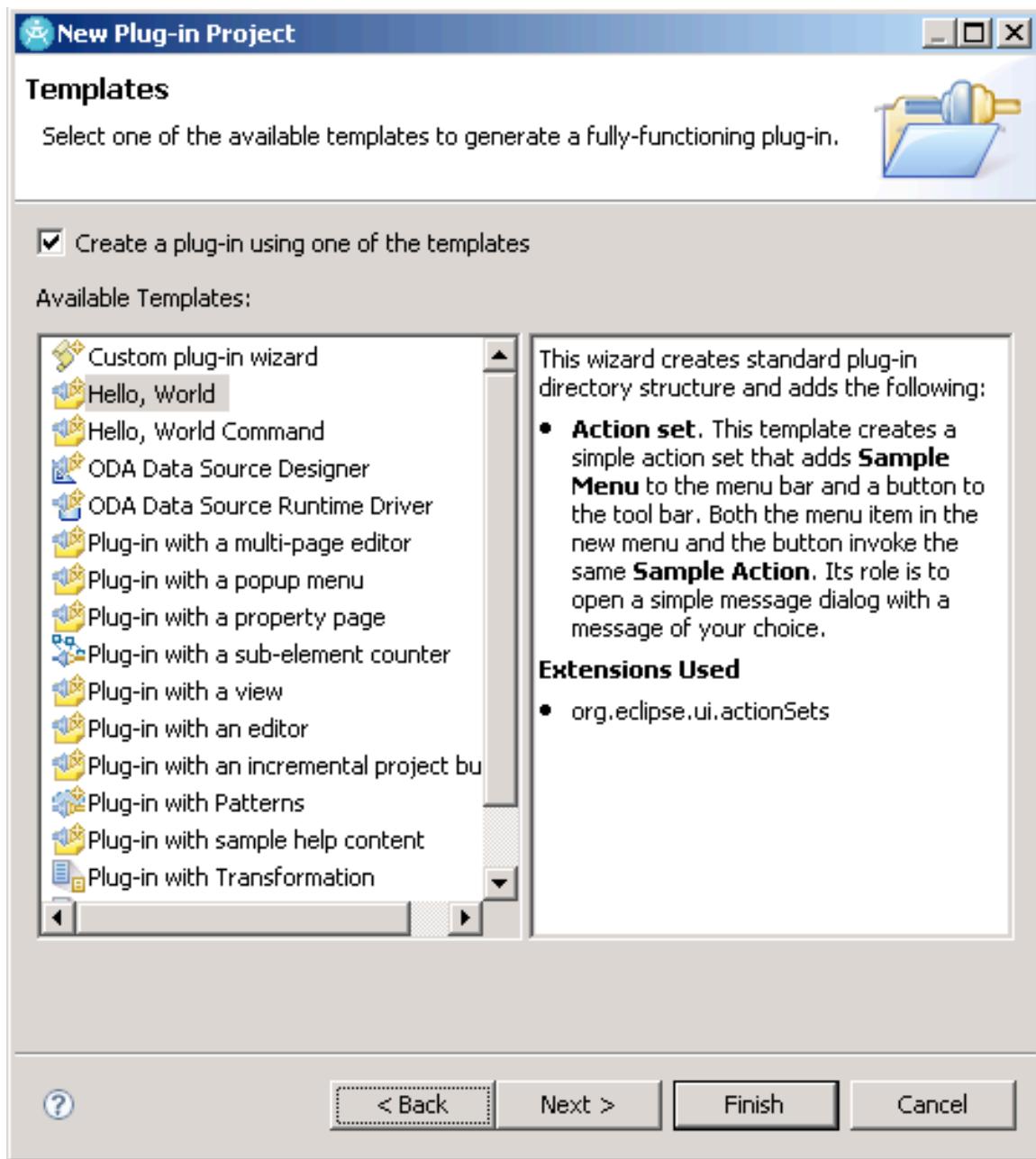
Enter the name of the project; com.eett.exercises.plugins. Keep Use default location and Create a Java project checked. Retain the defaults for the other properties. Click on Next >.



Specify the Plug-in Content. Keep the Plug-in ID, Plug-in Version, Plug-in Provider and Execution Environment defaults. Change the Plug-in Name to EETT Plug-in Exercises. Uncheck Generate an activator and make sure that This plug-in will make contributions to the UI is checked. Click the Next > button.

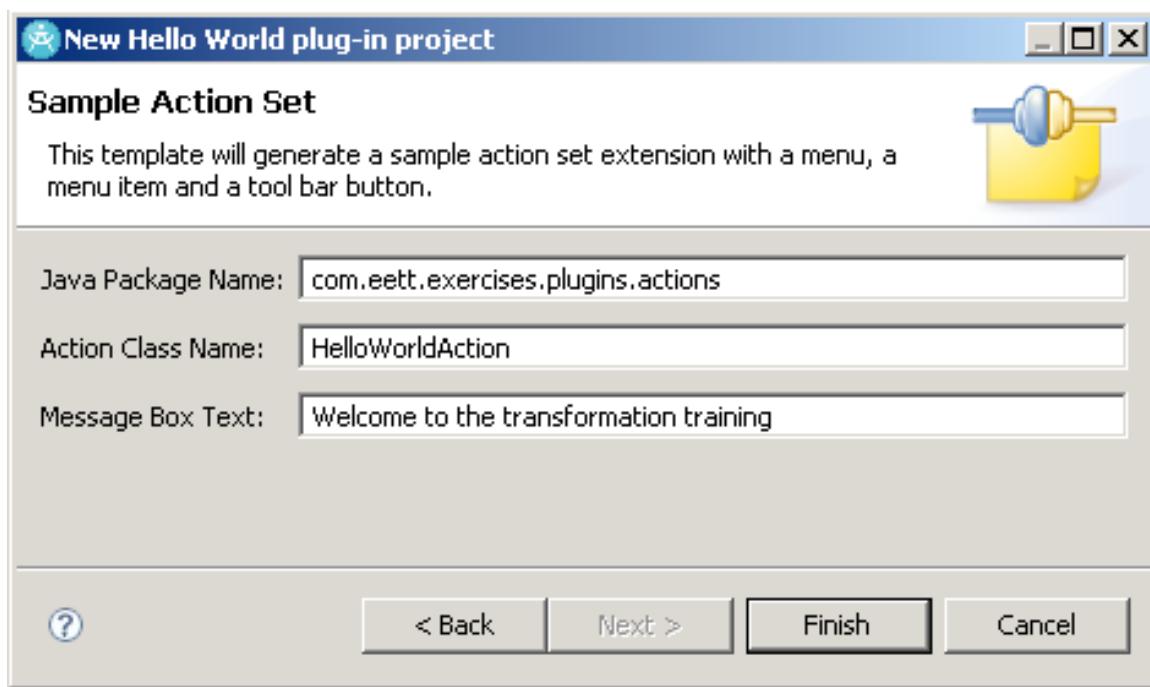


On the Templates page check the 'Create a plug-in using one of the templates' and select the 'Hello, World' option. Click Next >.



By using a Template most of the code necessary to create a menu and an Action to handle selecting the menu is generated for you.

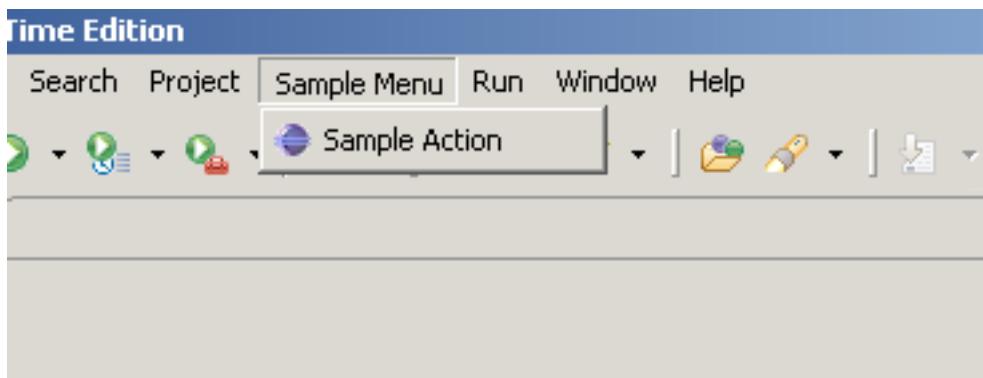
In the Sample Action Set page you can configure the code that will be generated. Change the Action Class Name to HelloWorldAction and the Message Box Text to Welcome to the transformation training. Click Finish.

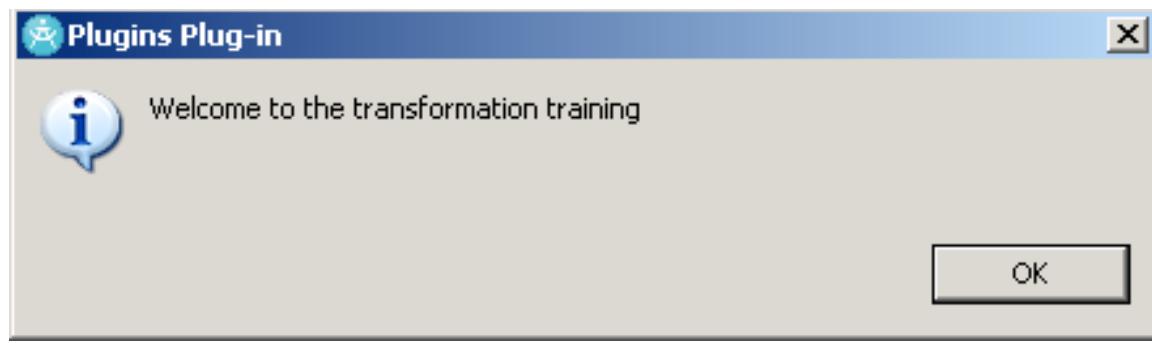


When prompted to switch to the Plug-in Development perspective click Yes.

1.3 Test the plug-in

At this point we can launch a Runtime Workbench and see our plug-in in action. To view the default that is generated select Run As → Eclipse Application from the project's context menu.



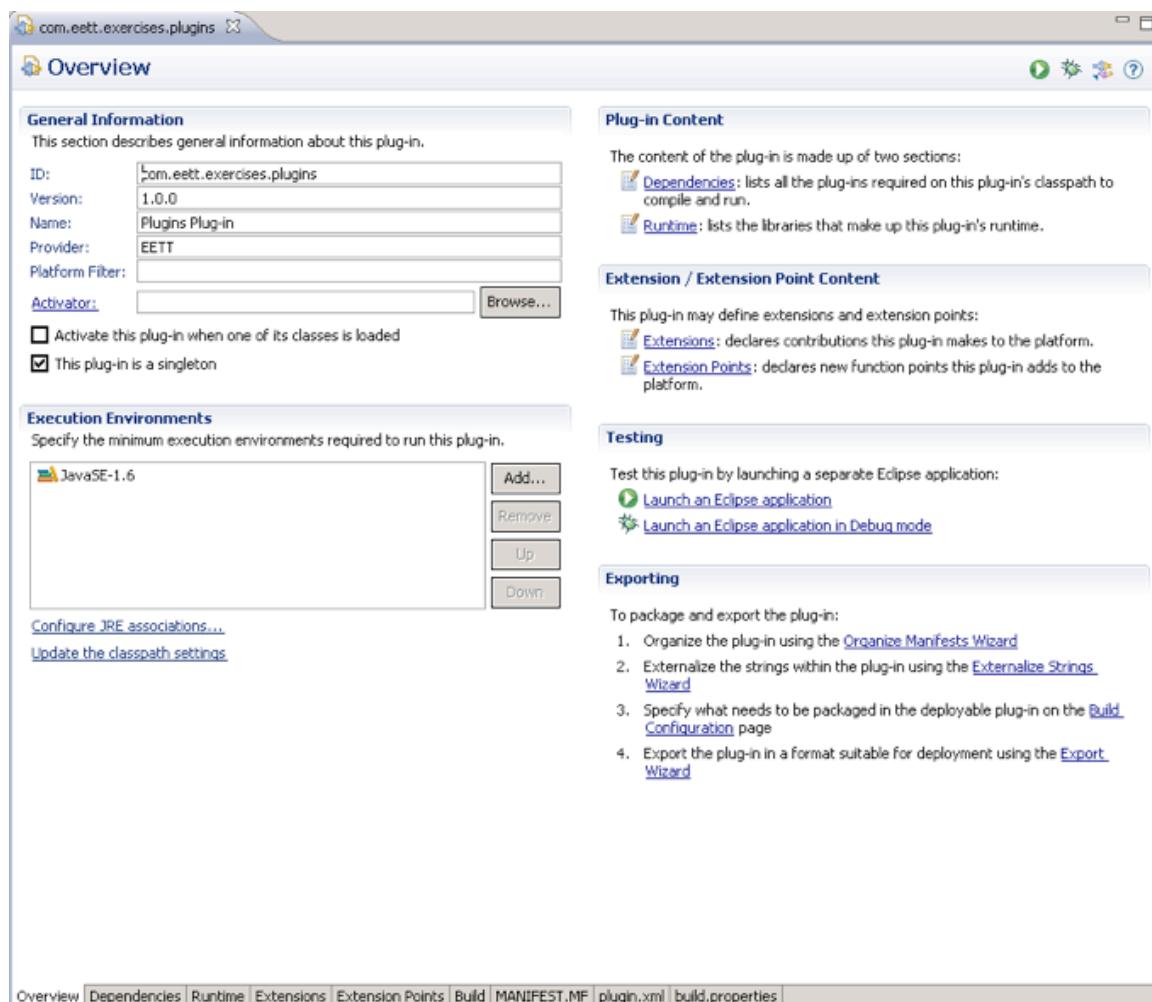


This is an Eclipse version of the classic Hello World application. We will explore in more detail and augment the default that was created for us.

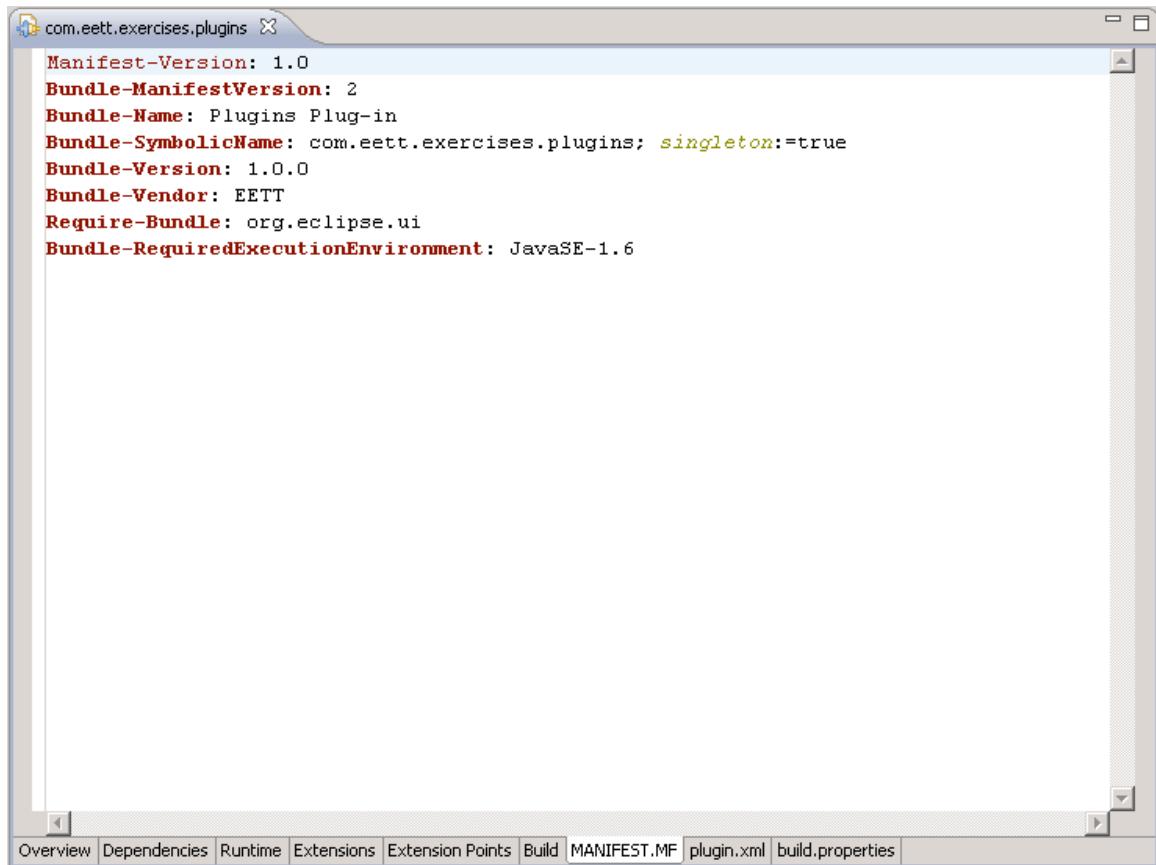
1.4 Define the plug-in manifest files

The project wizard that we used in the previous steps created the plug-in manifest files for us. The plugin.xml contains our extensions to the Eclipse workbench and META-INF/MANIFEST.MF describes the plug-in and its dependencies.

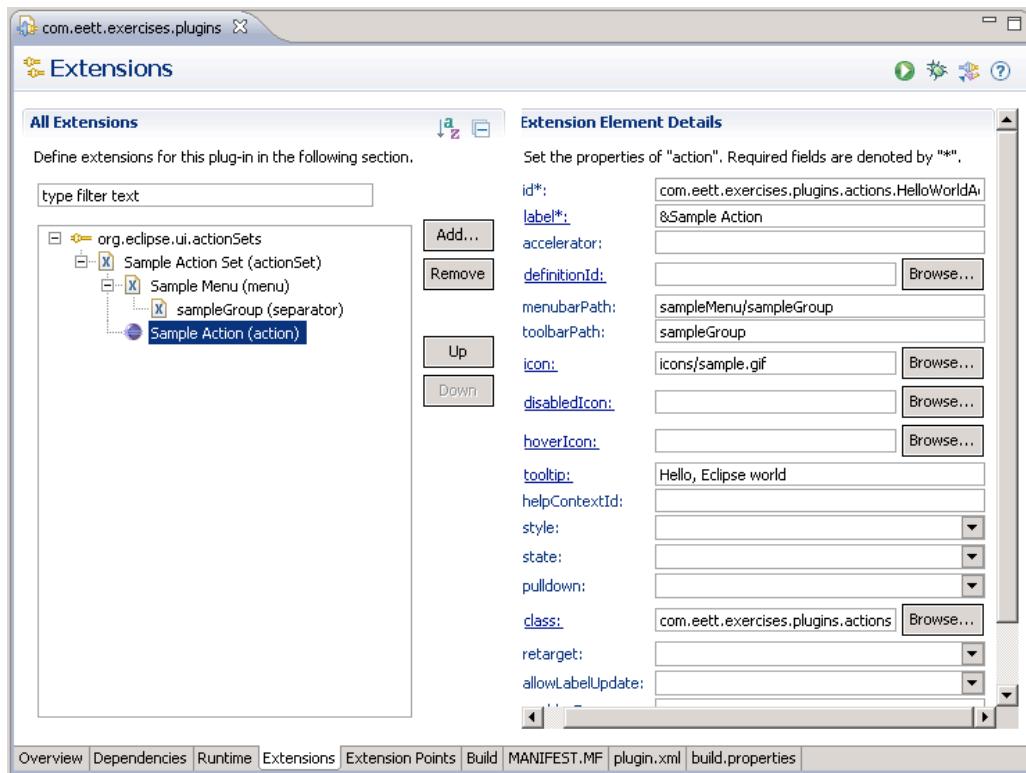
Open the MANIFEST.MF file in the Plug-in Manifest Editor and explore the values using the Overview, Dependencies and Runtime pages.



You can also change the MANIFEST.MF manually using the MANIFEST.MF page in the editor. We do not need to change this file at this point.



Open the plugin.xml using the Plug-in Manifest Editor. The current file should look like the following. We will change some of these values in order to customize the defaults that were generated by the wizard.



(xml content on next page)

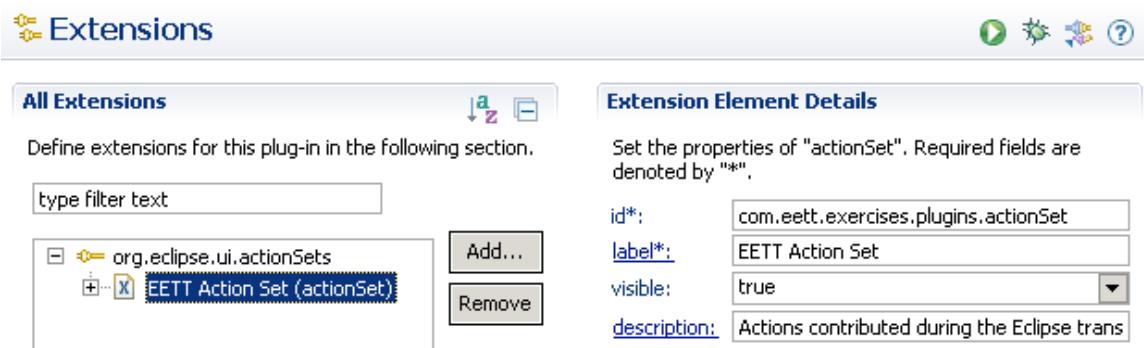
```

<plugin>
  <extension
    point="org.eclipse.ui.actionSets">
    <actionSet
      label="Sample Action Set"
      visible="true"
      id="com.eett.exercises.plugins.actionSet">
      <menu
        label="Sample & Menu"
        id="sampleMenu">
        <separator
          name="sampleGroup">
        </separator>
      </menu>
      <action
        label="& Sample Action"
        icon="icons/sample.gif"
        class="com.eett.exercises.plugins.actions.HelloWorldAction"
        tooltip="Hello, Eclipse world"
        menuBarPath="sampleMenu/sampleGroup"
        toolbarPath="sampleGroup"
        id="com.eett.exercises.plugins.actions.HelloWorldAction">
      </action>
    </actionSet>
  </extension>
</plugin>

```

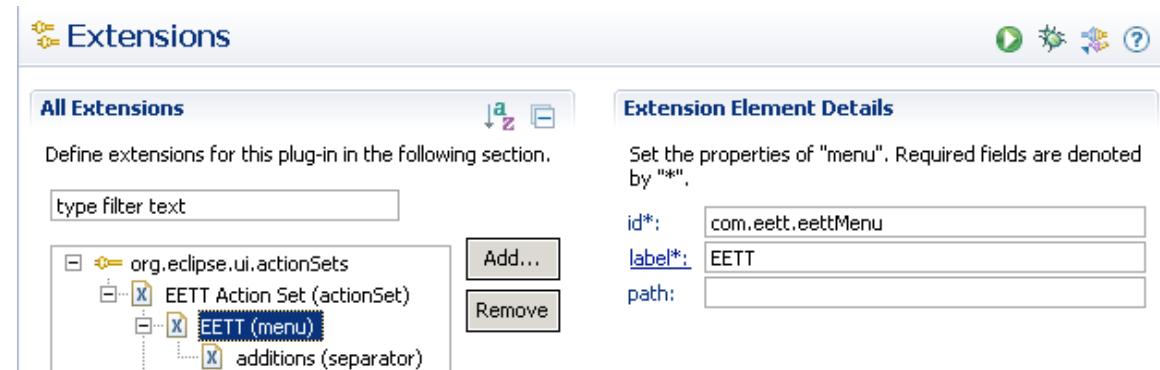
When we ran the plug-in that was generated the menu item names were not very descriptive or customized. We will change the names of the menus from Sample... to something more specific to our exercises. We will reuse the main menu item that is contributed to the workbench so we need to ensure that its name is applicable and that its id is intuitive.

Start by changing the label of the actionSet to EETT Action Set, which is the label used by the Workbench to represent this action set to the user. We will also set the description field to 'Actions contributed during the Eclipse transformation training'. This change will show up in the perspective configuration where you can show and hide action sets.

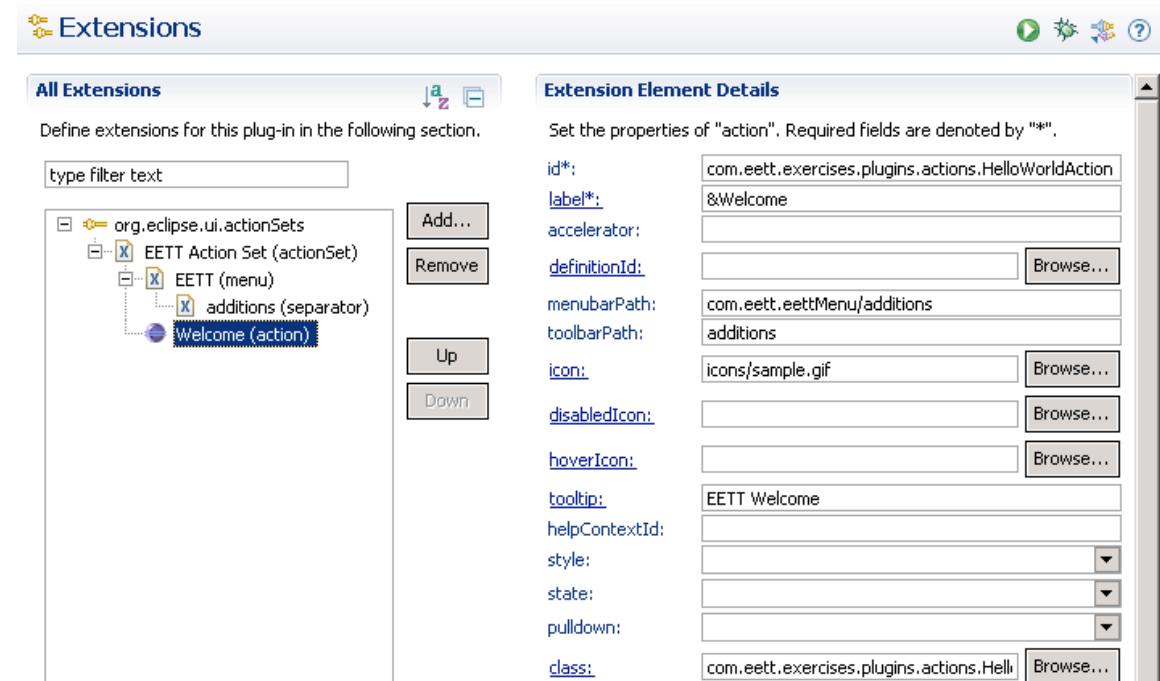


Change the menu label to EETT, which will be the label used in the main menu of the workbench. Also, change the id of the menu to com.eett.eettMenu, this id is used to refer to the menu to add actions.

Change the menu's separator to have the name additions rather than sampleGroup. This separator is used as part of the path to add actions to the EETT menu.

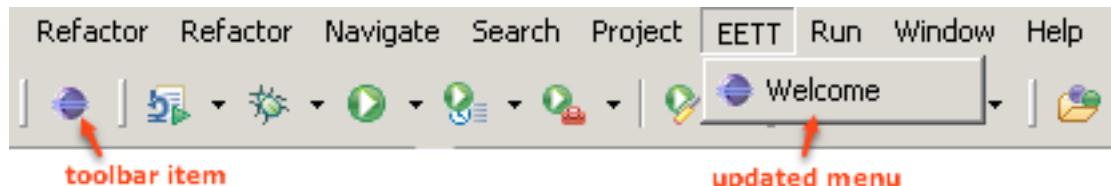


We can now update the action to reflect the changes we have made to the menu and action set. We will start by changing the label to Welcome to better reflect what the action does. In the previous steps we changed the id of the menu and the name of its separator so we have to update the path menubarPath to reflect this; change it to com.eett.eettMenu/additions. We can also change the tooltip to EETT Welcome.

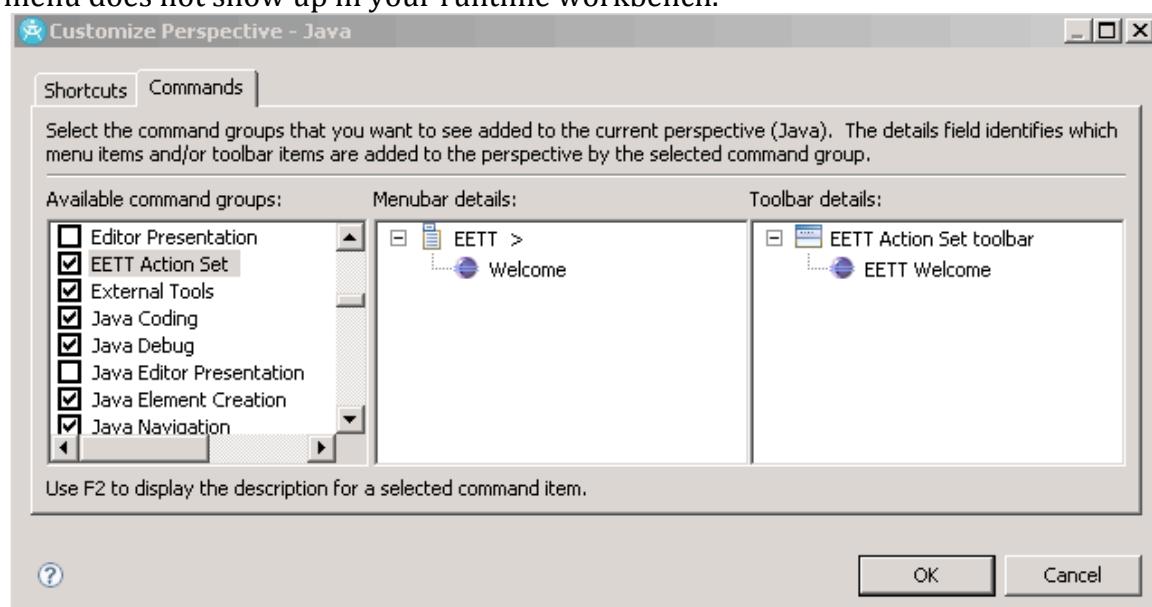


The plugin.xml file should now resemble

```
<plugin>
  <extension
    point="org.eclipse.ui.actionSets">
    <actionSet
      label="EETT Action Set"
      visible="true"
      id="com.eett.exercises.plugins.actionSet">
      <menu
        label="EETT"
        id="com.eett.eettMenu">
        <separator
          name="additions">
        </separator>
      </menu>
      <action
        label="&Welcome"
        icon="icons/sample.gif"
        class="com.eett.exercises.plugins.actions.HelloWorldAction"
        tooltip="EETT Welcome"
        menuBarPath="com.eett.eettMenu/additions"
        toolbarPath="additions"
        id="com.eett.exercises.plugins.actions.HelloWorldAction">
      </action>
    </actionSet>
  </extension>
</plugin>
```



To view our action set name changes we can open the Customize Perspective dialog; Window → Customize Perspective... This is also a good place to look if the menu does not show up in your runtime workbench.



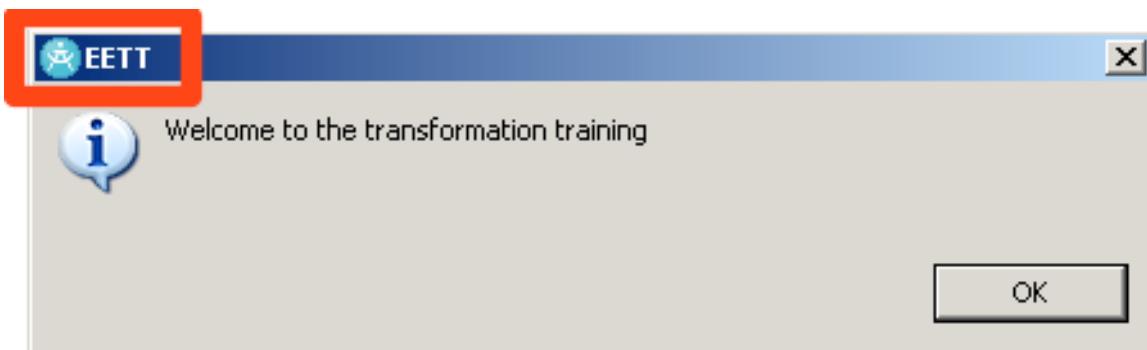
1.5 Implement the action delegate

This part of the exercise will modify the default action delegate that was generated by the wizard. Switch to the Java perspective and open the `HelloWorldAction.java` file. The code is documented, take a few minutes to familiarize yourself with it.

We are going to change the title for the dialog box. In the run method change the second parameter of `MessageDialog.openInformation` to "EETT". *Note that it is best practice to externalize strings such as this one so that they can be localized.* Your run method should now resemble.

```
public void run(IAction action) {  
    MessageDialog.openInformation(  
        window.getShell(),  
        "EETT",  
        "Welcome to the transformation training");  
}
```

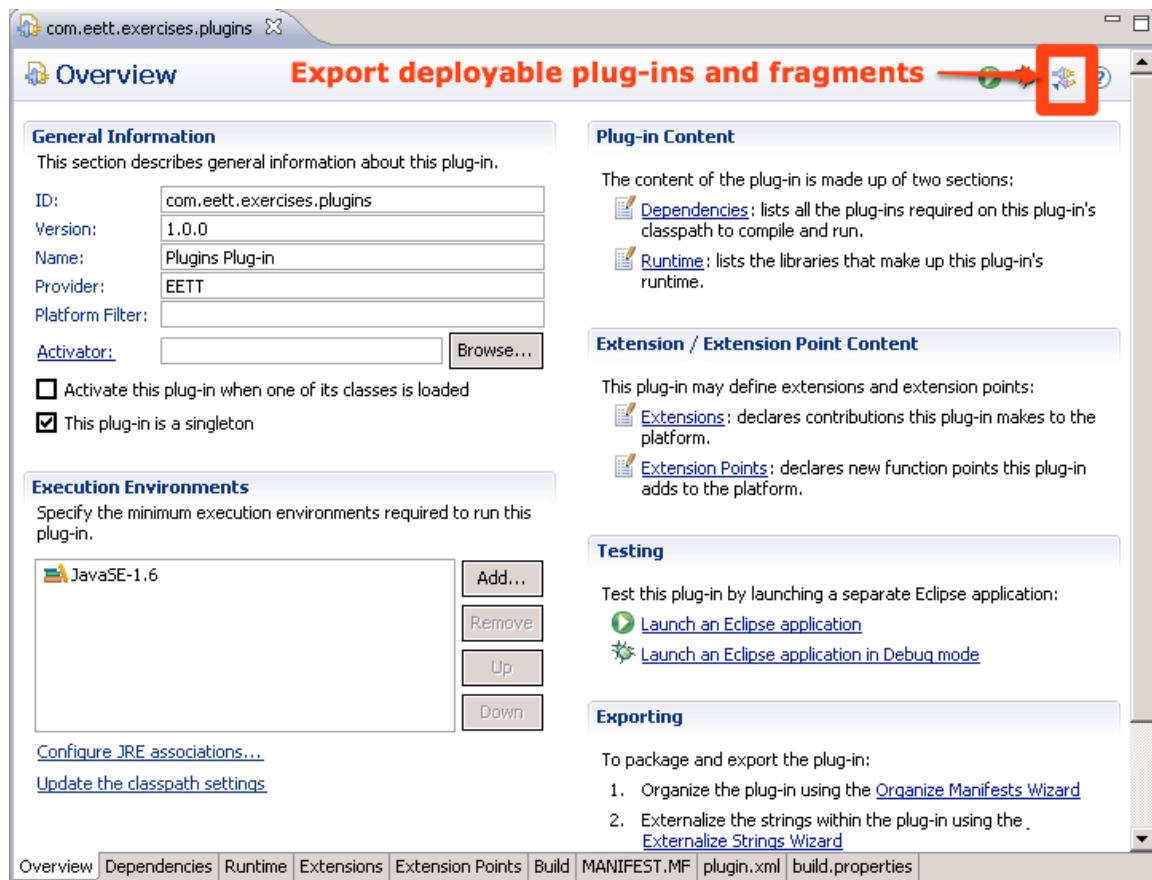
Save your changes and launch the runtime workbench to ensure that your changes were made. It should now look like.



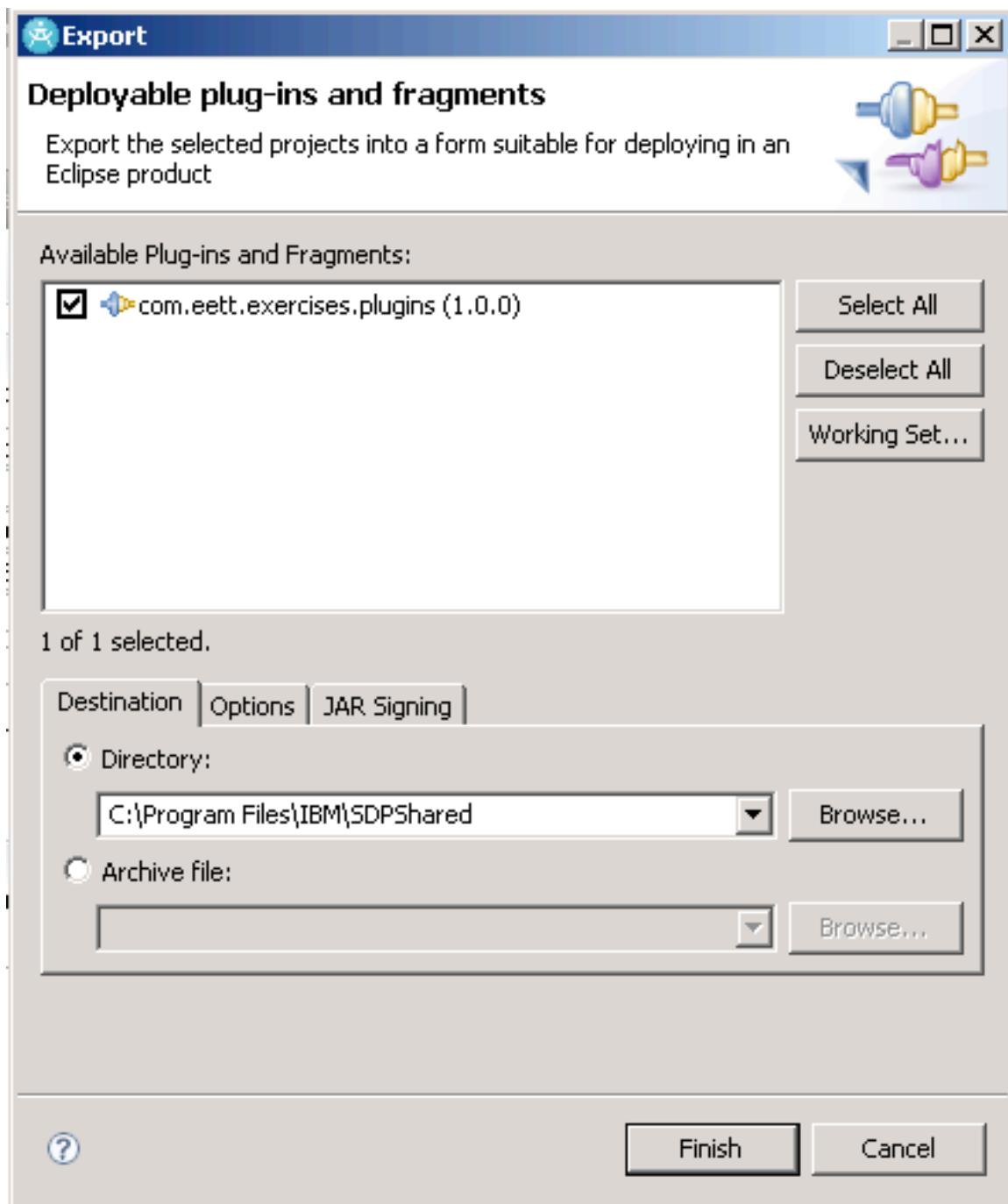
1.6 Publish the plug-in

We now have a plug-in and we can publish it to our installation so that it is available without running the workbench.

Open the Plug-in Manifest Editor for our plug-in and switch to the Overview page. In the top right-hand corner is a button to export the plug-in. Click on the button.



In the dialog make sure your plug-in is selected in the list of Available Plug-ins and specify Destination Directory as a dropdown directory in your Eclipse install. With RSA-RTE this will likely be SDP directory of your install. Leave the other settings to their defaults.



Click the Finish button and restart your workbench. When the workbench restarts it will have the EETT menu just like in the runtime workbench. Test to make sure that it behaves the same.

Through out the course we will add additional actions to the EETT menu.

2 Eclipse Modeling Framework Exercises

In this exercise a metamodel for the structural part of ROOM will be developed in order to gain a better understanding of EMF. The exercise will start by creating an Ecore model of the ROOM language. This model is used to create dynamic instances and to generate the code for the metamodel and a default tree based editor. There will be activities that show how to work with the generated code in order to implement the derived attributes, implement operations and override code that is generated by EMF.

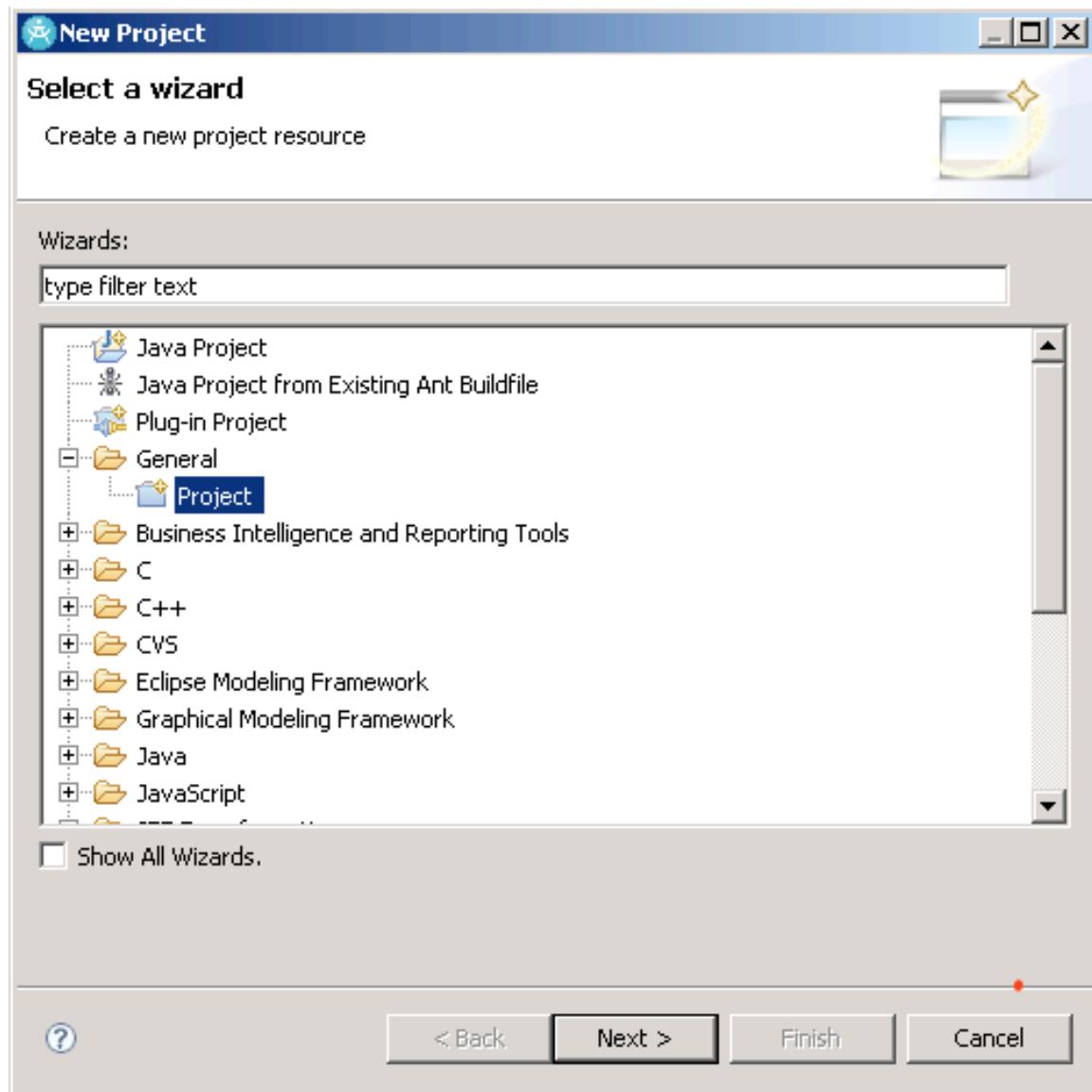
The last two activities of the exercise will practice working with transactions and querying a model.

2.1 Building an Ecore Model

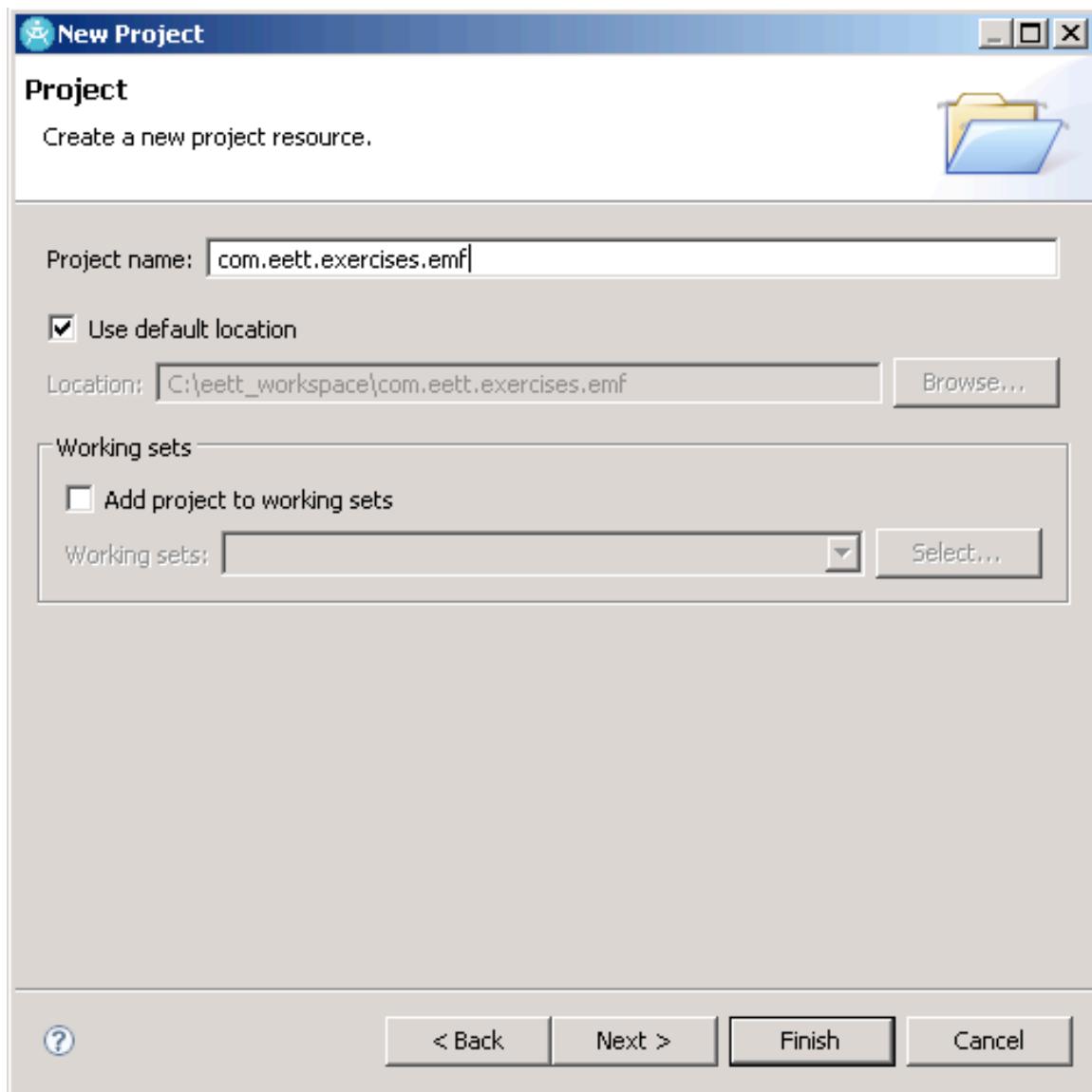
Create a new project basic project.

From the main menu select File → New → Project...

In the New Project Wizard select Project under the General group, and click the Next > button.



Provide a Project Name, com.eett.exercises.emf and click the Finish button

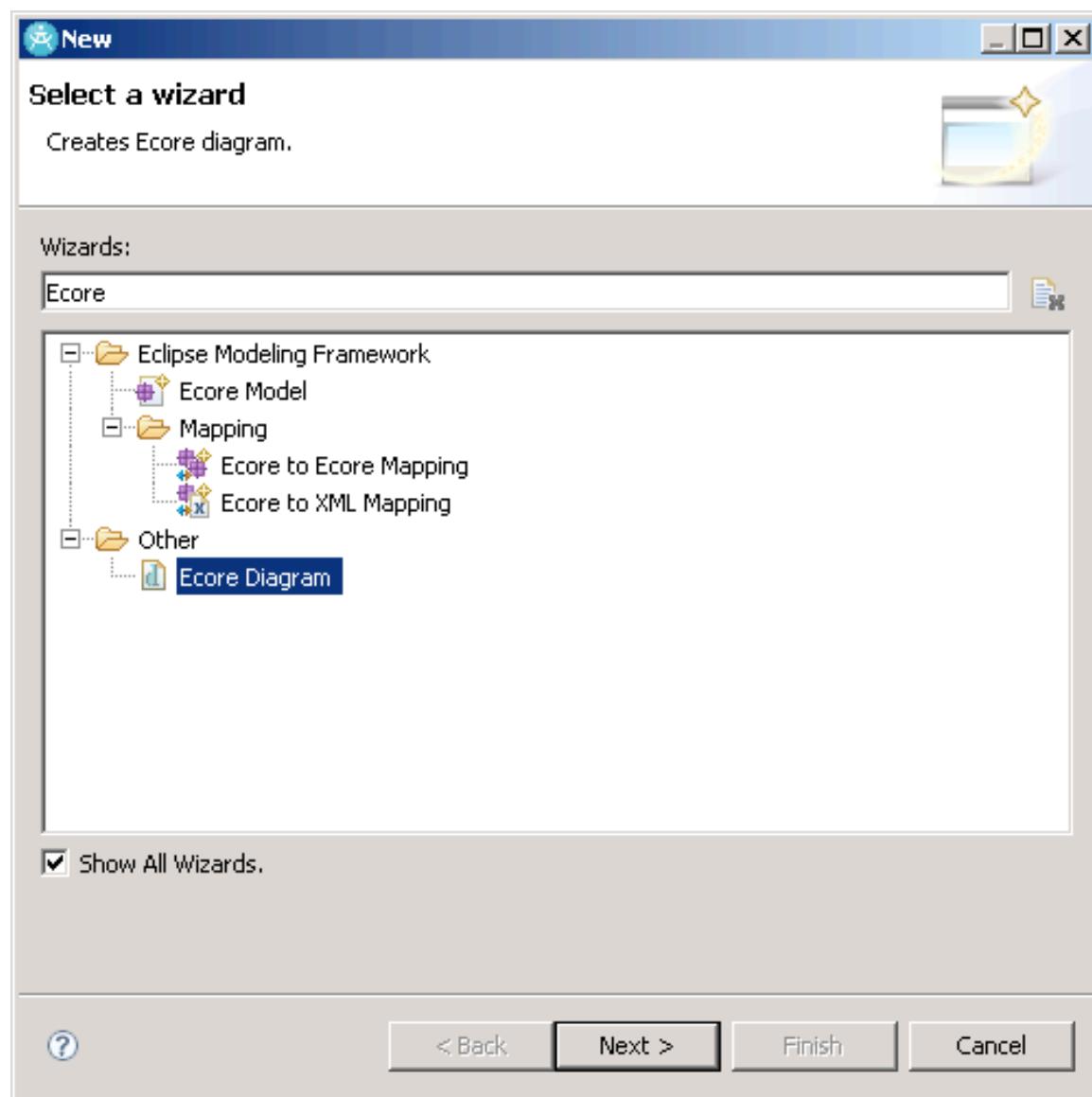


Create a new EMF model.

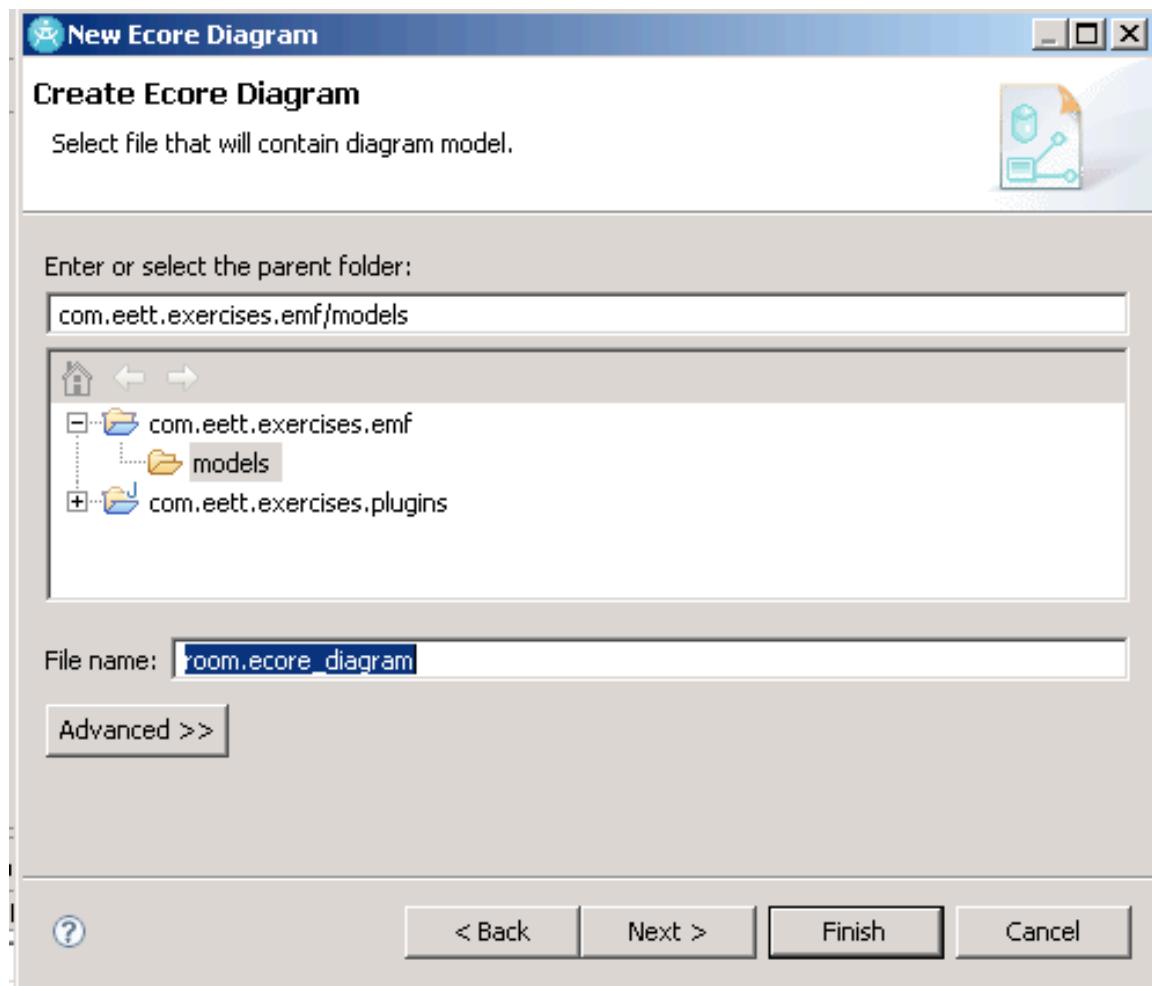
Add a models folder to the com.eett.exercises.emf project, to store our model artifacts

Select the models folder and choose File → New → Other...

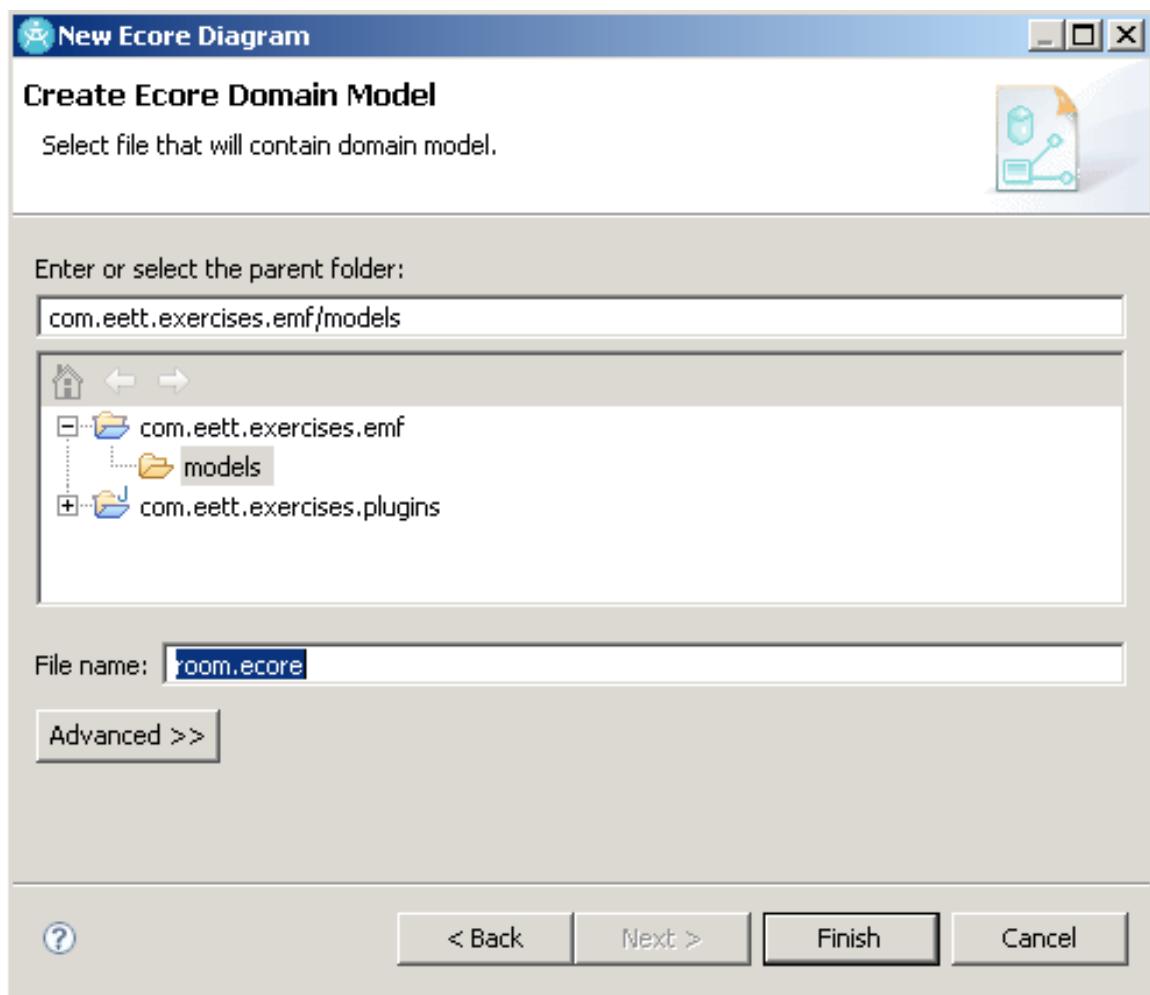
In the New wizard select Ecore Diagram from Other (you may have to check Show All Wizards) and click the Next > button



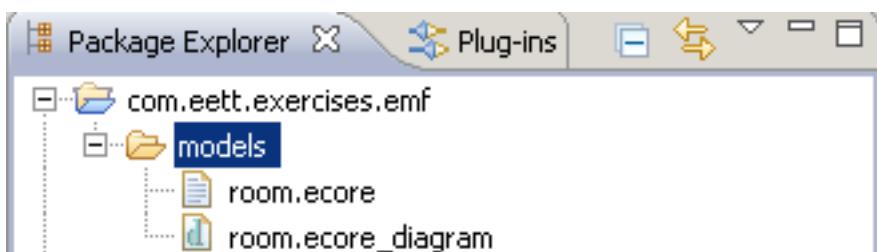
Set the name of the Ecore Diagram to room.ecore_diagram and click the Next > button



Set the name of the Ecore model to room.ecore and click the Finish button



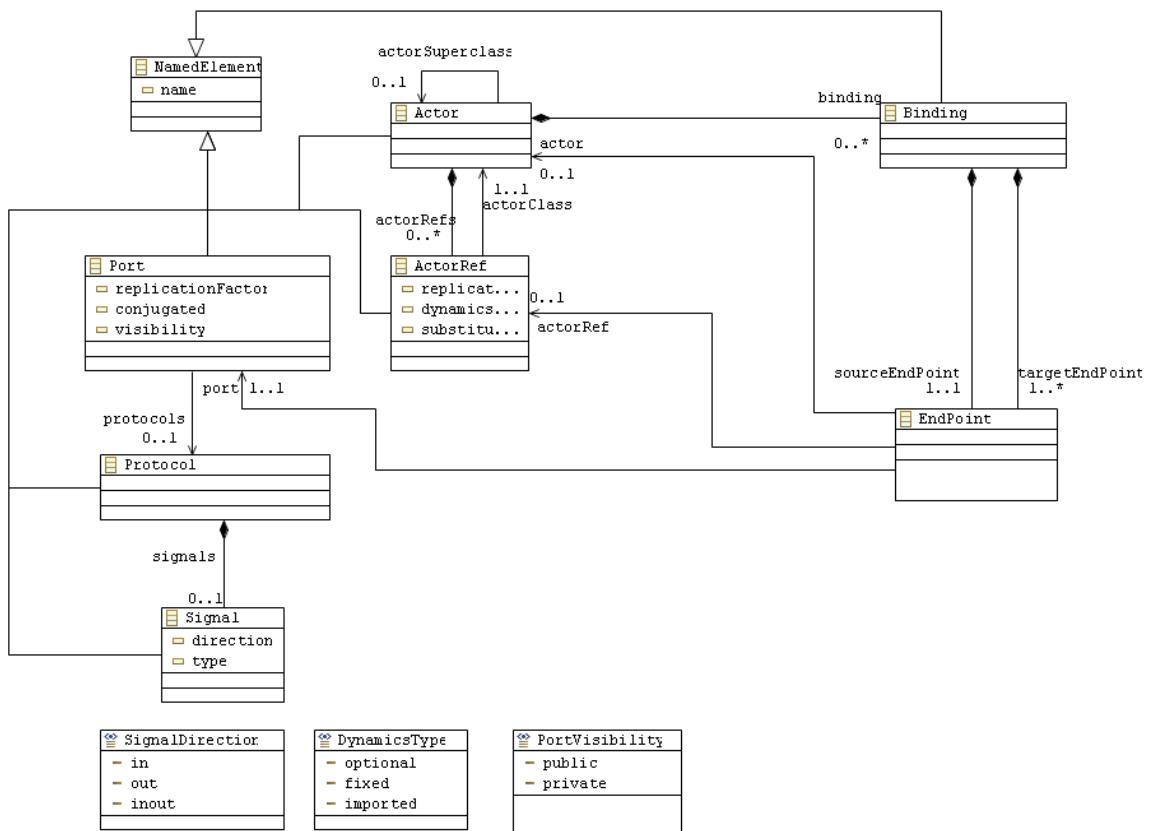
Your project should now look like



Open the Ecore diagram and set the properties in the property grid to the following, this configures the properties of the root ePackage. To access the properties right-click in the diagram editor and select Show Properties View.

ROOM	
Core	Property
Rulers & Grid	Name <input type="text" value="ROOM"/>
Appearance	Ns Prefix <input type="text" value="room"/>
	Ns URI <input type="text" value="http://www.eett.com/room/2008"/>

In the diagram editor build a model that looks like the following



NamedElement (Abstract EClass)

Attributes

Name	Type	Lower bound	Upper bound
name	EString	0	1

References

Name	Type	Containment	Opposite	Lower bound	Upper bound

Actor (EClass) extends NamedElement

Attributes

Name	Type	Lower bound	Upper bound

References

Name	Type	Containment	Opposite	Lower bound	Upper bound
actorSuperClass	Actor	false		0	1
bindings	Binding	true	actor	0	-1
actorRefs	ActorRef	true	container	0	-1
ports	Port	true		0	-1

ActorRef (EClass)

Attributes

Name	Type	Lower bound	Upper bound	Default
replicationFactor	EInt	0	1	
dynamicsType	DynamicsType	0	1	fixed
substitutable	EBoolean	0	1	false

References

Name	Type	Containment	Opposite	Lower bound	Upper bound
actorClass	Actor	false		1	1
container	Actor	false	actorRefs	1	1

Port (EClass)

Attributes

Name	Type	Lower bound	Upper bound	Default
replicationFactor	EInt	0	1	1
conjugated	EBoolean	0	1	false
visibility	PortVisibility	0	1	public

References

Name	Type	Containment	Opposite	Lower bound	Upper bound
protocol	Protocol	false		0	1

Protocol (EClass)

Attributes

Name	Type	Lower bound	Upper bound

References

Name	Type	Containment	Opposite	Lower bound	Upper bound
signals	Signal	true		0	-1

Signal (EClass)

Attributes

Name	Type	Lower bound	Upper bound	Default
direction	SignalDirection	0	1	in
type	EString	0	1	

References

Name	Type	Containment	Opposite	Lower bound	Upper bound

SignalDirection (EEnum)

Name	Value
IN	0
OUT	1
INOUT	2

DynamicsType (EEnum)

Name	Type
optional	0
fixed	1
imported	2

PortVisibility(EEnum)

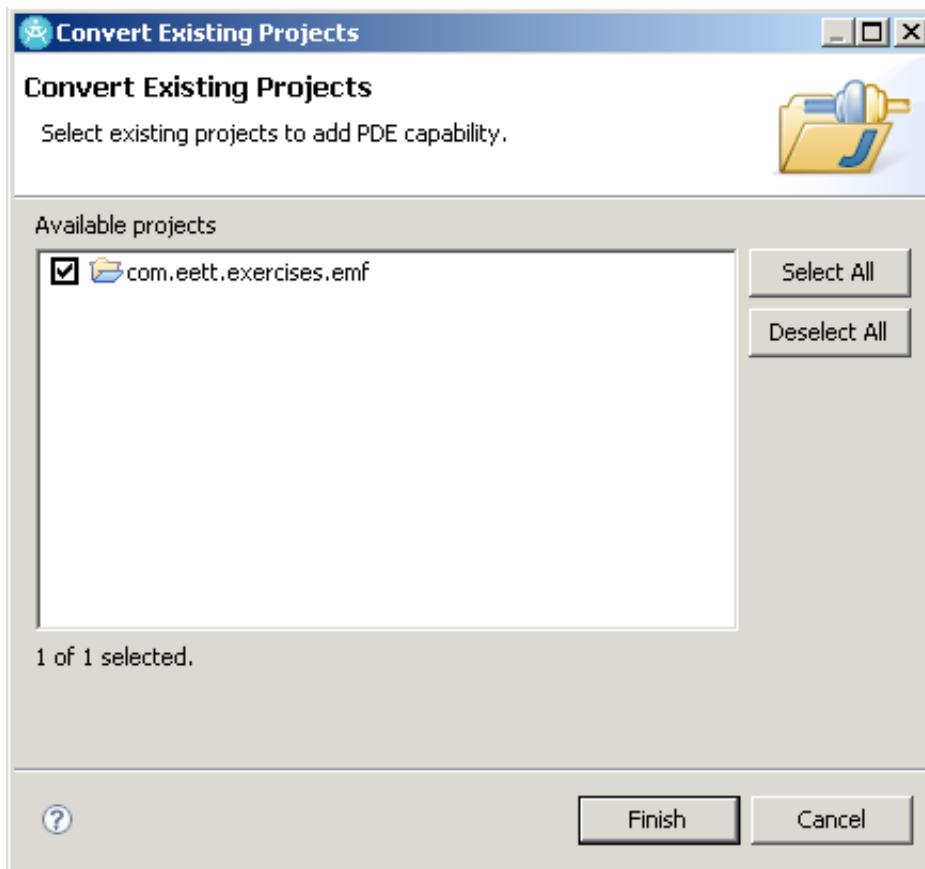
Name	Type
public	0
private	1

When finished creating the model validate it to ensure that there are no errors. To validate select the root package and Sample Ecore Editor → Validate from the main menu.

We have built a metamodel for a subset of the ROOM language, we can now look at creating the generator model and creating models using the metamodel.

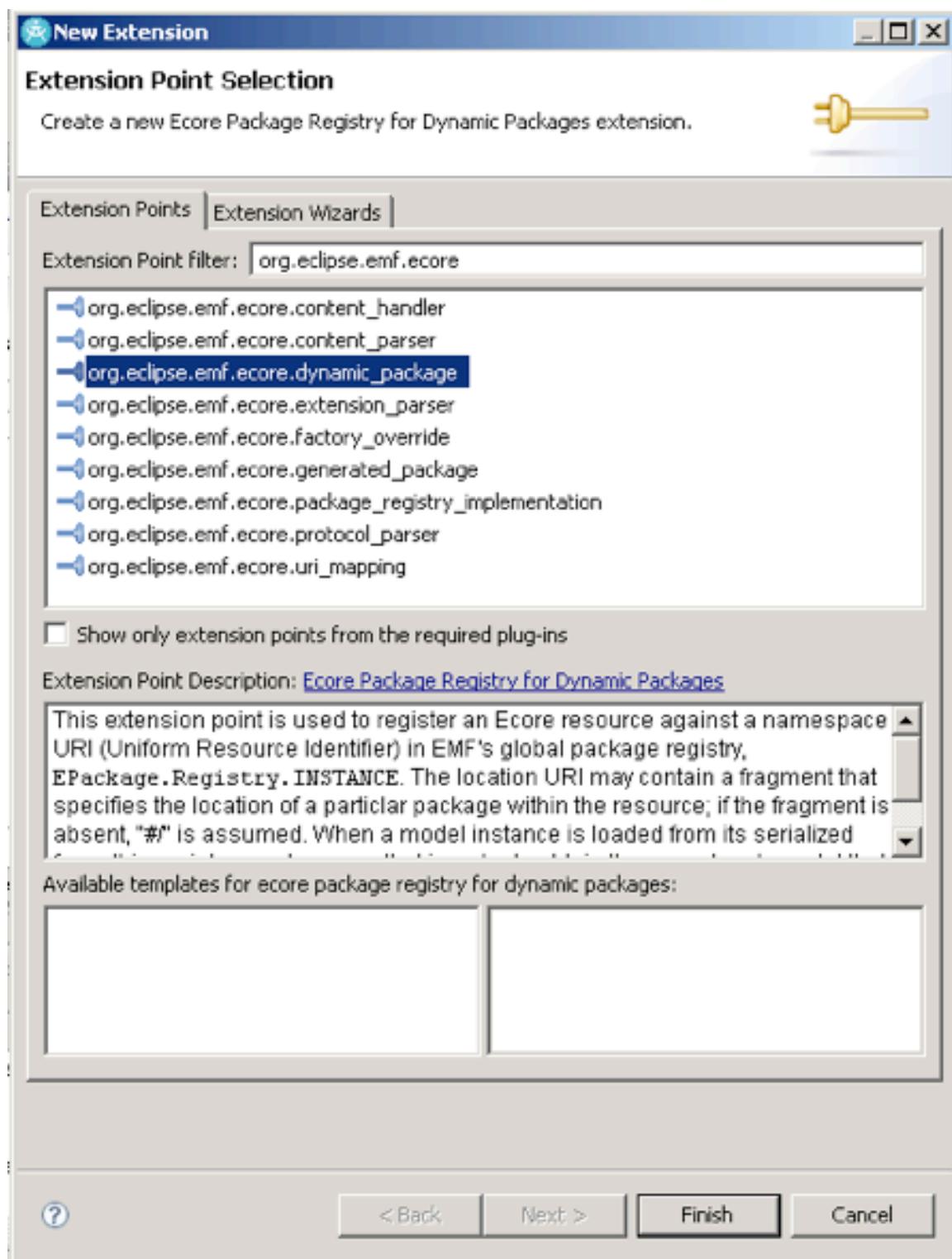
Publish the Model so that it can be used dynamically

We will first convert our com.eett.exercises.emf project to a plugin project. Select the project and choose PDE Tools → Convert Projects to Plug-in Projects... Make sure that com.eett.exercises.emf is checked and click Finish

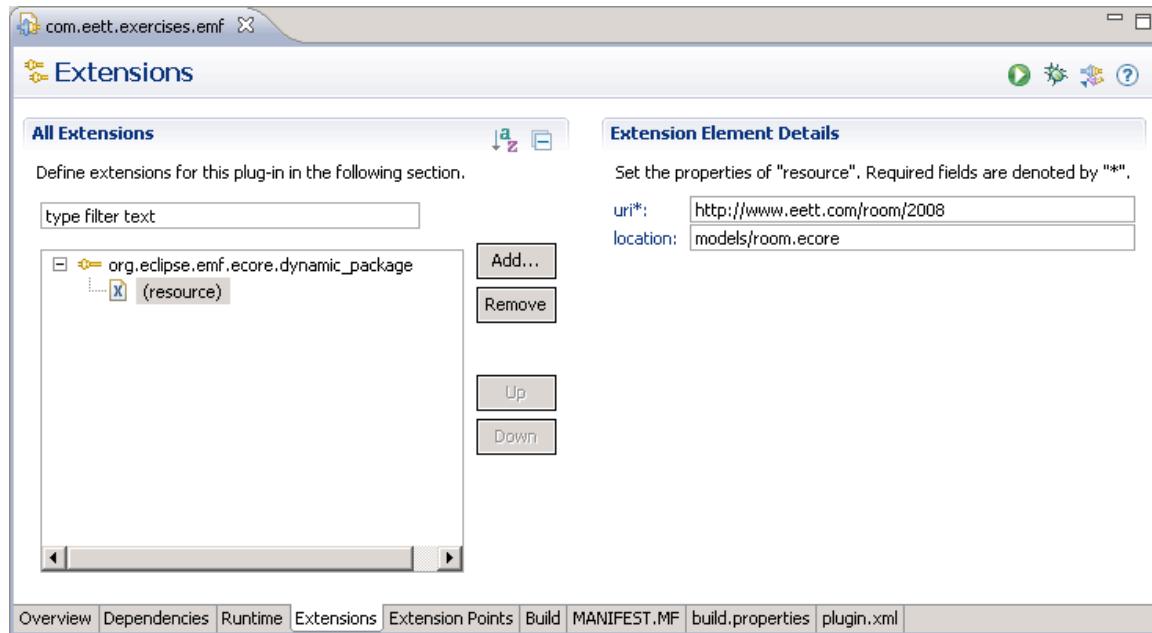


Open the META-INF/MANIFEST.MF in the Plug-in Manifest Editor and change the Name of the plug-in to be EETT EMF Models and the provider to EETT. Save your changes and switch to the Extensions page of the editor.

On the Extensions page click the Add... button. In the Extension Point Selection dialog make sure that Show only extension points from the required plug-ins is unchecked. Then select the com.eclipse.emf.ecore.dynamic_package extension point. Click the Finish button.



Set the extension points URI field to be the NS URI of the room.ecore model and the location to the location of the room.ecore model. Save your changes.



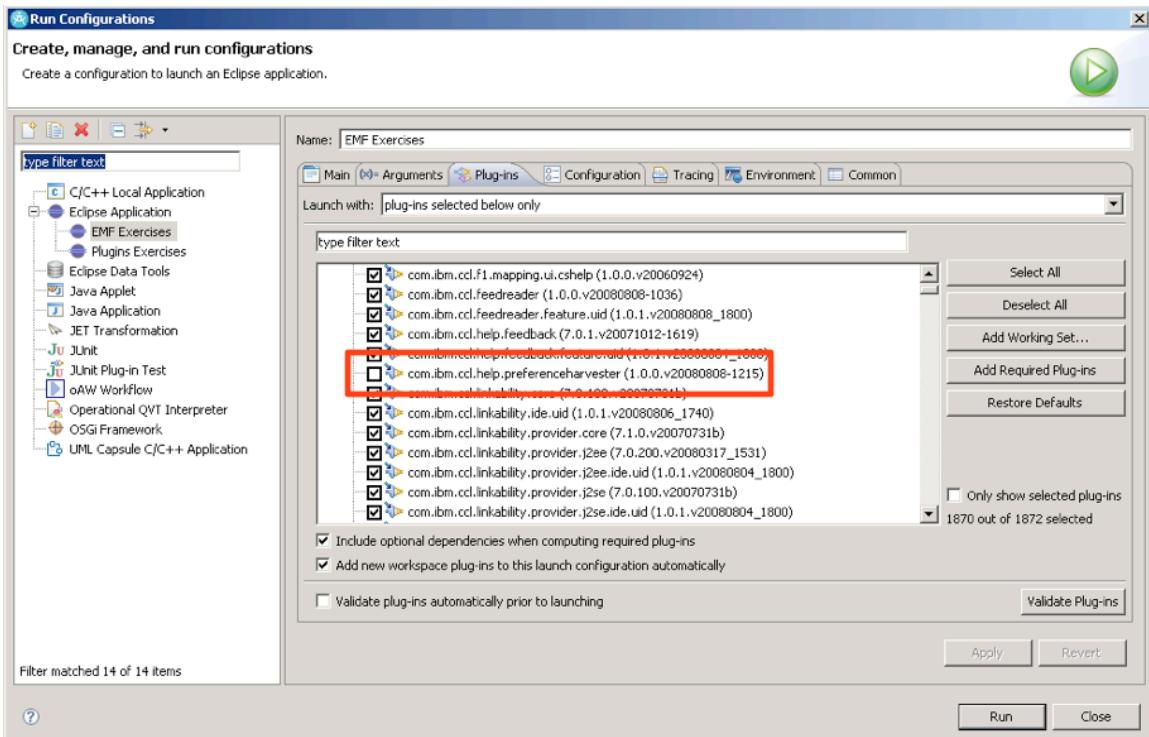
2.2 Creating a Dynamic Instance

Once the metamodel has been created it is possible to create instances of the ROOM metamodel that was defined in the previous exercises. The reflective API of EMF allows a generic editor to be provided to create what are called dynamic instances of the model. This is as opposed to generating the implementation of the metamodel and using either the generated editor or building a custom editor.

In the EMF Ecore editor select the Actor element, and select Create Dynamic Instance from the context menu.

Open the Run Configuration dialog (Run → Run Configurations...). Create a new Eclipse Application configuration, EMF Exercises, and switch to the Plug-ins page. Change the Launch with field to ‘plug-ins selected below only’.

In the list of plug-ins uncheck com.ibm.ccl.help.preferenceharvester, and click Apply followed by Run.



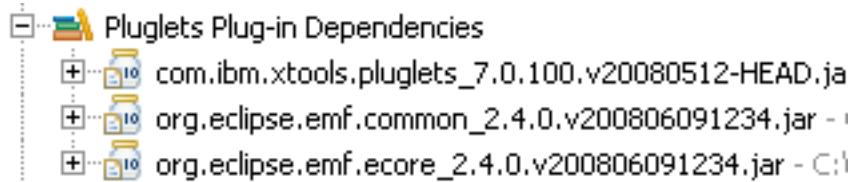
This should launch a runtime workbench, which we will use to programmatically work with our ROOM metamodel. To do this we will use Pluglets, which are an IBM Modeling Platform specific technology for extending the workbench. A pluglet provides script like capabilities, to perform tasks without having to build full plug-ins.

In the runtime workbench create a new Pluglets project, named com.eett.exercises.emf.plugin. Don't worry about the other settings in the project wizard.

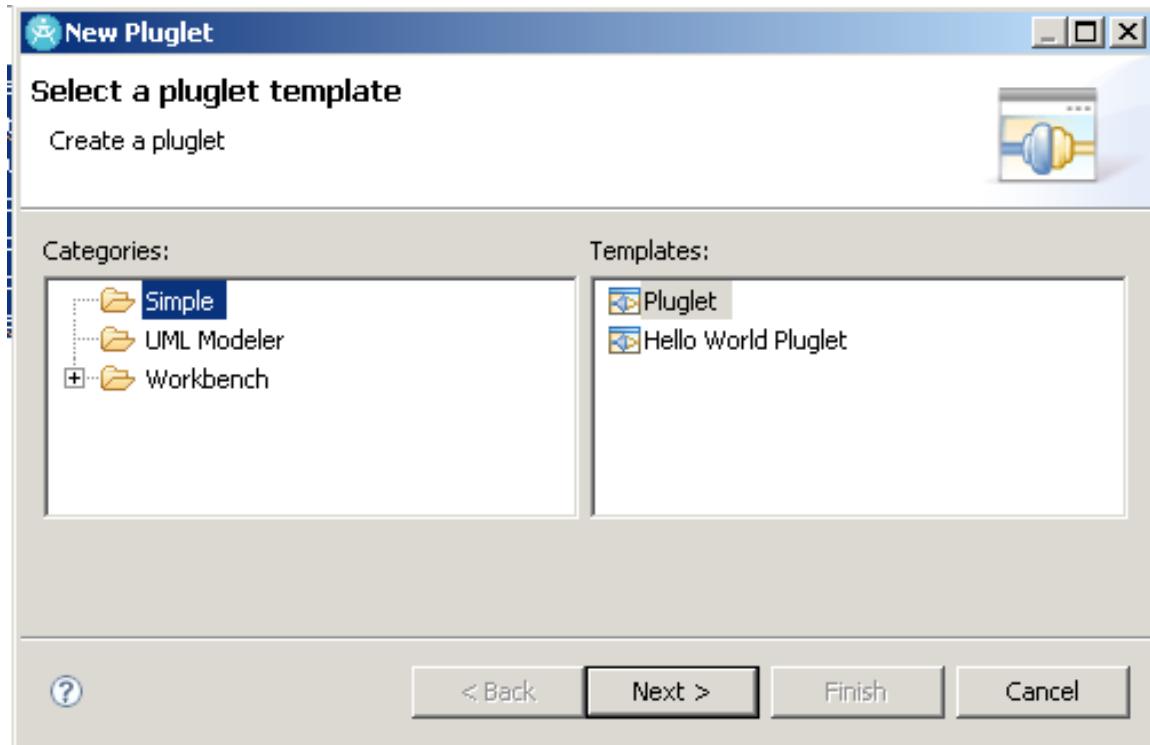
The project will contain a pluglet.xml file which is used to capture dependencies on plug-ins. We going to use EMF in this exercise, therefore we need to add a dependency on org.eclipse.emf.ecore. Open pluglet.xml with a text editor and modify it to look like the following and save the file

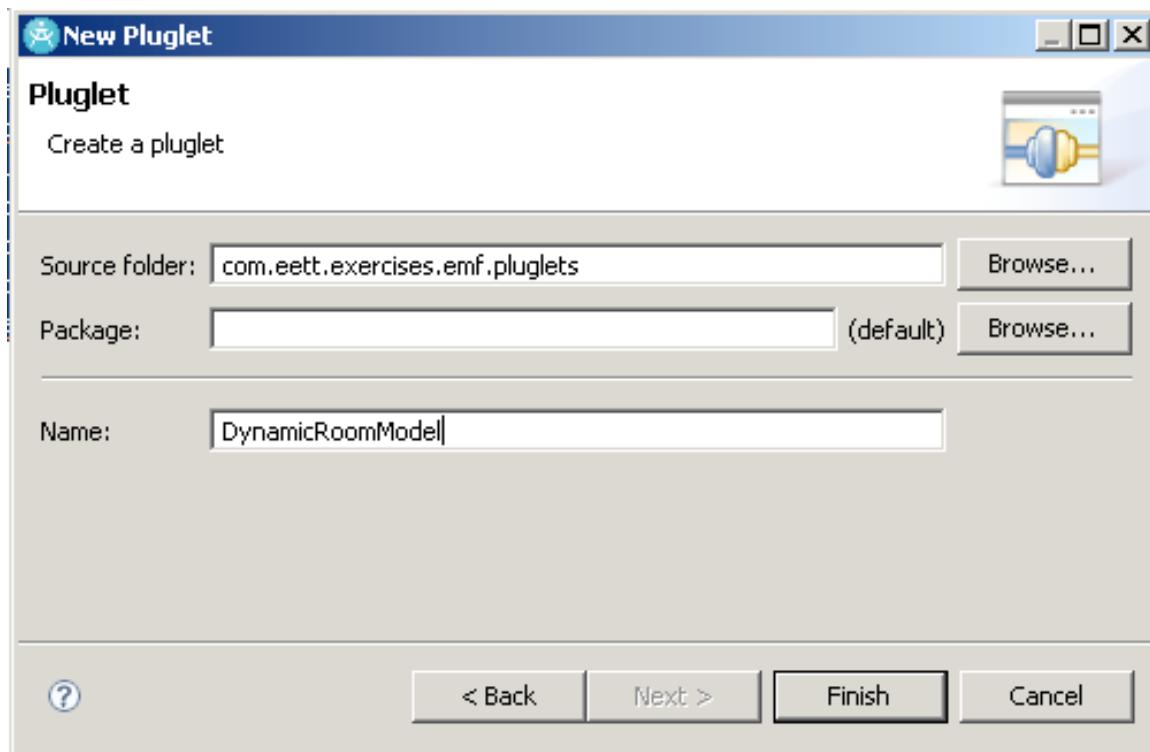
```
<?xml version="1.0" encoding="UTF-8"?>
<pluglets>
    <require>
        <import plugin="com.ibm.xtools.plugin"/>
        <import plugin="org.eclipse.emf.ecore"/>
    </require>
</pluglets>
```

The Pluglets Plug-in Dependencies should now include org.eclipse.emf.ecore and org.eclipse.emf.common.



Now create a new Pluglet (File → New → Pluglet) in the project. Select the Simple → Pluglet template. Click the Next > button. Set the name of the Pluglet to DynamicROOMModel and click Finish.





Open the DynamicRoomModel.java file and enter the code described below. This code creates two Actor objects, AbstractDyeingController and DyeingController. They are stored in separate resources, abstractDyeingController.xmi and dyeingController.xmi respectively. We then set the actorSuperclass of DyeingController to be AbstractDyeingController.

```

import java.io.IOException;

import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.EClass;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.EPackage;
import org.eclipse.emf.ecore.EStructuralFeature;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.emf.ecore.resource.ResourceSet;
import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;

import com.ibm.xtools.plugin.Pluglet;

public class DynamicRoomModel extends Pluglet {
    private static final String MODELS_FOLDER
        = "platform:/resource/com.eett.exercises.emf.plugin/models/";

    public void plugletmain(String[] args) {

        // Retrieved the EPackage for the Room metamodel that we
        registered
        // It is retrieved using the string we specified as the URI when
        we
        // registered it in the dynamic_package extension point
        EPackage room
            = EPackage.Registry.INSTANCE
                .getEPackage("http://www.eett.com/room/2008");

        if(room != null) {
            // Create a ResourceSet so that we can create Resources
            // to store the model elements we create in
            ResourceSet resourceSet = new ResourceSetImpl();

            // Create the Resources to store the objects that we create in
            Resource controllerResource
                = resourceSet.createResource(URI.createURI(
                    MODELS_FOLDER + "dyeingController.xmi"));
            Resource abstractControllerResource
                = resourceSet.createResource(URI.createURI(
                    MODELS_FOLDER + "abstractDyeingController.xmi"));

            // In order to be able to create an Actor object we need
            // have its EClass which we can get from the EPackage
            EClass actorEClass = (EClass) room.getEClassifier("Actor");

            // Similarly, to set the features of an Actor we
            // need the EStructuralFeature objects for them which we
            // can get from the EClass
            EStructuralFeature actorNameAttribute
                = actorEClass.getEStructuralFeature("name");
            EStructuralFeature actorSuperclass
                = actorEClass.getEStructuralFeature("actorSuperclass");

            // Create the AbstractDyeingController Actor and add it
    }
}

```

```

// to the Resource we created for it
EObject abstractController =
    room.getEFactoryInstance().create(actorEClass);
abstractController
    .eSet(actorNameAttribute, "AbstractDyeingController");
abstractControllerResource.getContents()
    .add(abstractController);

// Create the DyeingController Actor and add it
// to the Resource we created for it
EObject controller =
    room.getEFactoryInstance().create(actorEClass);
controller
    .eSet(actorNameAttribute, "DyeingController");
controllerResource.getContents().add(controller);

// Set the actorSuperclass feature of the dyeingController
// to be the abstractController
controller.eSet(actorSuperclass, abstractController);

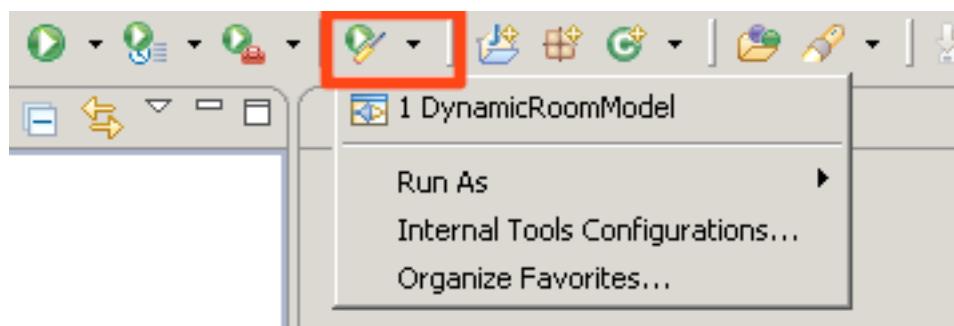
// Save and unload the Resources we created and added
// contents to
try {
    abstractControllerResource.save(null);
    controllerResource.save(null);
    abstractControllerResource.unload();
    controllerResource.unload();
} catch (IOException e) {
    out.println("Could not save the resources.");
}

}

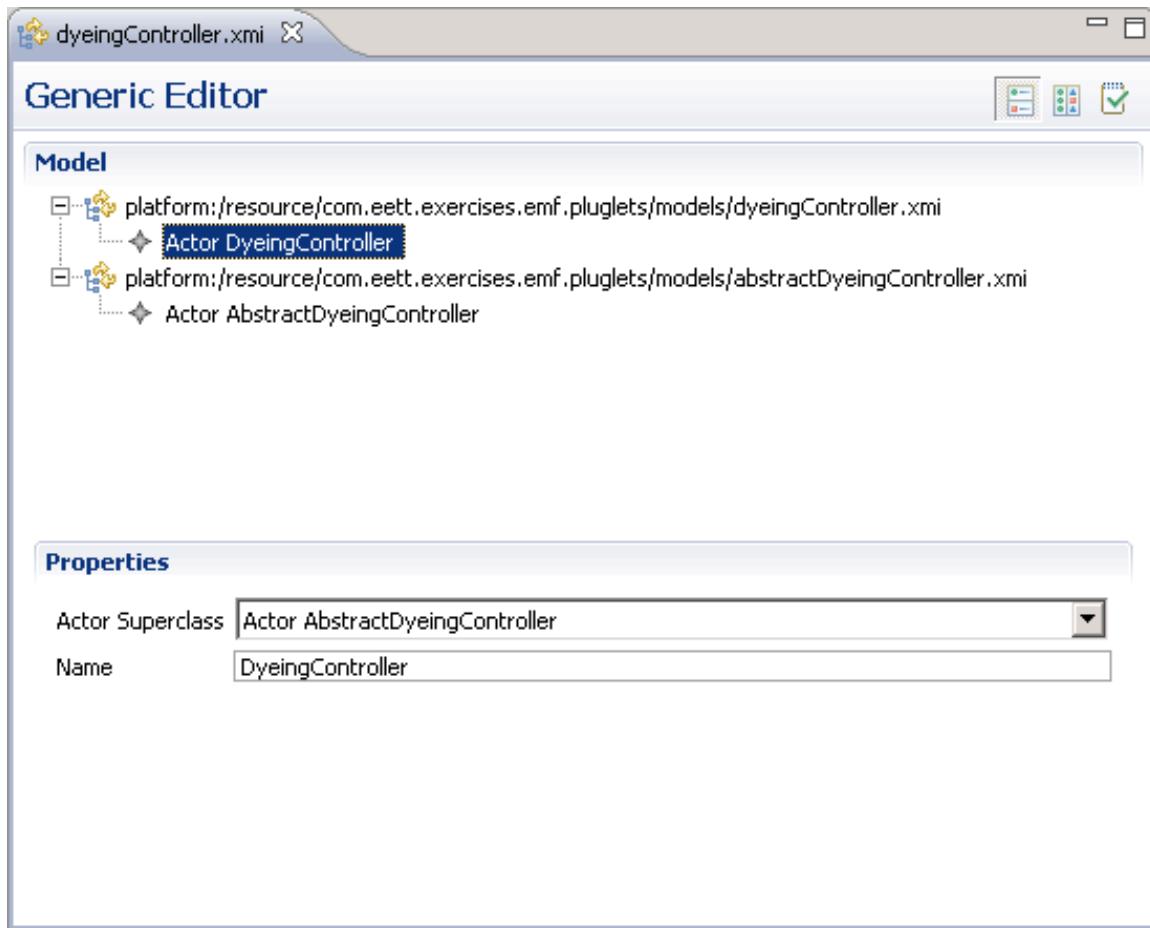
else {
    out.println("Could not get the room EPackage");
}
}
}

```

Run the Pluglet by selecting Run → Internal Tools → DynamicRoomModel or from the toolbar as shown below.



Running the Pluglet should create a new folder models in your project with two files. Open the dyeingController.xmi file with the Generic Editor. Notice how it also loaded the abstractDyeingController.xmi because there is a cross-reference to it (actorSuperclass of DyeingController).

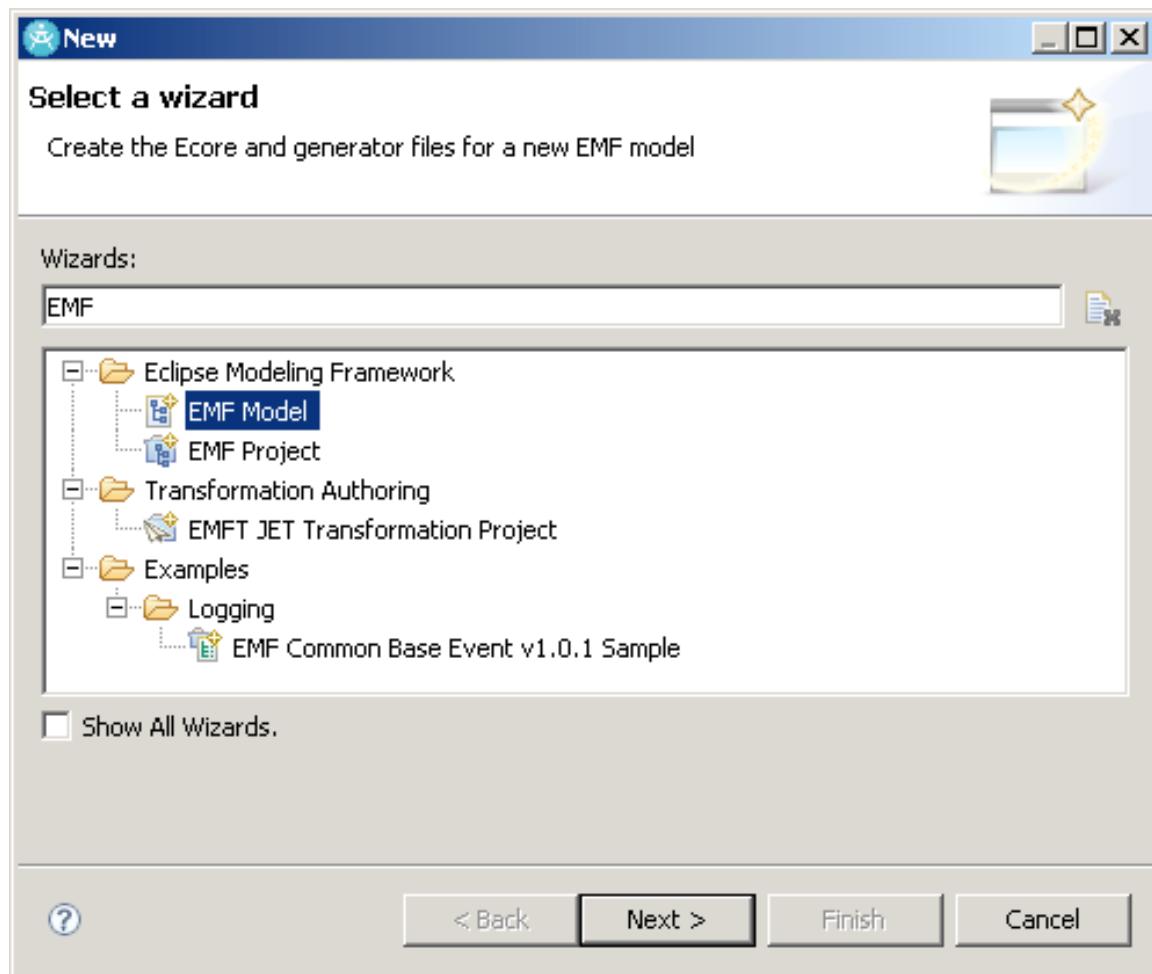


2.3 Generating code for the Ecore Model

The Ecore model created in the previous exercise specifies the concepts, relationships and capabilities within our ROOM domain. In order to be able to generate code for this model we need to create a generator model that decorates the metamodel you defined with additional information to help with generation.

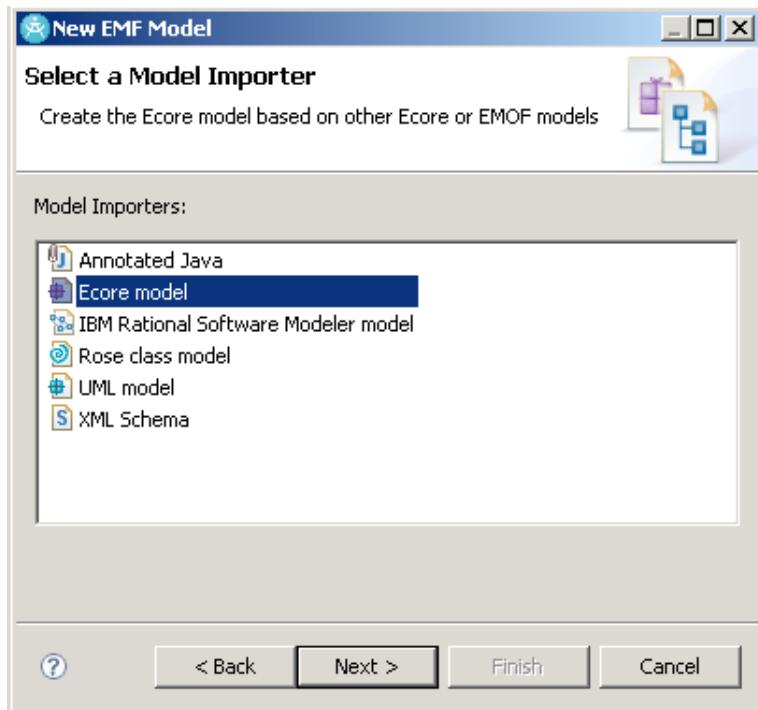
On the models folder in the com.eett.exercises.emf project right-click and select New → Other...

In the New wizard select EMF Model under the Eclipse Modeling Framework folder and click Next >

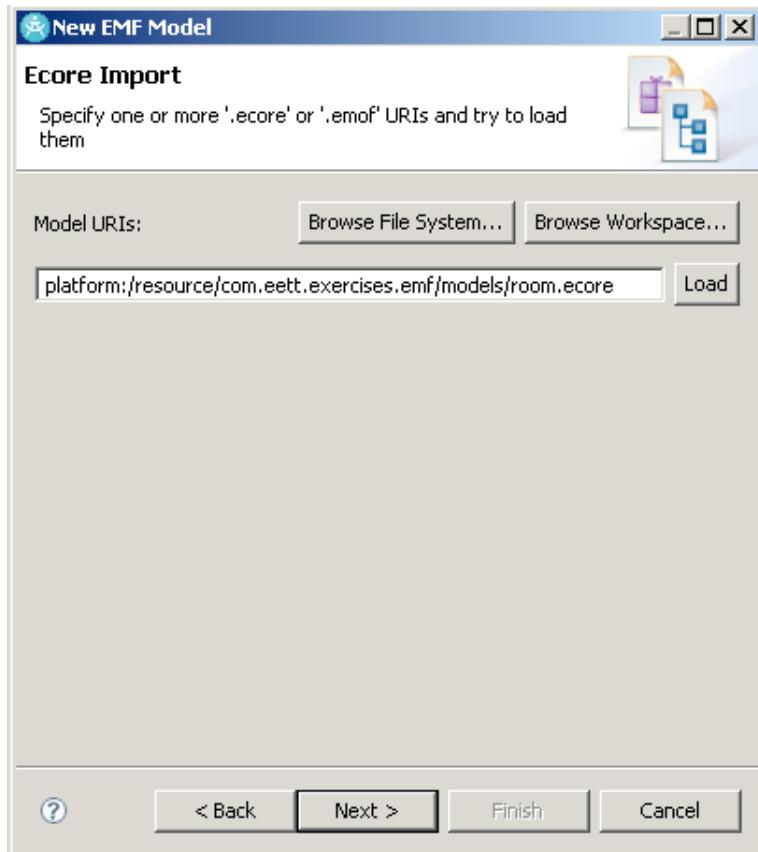


Set the file name to room.genmodel and click the Next > button

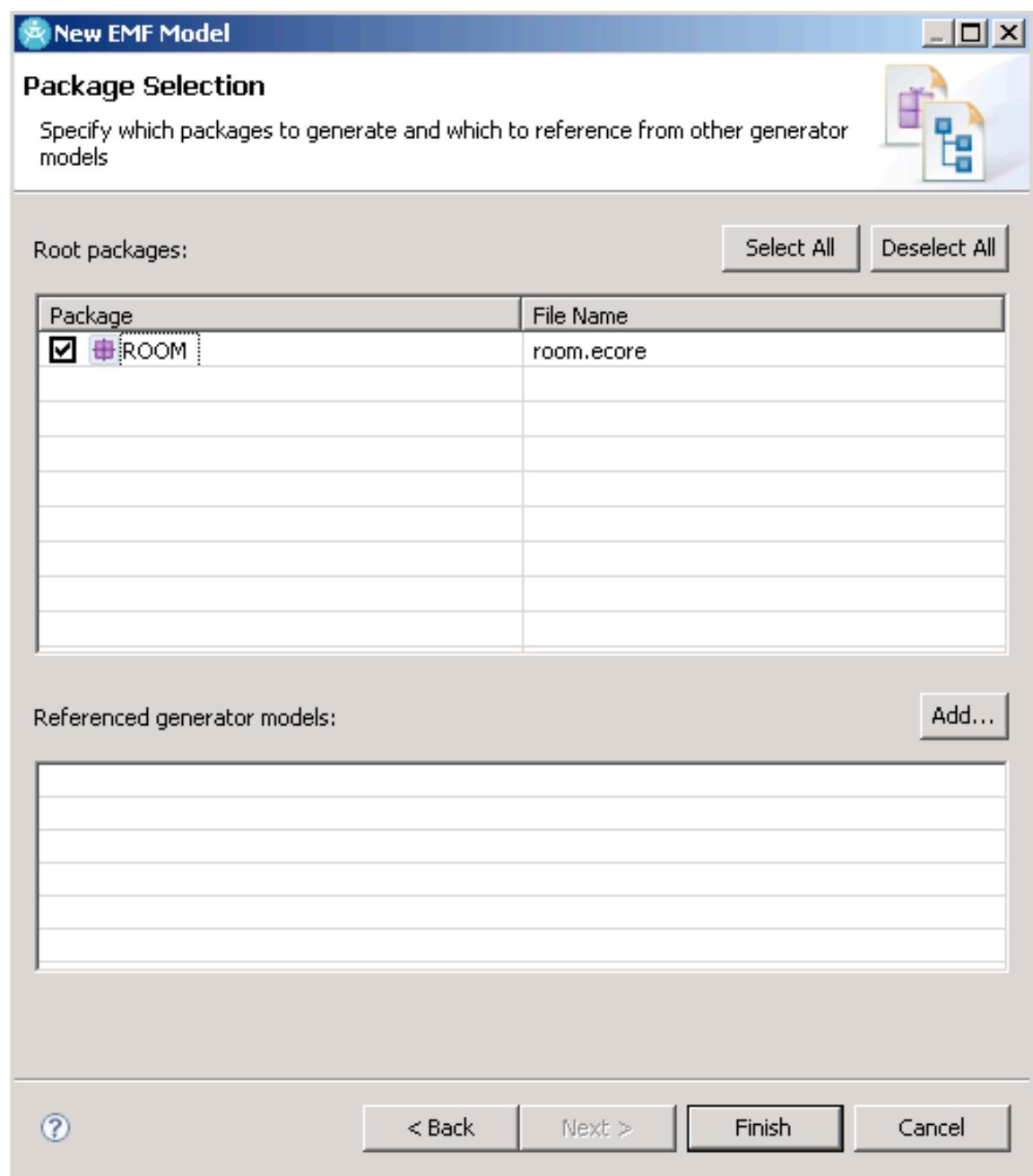
A Model Importer must be selected to create our generator model, select the Ecore Model option and click Next >



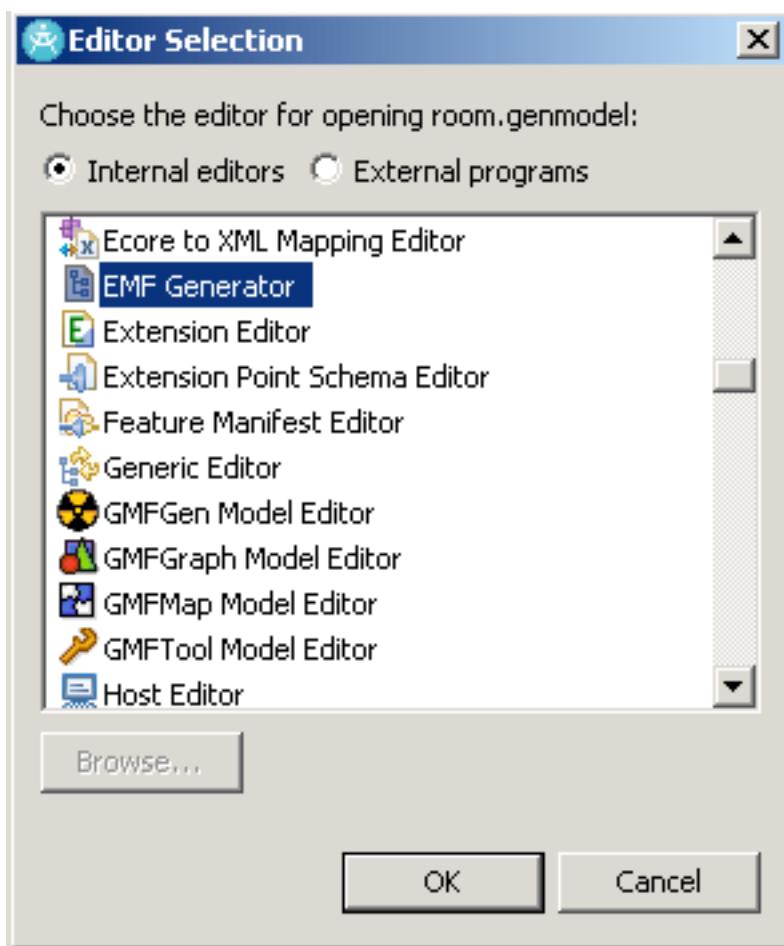
On the Ecore Import page use the Browse Workspace to locate and select the room.ecore model and click the Load button and then Next >



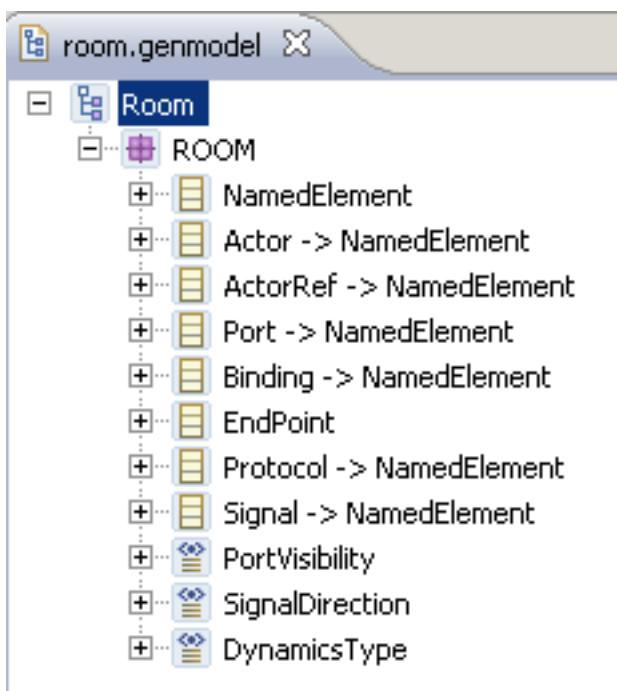
The following dialog should appear, select the Finish button



Open the new room.genmodel with the EMF Generator editor,



The model should look like the following

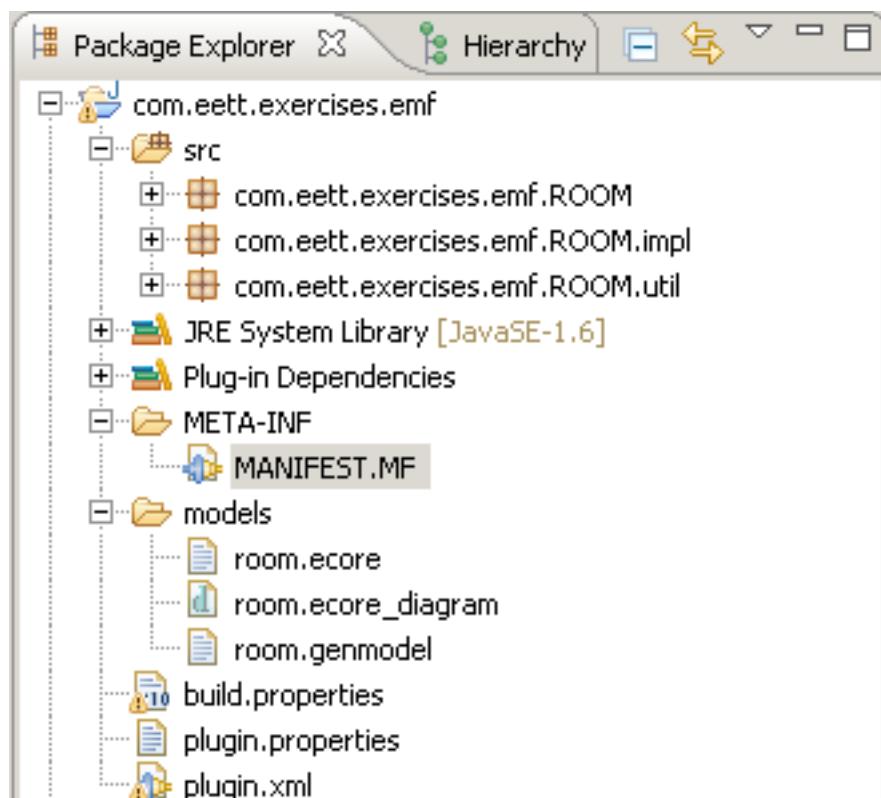


Explore the generator model observing the additional properties that have been added to the elements from our Ecore model. At this point the only property that we need to change is Base Package on the root EPackage ROOM. Change this property to com.eett.exercises.emf, this is the package that the generated code will be generated into.

Property	Value
All	
Base Package	com.eett.exercises.emf
Prefix	ROOM
Ecore	
Edit	
Editor	
Model	
Package Suffixes	
Tests	

With the generator model open right-click on the root package element and select Generate All, this will generate three new plug-ins (com.eett.exercises.emf.edit, com.eett.exercises.emf.editor, and com.eett.exercises.emf.test) and add source to the com.eett.exercises.emf project that the model is in. It will also modify the com.eett.exercises.emf project adding the plug-in and Java nature to the project.

The com.eett.exercises.emf project should now resemble



If there are errors in the three other projects we need to update the plug-in manifest for the com.eett.exercises.emf. Open the MANIFEST.MF file in the Plug-in Editor and change to the Runtime page. We need to export the com.eett.exercises.emf.ROOM and com.eett.exercises.emf.ROOM.util packages that were generated because the .edit, .editor and .test plug-ins are dependent on them.

```

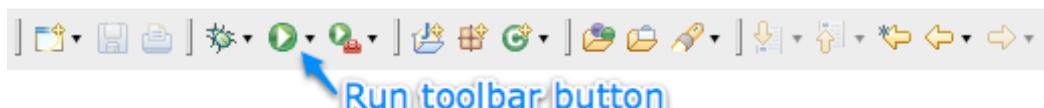
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Emf
Bundle-SymbolicName: com.eett.exercises.emf;singleton:=true
Bundle-Version: 1.0.0
Require-Bundle: org.eclipse.emf.ecore
Export-Package: com.eett.exercises.emf.ROOM,
    com.eett.exercises.emf.ROOM.util
Bundle-RequiredExecutionEnvironment: JavaSE-1.6

```

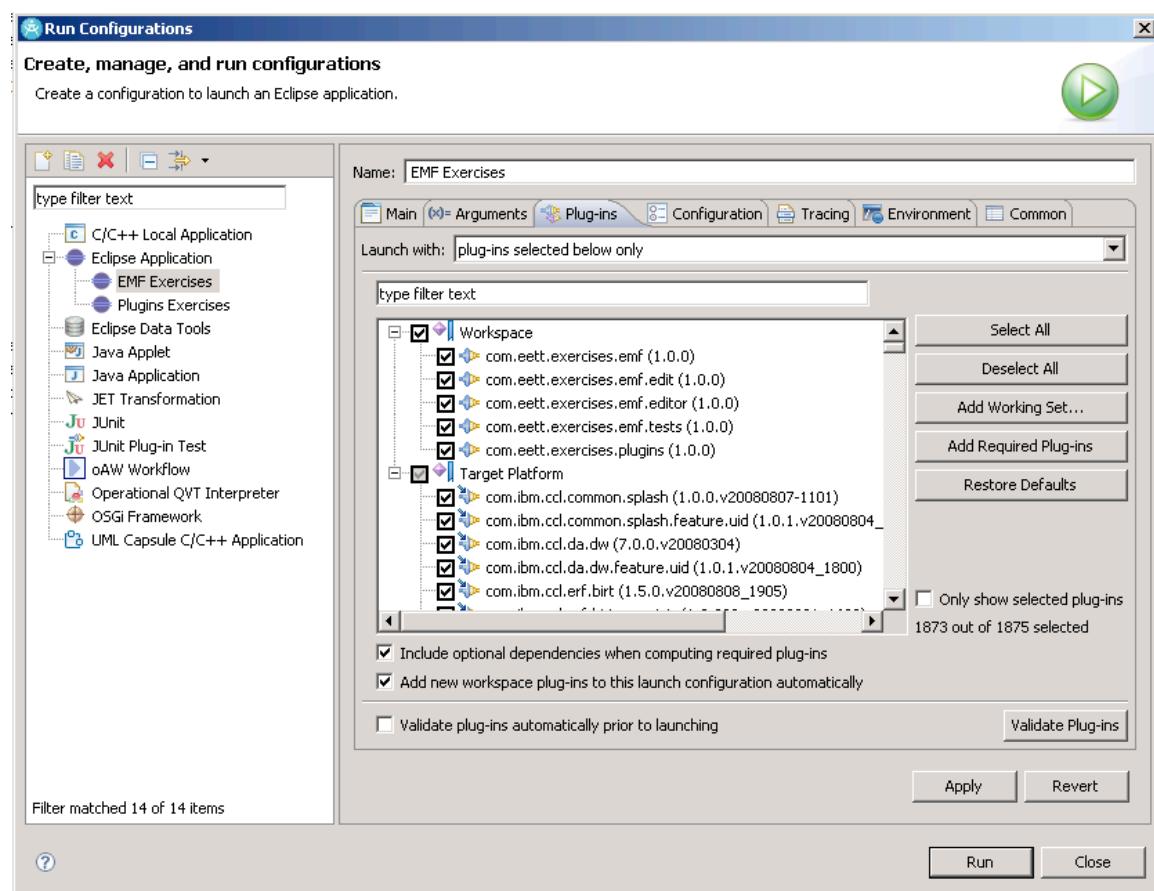
We need to change the registration of the metamodel to point to the generated package. Comment out the org.eclipse.emf.ecore.dynamic_package extension in plugin.xml and add a new extension org.eclipse.emf.ecore.generated_package. Set the URI to be <http://www.eett.com/room/2008> and the class to com.eett.exercises.emf.ROOM.ROOMPackage.

For now we won't worry about the code that was generated, instead we will create a model using the generated model editor.

Switch to the Java perspective and open the Run Configurations dialog by selecting Run → Run Configurations... from the main menu or by using the toolbar.

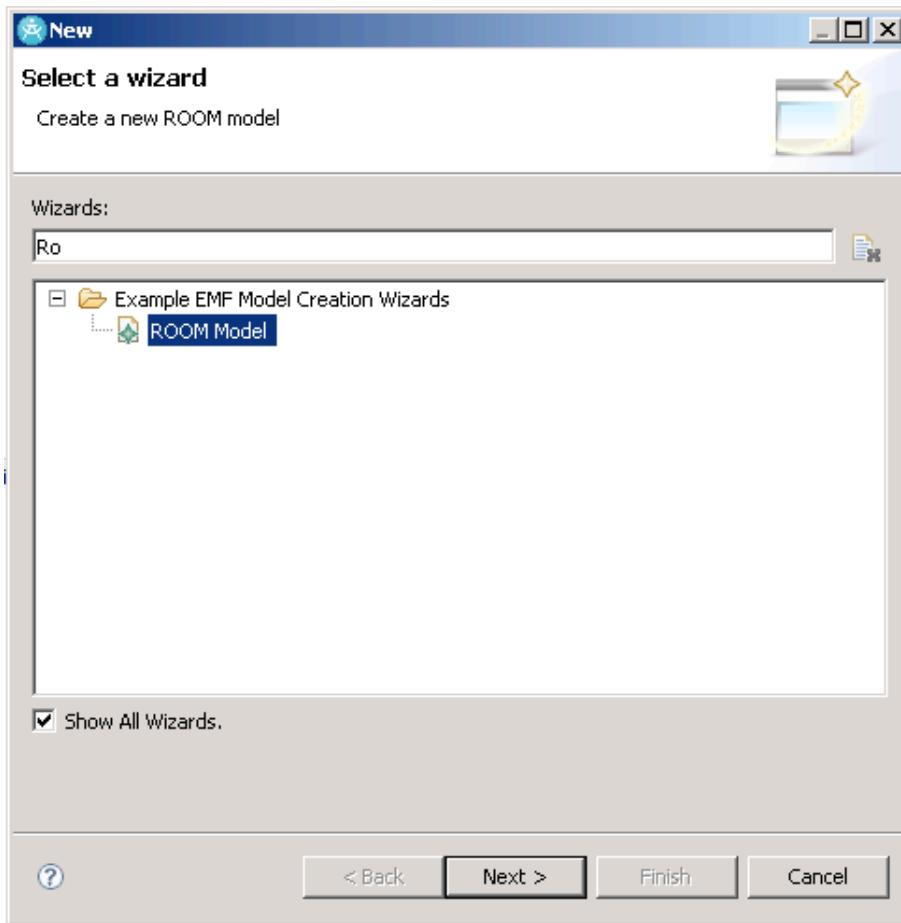


Select EMF Exercises Run Configuration by selecting the Eclipse Application entry on the left and make sure that our new plug-ins are included.



Now click on the Apply button followed by the Run button, this should launch a Runtime Workbench that will enable you to use the editor that was generated for the ROOM metamodel.

To create a ROOM model select the models folder in our com.eett.exercises.emf.plugin project and File → New → Other... In the New wizard select the ROOM Model wizard and click Next >.



Set the name of the model to DyeingController.room and click Next >. In the ROOM Model wizard set the Model Object to Actor and click Finish.

This will create a new resource with an Actor object in it and open it in the ROOM Editor, which is customized to the ROOM metamodel. Notice that the shortcoming of our current metamodel is that we have no top container element allowing us to create other elements of our model in the same resource.

Exit the runtime workbench and modify the metamodel to include a Model EClass that can contain Actors and Protocols. Reload the generator model (in the EMF Generator editor, Generator → Reload... from the main menu.) Regenerate the model and re-launch the runtime workbench to be able to create a ROOM Model with Model as the Model Object.

Possible improvements

- Change the Wizard category of the ROOM Model in the wizard
- By changing the _UI_Wizard_category property in the plugin.properties file
- Add a perspective where the ROOM Model is listed in the New Menu with having to go through the Select a wizard

2.4 Implement a derived attribute and operation

In this exercise we will add a derived attribute and an operation into our model and then provide the logic for them. An Actor has a reference to Port and a port has different attributes, one of which is visibility. We will consider all public Ports to be interface ports, so we can derive the list of interface ports from the 'ports' existing ports reference. Similarly, we may want to have an operation on an Actor for creating an interface port, given a name.

Open the room.ecore model in the Sample Ecore Model Editor and modify the Actor EClass adding

2.5 Querying our model

In this exercise you will create a plug-in that extends the ROOM Model Editor with a query that will find all Actor's that do not have a actorSuperclass. In order to do this we will add an action to the model editor that invokes the query we will write.

Start by creating a new plug-in project with

ID	com.eett.exercises.emf.query,
Name	EETT EMF Query Exercise
Provider	EETT
Version	1.0.0
This plug-in is a singleton	<input checked="" type="checkbox"/>

The screenshot shows the Eclipse Plug-in Editor interface. The title bar says "com.eett.exercises.emf.query". The main area has a tab titled "Overview".

General Information

This section describes general information about this plug-in.

ID:	com.eett.exercises.emf.query
Version:	1.0.0
Name:	EETT EMF Query Exercise
Provider:	EETT
Platform Filter:	
Activator:	<input type="text"/> <input type="button" value="Browse..."/>

Activate this plug-in when one of its classes is loaded
 This plug-in is a singleton

Execution Environments

Specify the minimum execution environments required to run this plug-in.

JavaSE-1.6	<input type="button" value="Add..."/> <input type="button" value="Remove"/> <input type="button" value="Up"/> <input type="button" value="Down"/>
------------	--

[Configure JRE associations...](#)
[Update the classpath settings](#)

Navigation tabs at the bottom: Overview, Dependencies, Runtime, Extensions, Extension Points, Build, MANIFEST.I

Switch to the Dependencies page of the Plug-in Editor and add the following dependencies

- org.eclipse.ui
- org.eclipse.core.runtime
- org.eclipse.emf.ecore
- org.eclipse.emf.query
- com.eett.exercises.emf
- com.eett.exercises.emf.editor

In order for the query to accessible from the ROOM Editor we need to add an extension to the editor. Switch to the Extensions page of the Plug-in Editor and click the Add... button in the All Extensions section.

From the Extension Point Selection dialog select the org.eclipse.ui.editorActions extension point (you may have to uncheck the 'Show only extension points from the required plug-ins' option if you haven't all the dependencies listed above.)

If an editorContribution isn't automatically added for you, select the extension point and New → editorContribution from its context menu. Set the details for the editorContribution to:

id	com.eett.exercises.emf.query.editorContribution
targetID	com.eett.exercises.emf.ROOM.presentation.ROOMEditorID

The targetID field specifies the identifier that was generated for the ROOM editor, this is necessary for the workbench to know the specific editor that we are contributing to. A list of available editors can be found by clicking the Browse... button beside the field.

Add a menu to the editorContribution through its context menu and set its details as shown in the following table. The path field identifies the menu that we are contributing to in the ROOM editor, this id is defined in the ROOM editor plug-in that was generated from the ROOM EMF model. Add a separator to this menu called additions.

id	com.eett.exercises.emf.queryMenuID
label	&Query
path	com.eett.exercises.emf.ROOMMdd /addtions

Add an action to the editorContribution through its context menu and set its details as shown in the following table. We are adding this action to the menu that we defined in the previous step, as referenced in the menubarPath. In the next step we will create the class we have defined in the class field.

id	com.eett.exercises.emf.query.NoSuperclassQuery
label	Actors with No Superclass
class	com.eett.exercises.emf.query.actions.NoSuperclassQuery
menubarPath	com.eett.exercises.emf.ROOMMdd / com.eett.exercises.emf.queryMenuID/addtions

We do not want the action to be enabled for anything so we will add an enablement to our action to control when it can be invoked. Add an enablement with an objectClass through the action's context menu and set the name to org.eclipse.emf.ecore.EObject which will restrict the enablement to only objects of type EObject.

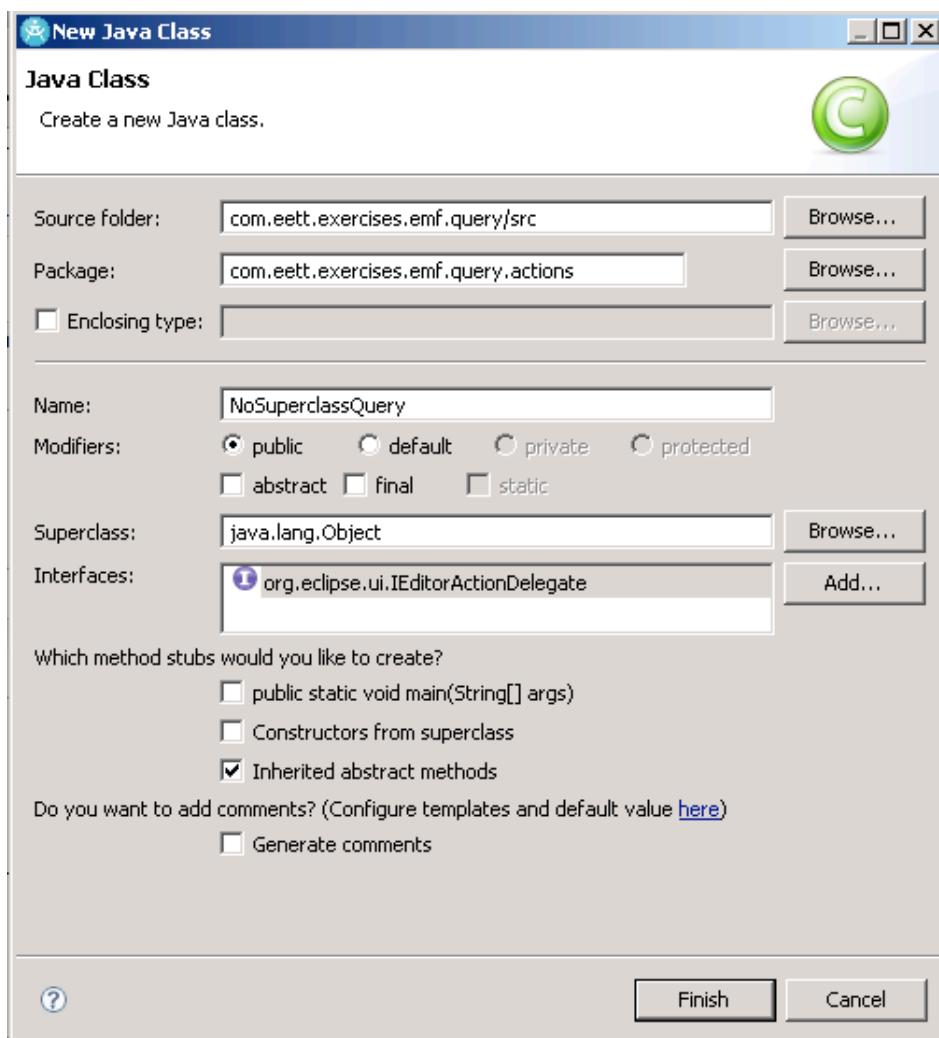
The plugin.xml page should now resemble:

```

<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
<extension
  point="org.eclipse.ui.editorActions">
<editorContribution
  id="com.eett.exercises.emf.query.editorContribution"
  targetID="com.eett.exercises.emf.ROOM.presentation.ROOMEditionID">
<menu
  id="com.eett.exercises.emf.queryMenuID"
  label="&Query"
  path="com.eett.exercises.emf.ROOMMenuID/additions">
  <separator name="additions" />
</menu>
<action
  class="com.eett.exercises.emf.query.actions.NoSuperclassQuery"
  id="com.eett.exercises.emf.query.NoSuperclassQuery"
  label="Actors with No Superclass"
  menubarPath="com.eett.exercises.emf.ROOMMenuID/
com.eett.exercises.emf.queryMenuID/additions"
  style="push">
  <enablement>
    <objectClass name="org.eclipse.emf.ecore.EObject"></objectClass>
  </enablement>
</action>
</editorContribution>
</extension>
</extension>

```

We will now create the handler for our query action in the src folder of the plugin add a Class using the wizard (New → Class). Set the name to NoSuperclassQuery and the package to com.eett.exercises.emf.query.actions. The handler must implement the IEditorActionDelegate.



In the newly created class there are two methods that were interested implementing setActiveEditor and run, we will also add an additional operation to perform the actually query.

The setActiveEditor will allow us to capture the current editor and to access the selected model element to use as the source for our query. To support the method add an editor field to the class of type ROOMEeditor. The behavior of setActiveEditor follows.

```
@Override
public void setActiveEditor(IAction action, IEditorPart targetEditor)
{
    if(editor instanceof ROOMEeditor) {
        editor = (ROOMEeditor) targetEditor;
    }
}
```

The run action will invoke the query that we will write and then select the objects returned by the query in the editor. The behavior of run follows.

```

@Override
public void run(IAction action) {
    Collection<EObject> selectedObjects
        = new ArrayList<EObject>();

    // get the selected object from the editor
    if(editor != null
        && editor.getSelection() instanceof
    IStructuredSelection){

        IStructuredSelection structuredSelection =
            (IStructuredSelection) editor.getSelection();
        for(Object next : structuredSelection.toList()){
            if(next instanceof EObject){
                selectedObjects.add((EObject) next);
            }
        }
    }

    // execute the query
    Collection<EObject> result = performQuery(selectedObjects);

    // select the query results in the editor
    if(!result.isEmpty()){
        editor.setSelectionToViewer(result);
    }
}
}

```

Finally, we write a query that takes the selected element in the editor and find all the Actors that do not have the actorSuperclass reference set.

```

private Collection<EObject> performQuery
    (Collection<EObject> selectedObjects) {

    // create a condition that tests whether an Actor's
    // actorSuperclass feature to see if it is null
    EObjectCondition condition = new
    EObjectReferenceValueCondition(
        ROOMPackage.eINSTANCE.getActor_ActorSuperclass(),
        EObjectInstanceCondition.IS_NULL);

    SELECT query = new SELECT(
        new FROM(selectedObjects),
        new WHERE(condition));

    return query.execute();
}

```

This completes the creation of the plug-in that will perform the query. It needs to be tested in a runtime workbench. Create a runtime workbench including this plug-in and the ones that it requires. Open an existing ROOM model or create a new one using the ROOM Editor, ensure that the ROOM Editor menu contains a query sub-menu.

3 UML Module Exercises

3.1 UML Model Exercise

3.1.1 What this exercise is about

In this exercise, you will create a UML model of a dyeing system. The system that consists of a dye tank that has a fill valve and a drain valve, the tank has a high sensor to determine when it is filled to its maximum and low sensor to determine when it is getting low on dye. A controller receives signals from the sensors in order to open and close the appropriate valves.

At the end of this exercise, you should be able to:

Create a UML model using the UML Model Editor, and

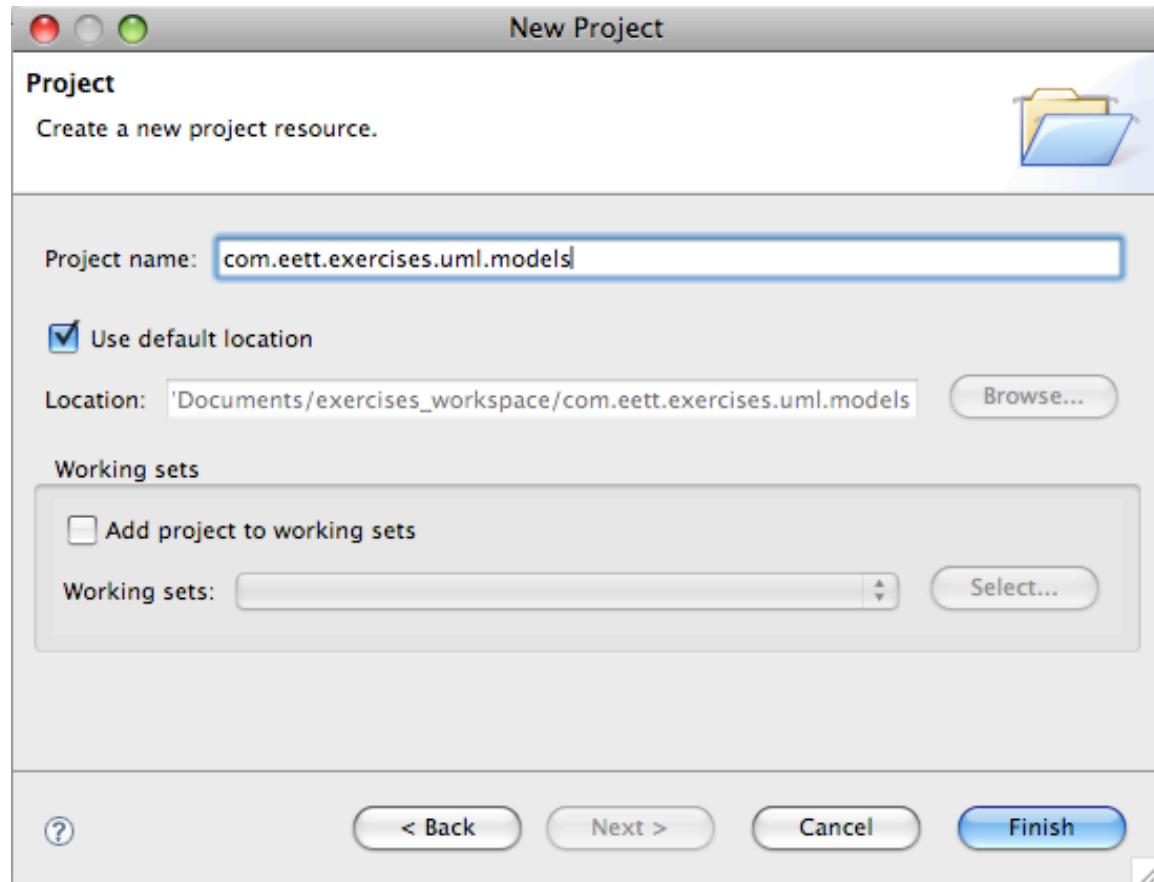
Programmatically create UML model elements

3.1.2 Exercise Instructions

Start by creating a project that will contain the models that we create in this exercise and others.

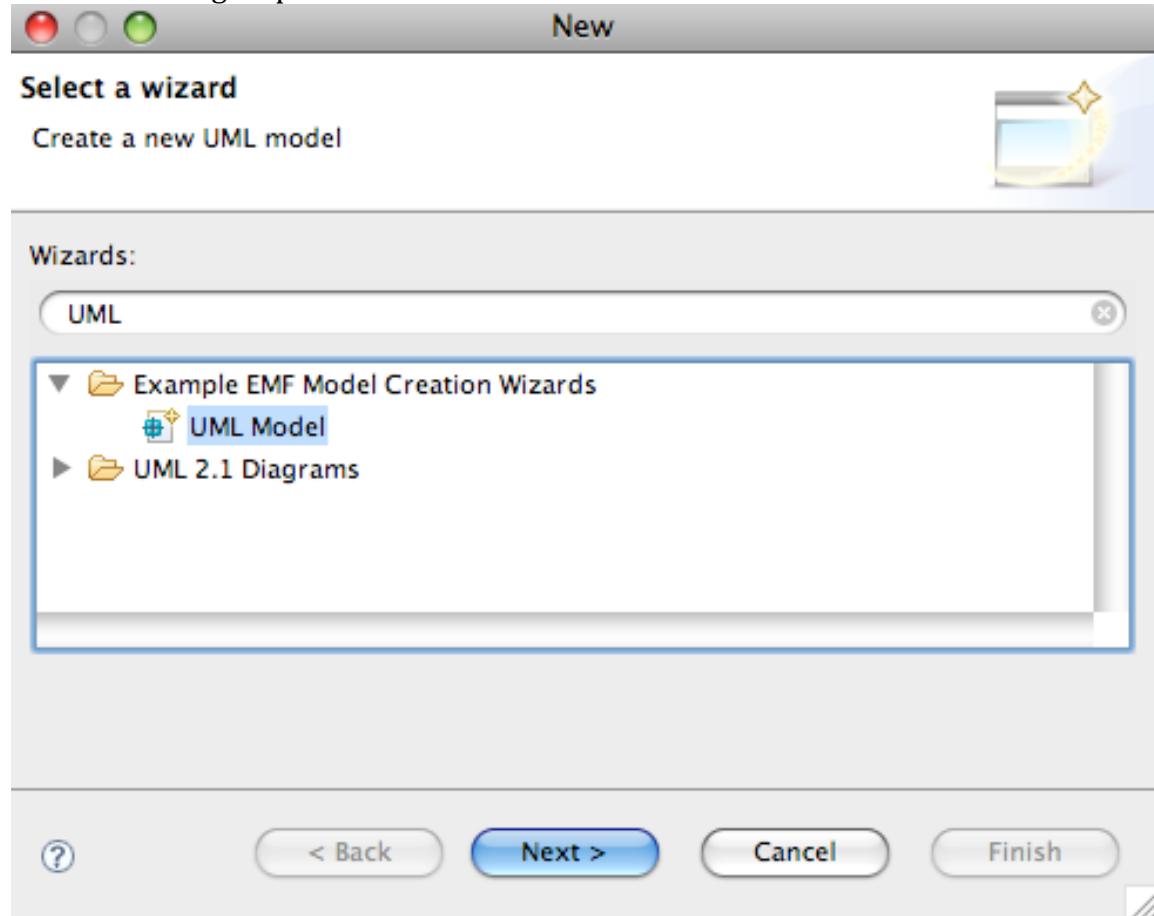
1. Create a new basic Eclipse project

1. Select File → New → Project... → Project
2. Set the Project name to com.eett.exercises.uml.models
3. Keep Use default location checked
4. Click Finish >

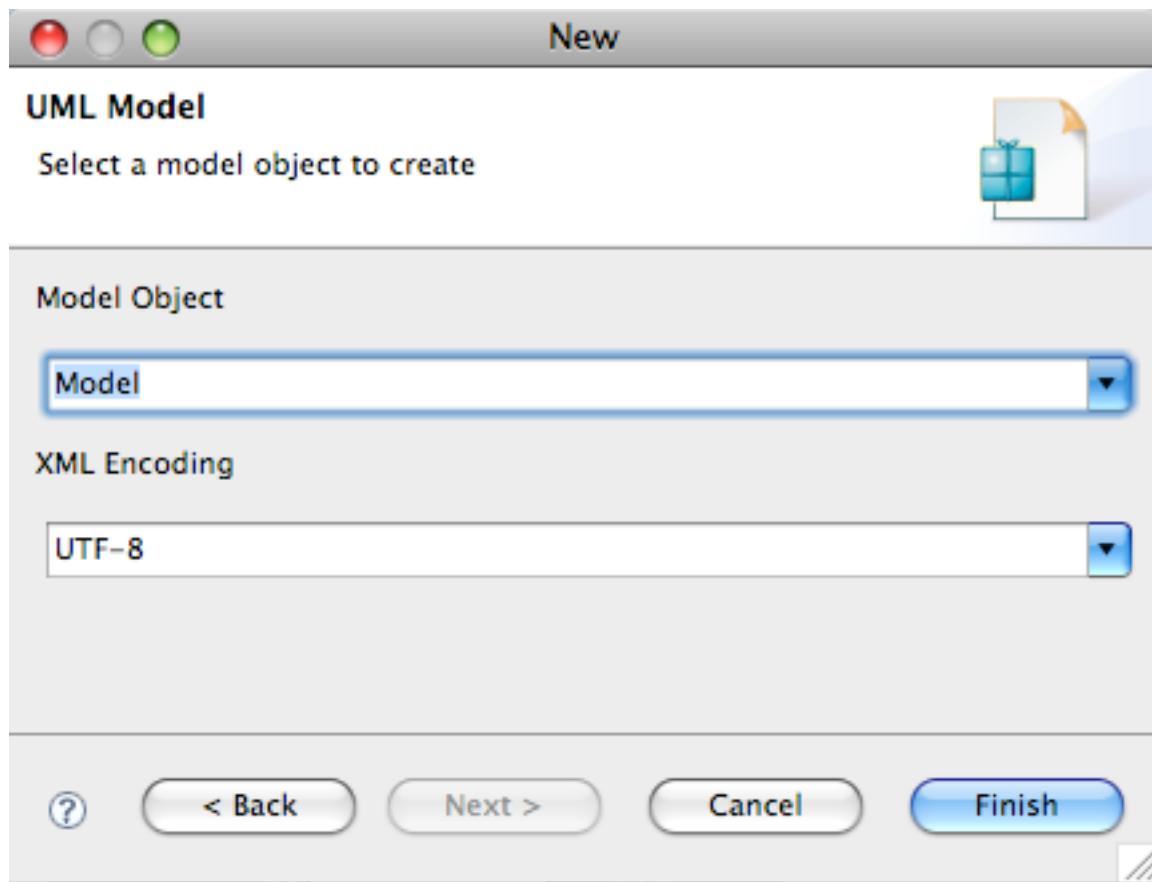


You should now have an empty project in your workspace

2. Use the project context menu we will add a folder to store our models in.
 1. Select the project and New → Folder from its context menu.
 2. Set the Folder name to models and click Finish
3. We will now create a UML model in our project and populate it with our dyeing system model
 1. Select the models folder and New → Other... from its context menu.
 2. In the New wizard select UML Model in the Example EMF Model Creation Wizards group.



3. Set the File Name to dyeingSystem.uml and click Next >
4. Set the Model Object field to Model and leave the XML Encoding field to the default value and click Finish



The project in your workspace should now contain a `dyeingSystem.uml` file, that you can open using the UML Model Editor.

3.2 UML Keywords Exercise

3.2.1 What this exercise is about

In this exercise, you will create a UML model using the UML Model Editor and tag the elements in the model using key words. You will also extend the UML model editor with actions to automatically add specific keywords to model elements.

At the end of this exercise, you should be able to:

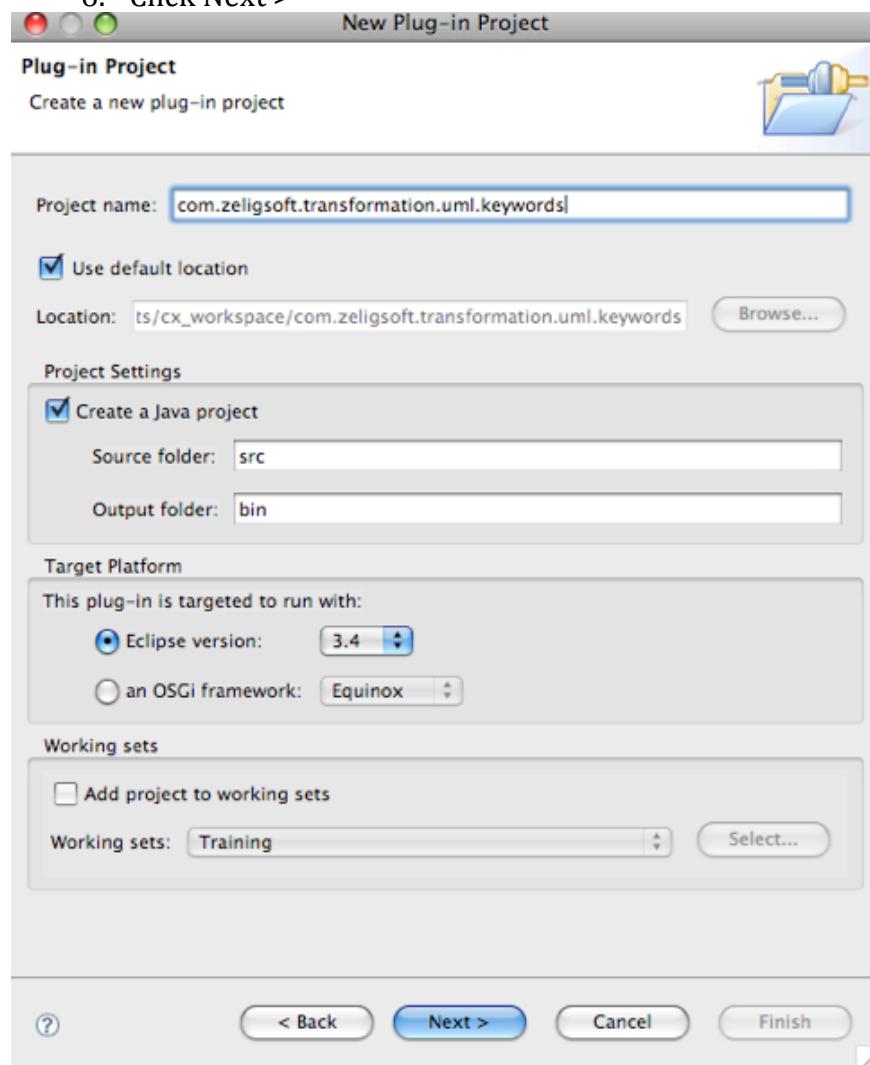
- Set key words in a UML model using the UML Model Editor
- Add actions to the UML Model Editor to tag elements with key words

3.2.2 Exercise Instructions

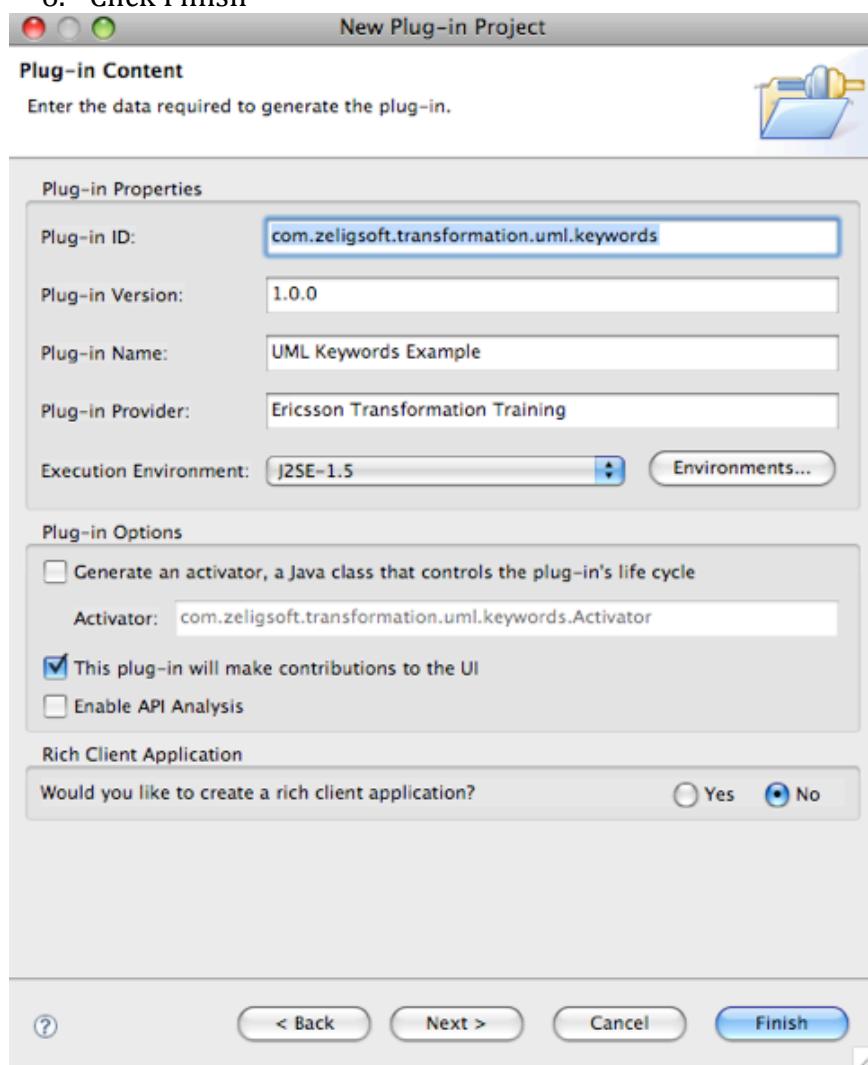
Start by creating a project with the name `com.eett.exercises.uml.keywords` in the workspace.

4. Switch to the Plug-in Development perspective, if not already there.
 1. Select Window → Open Perspective → Other... → Plug-in Development

5. Create a new Plug-in project as we will be adding actions to an editor
 1. Select File → New → Project... → Plug-in Project
 2. Set the Project name to com.eett.exercises.uml.keywords
 3. Keep Use default checked
 4. Keep Create a Java project checked
 5. Make sure that the Eclipse version is set to 3.4
 6. Click Next >

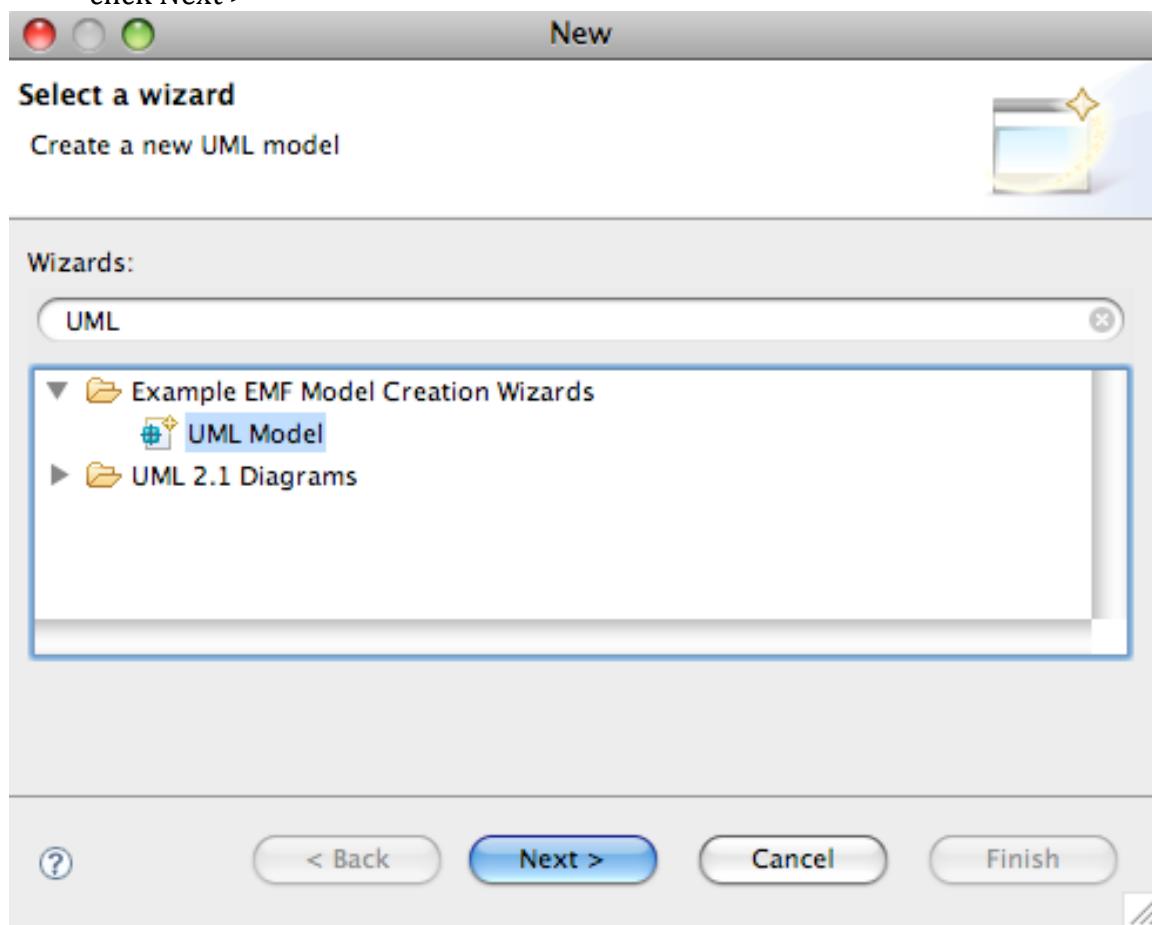


6. Configure the Plug-in Content dialog
 1. Keep the defaults for Plug-in ID and Plug-in Version
 2. Set the Plug-in Name to UML Keywords Example
 3. Set the Plug-in Provider to EETT
 4. Make sure that “This plug-in will make contributions to the UI” is the only Plug-in Option checked
 5. Make sure that Would you like to create a rich client application? Is set to No
 6. Click Finish

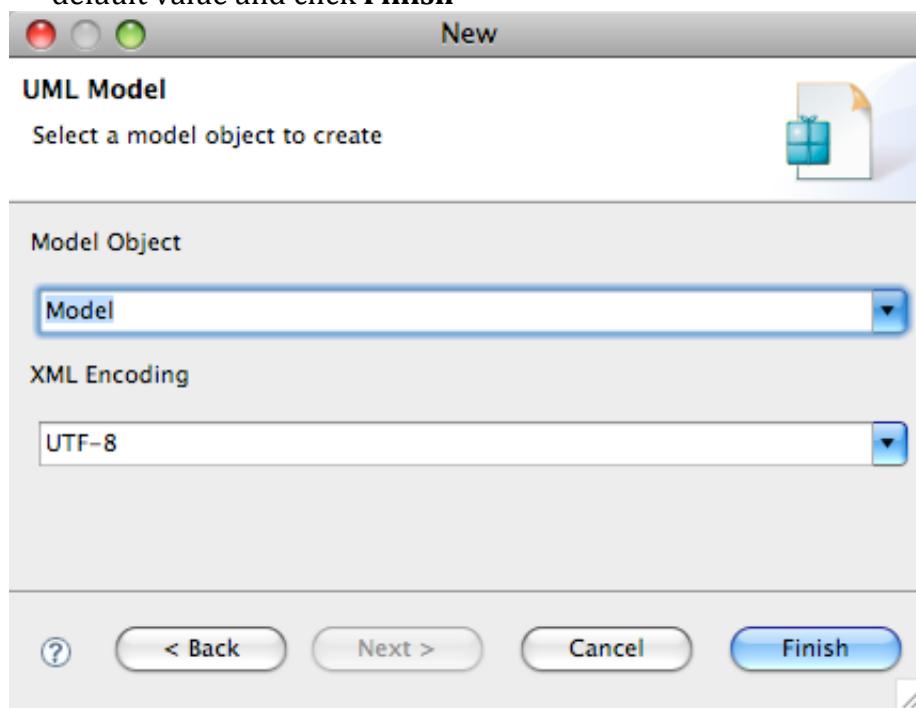


7. Create a UML model named dyeingSystem.uml in which we will build a UML model of a Dyeing system that consists of a dye tank that has a fill valve and a drain valve, the tank has a high sensor to determine when it is filled to its maximum and low sensor to determine when it is getting low on dye. A controller receives signals from the sensors in order to open and close the appropriate valves.
 1. Select the project created in the last step and New → Folder from the context menu
 2. Change the name of the folder to models

3. Select the models folder and File → New → Other... → UML Model and click Next >



8. Set the File Name to dyeingSystem.uml and click **Next >**
9. Set the Model Object field to Model and leave the XML Encoding field to the default value and click **Finish**



3.3 UML Profiles Exercise

3.3.1 What this exercise is about

In this exercise, you will create a Profile defining a ROOM like model. It will allow us to define, in UML, models that use the ROOM concepts. This includes for example identifying classes as ROOMActors and ports as ROOMPorts.

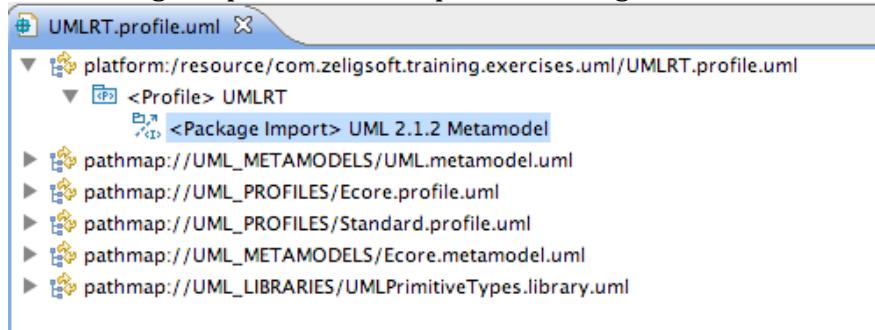
3.3.2 What you should be able to do

At the end of this exercise, you should be able to:

- Define a UML profile using the UML Model Editor
- Register and publish the profile to make it available at run-time
- Apply and use the profile in a UML model
- Add actions to the UML Model Editor to create elements with the stereotypes applied

3.3.3 Exercise instructions

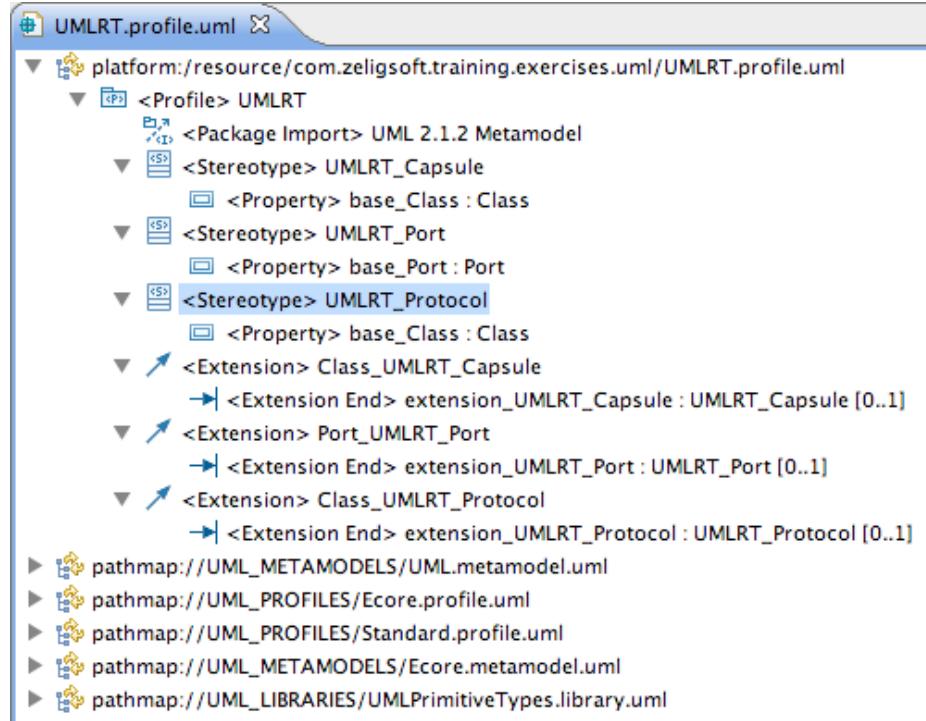
1. Select the **com.eett.exercises.uml** project and then choose File | New | Other...
2. Select UML Model in the Example EMF Model Creation Wizards folder and click the Next > button
3. Enter UMLRT.profile.uml as the file name and click the Next > button
4. Select Profile as the Model Object and click the Finish button
5. Open the new model file with the UML Model Editor and change the name of the Profile element to UMLRT
6. Load Resource, pathmap://UML_METAMODELS/UML.metamodel.uml
7. Add a Package Import and set Imported Package to uml



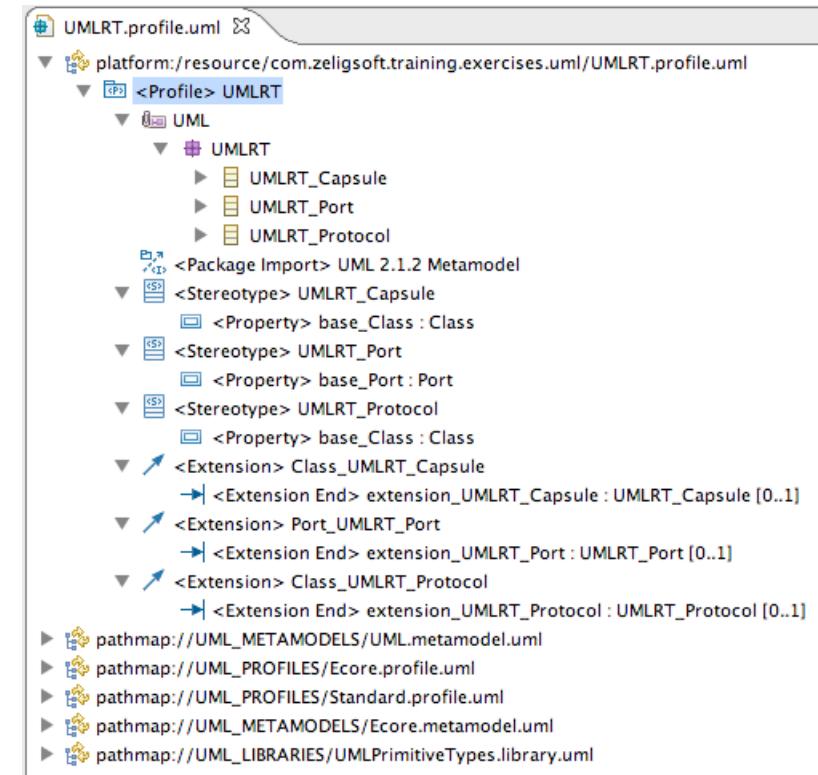
8. In the Profile create Stereotypes
 - a. UMLRT_Capsule
 - b. UMLRT_Port
 - c. UMLRT_Protocol

9. For each of the Stereotypes add a metaclass extension, select the Stereotype and select UML Editor | Stereotype | Create Extension ... from the main menu and select the appropriate metaclass

- UMLRT_Capsule – uml::Class
- UMLRT_Protocol – uml::Class
- UMLRT_Port – uml::Port

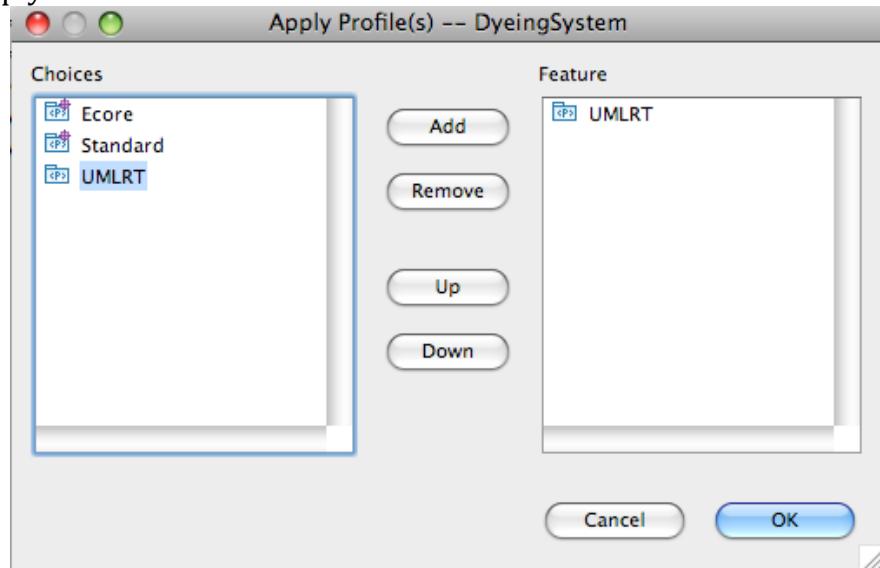


10. Make sure the UMLRT Profile element is selected in the model and then select UML Editor | Profile | Define, keeping the default options click the Ok button



3.4 Applying the Profile to a Model

1. Open the DyeingSystem.uml model with the UML Editor
2. Select UML Editor | Load Resource from the main menu
3. Enter UMLRT.profile.uml as the resource location and click the Ok button
4. Select the UMLRT Model element in the editor and UML Editor | Package | Apply Profile ...



5. Select Valve Class in the model and UML Editor | Element | Apply Stereotype ... and choose the UMLRT_Capsule Stereotype
 - a. Repeat for DyeingSystemController
 - b. Repeat for DyeingSystem
6. Select the Ports on each of the elements and apply the UMLRT_Port stereotype
7. Select the ValveControl Class and apply the UMLRT_Protocol stereotype

4 Transformation Exercises

4.1 M2M Setup

In order execute transformations from the ROOM Editor we need to add an extension to the editor. Create a new Plug-in project.

ID	com.eett.exercises.m2m.ui
Version	1.0.0
Name	EETT Transformations UI Exercise
Provider	EETT

Add the following dependencies.

- org.eclipse.core.runtime
- org.eclipse.ui
- com.eett.exercises.emf
- com.eett.exercises.emf.editor

Switch to the Extensions page of the Plug-in Editor and click the Add... button in the All Extensions section.

From the Extension Point Selection dialog select the org.eclipse.ui.editorActions extension point (you may have to uncheck the 'Show only extension points from the required plug-ins' option if you haven't all the dependencies listed above.)

If an editorContribution isn't automatically added for you, select the extension point and New → editorContribution from its context menu. Set the details for the editorContribution to:

id	com.eett.exercises.emf.query.editorContribution
targetID	com.eett.exercises.emf.ROOM.presentation.ROOMEditorID

The targetID field specifies the identifier that was generated for the ROOM editor, this is necessary for the workbench to know the specific editor that we are contributing to. A list of available editors can be found by clicking the Browse... button beside the field.

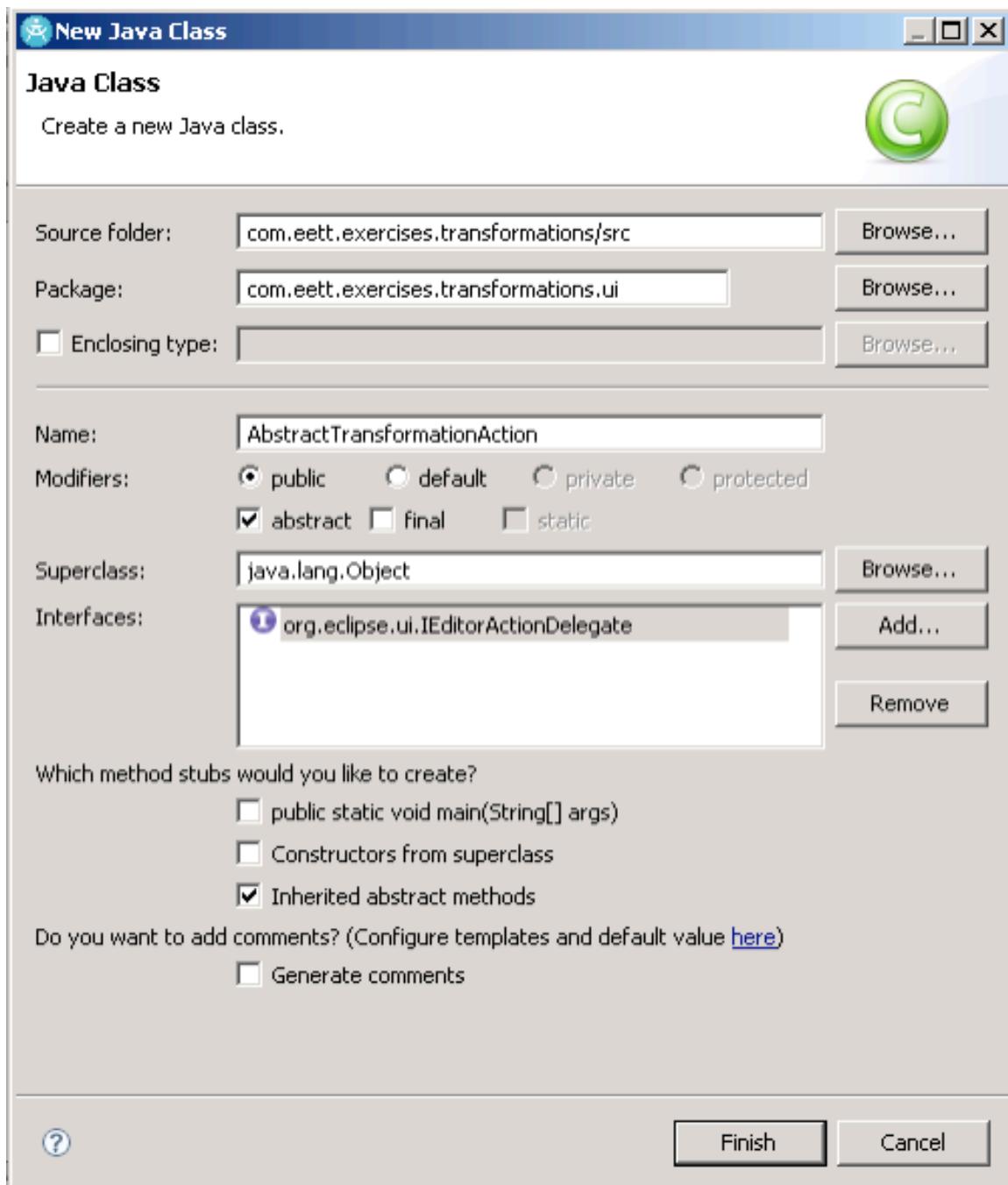
Add a menu to the editorContribution through its context menu and set its details as shown in the following table. The path field identifies the menu that we are contributing to in the ROOM editor, this id is defined in the ROOM editor plug-in that was generated from the ROOM EMF model. Add a separator to this menu called additions.

id	com.eett.exercises.transformations.MenuID
label	&Transform
path	com.eett.exercises.emf.ROOMMenuID/addtions

The plugin.xml page should now resemble:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension point="org.eclipse.ui.editorActions">
    <editorContribution
      id="com.eett.exercises.transformation.editorContribution"
      targetID="com.eett.exercises.emf.ROOM.presentation.ROOMEditionID">
      <menu
        id="com.eett.exercises.transformation.MenuID"
        label="&Query"
        path="com.eett.exercises.emf.ROOMMenuID/additions">
        <separator name="additions" />
      </menu>
    </editorContribution>
  </extension>
</plugin>
```

We will now create an abstract handler for our transformation actions in the src folder of the plugin add an abstract Class using the wizard (New → Class). Set the name to AbstractTransformationAction and the package to com.eett.exercises.transformations.actions. The handler must implement the IEditorActionDelegate.



The `setActiveEditor` method will allow us to capture the current editor and to access the selected model element to use as the source for our transformation. To support the method add an editor field to the class of type ROOMEeditor. The behavior of `setActiveEditor` follows.

```
@Override
public void setActiveEditor(IAction action, IEditorPart targetEditor)
{
    if(editor instanceof ROOMEeditor) {
        editor = (ROOMEeditor) targetEditor;
    }
}
```

We will add an additional method to the class to return the selected object. If there is more than one object selected it will pick the first one in the list.

```
// Return the object that is currently selected in the active editor
// if more than one is selected return the first one
protected EObject getSelectedObject() {
    if(editor != null)
        && editor.getSelection() instanceof IStructuredSelection)
    {
        IStructuredSelection structuredSelection =
            (IStructuredSelection) editor.getSelection();
        for(Object next : structuredSelection.toList()){
            if(next instanceof EObject){
                return (EObject) next;
            }
        }
    }
    return null;
}
```

4.2 M2M with Java

In this exercise we will create a model-to-model mapping from our ROOM model to a simplified model of the Java programming language. This requires the java.ecore modeled that will be provided. Before doing the exercises generate and register the model. The remainder of the exercise assumes that you have done this.

Create a new Plug-in Project.

ID	com.eett.exercises.m2m.java
Version	1.0.0
Name	EETT M2M Java Exercise
Provider	EETT

Dependencies

org.eclipse.emf.java
com.eett.exercises.emf

Add an action to the ROOM Editor in the com.eett.exercises.transformations plug-in. Add an action to the editorContribution through its context menu and set its details as shown in the following table. We are adding this action to the menu that we defined in the previous step, as referenced in the menubarPath. In the next step we will create the class we have defined in the class field.

id	com.eett.exercises.transformation.java.transform
label	Java Transform
class	com.eett.exercises.transformations.ui.actions.ROOM2JavaAction
menubarPath	com.eett.exercises.emf.ROOMMenuID/ com.eett.exercises.transformation.MenuID /additions

We do not want the action to be enabled for anything so we will add an enablement to our action to control when it can be invoked. Add an enablement with an objectClass through the action's context menu and set the name to com.eett.exercises.emf.ROOM.Model, which will restrict the enablement to only objects of type Model. Add a new handler for this action that specializes the AbstractTransformationAction we created in the previous exercise.

In the com.eett.exercises.m2m.java plug-in create a class ROOM2JavaTransformation in the com.eett.exercises.m2m.java package. This is the class we will use as the entry point for our transformation. This class will have a protected constructor and a single method for now, transform that takes a ROOM Model and creates Java JModel.

```
/*
 * A class that is the entry point of a M2M transformation
 * written in Java. Its purpose is to transform a
 * com.eett.exercises.emf.ROOM.Model into a
 * org.eclipse.emf.java.JModel.
 *
 * @author Zeligsoft
 *
 */
public class ROOM2JavaTransform {
    /**
     * We only want one of these transform entry point classes
     * created so we manage the creation of the instance and
     * make it accessible through this static field.
     */
    public static final ROOM2JavaTransform INSTANCE =
        new ROOM2JavaTransform();

    /**
     * Constructor is protected since we do not
     * allow others to create instances.
     */
    protected ROOM2JavaTransform() {

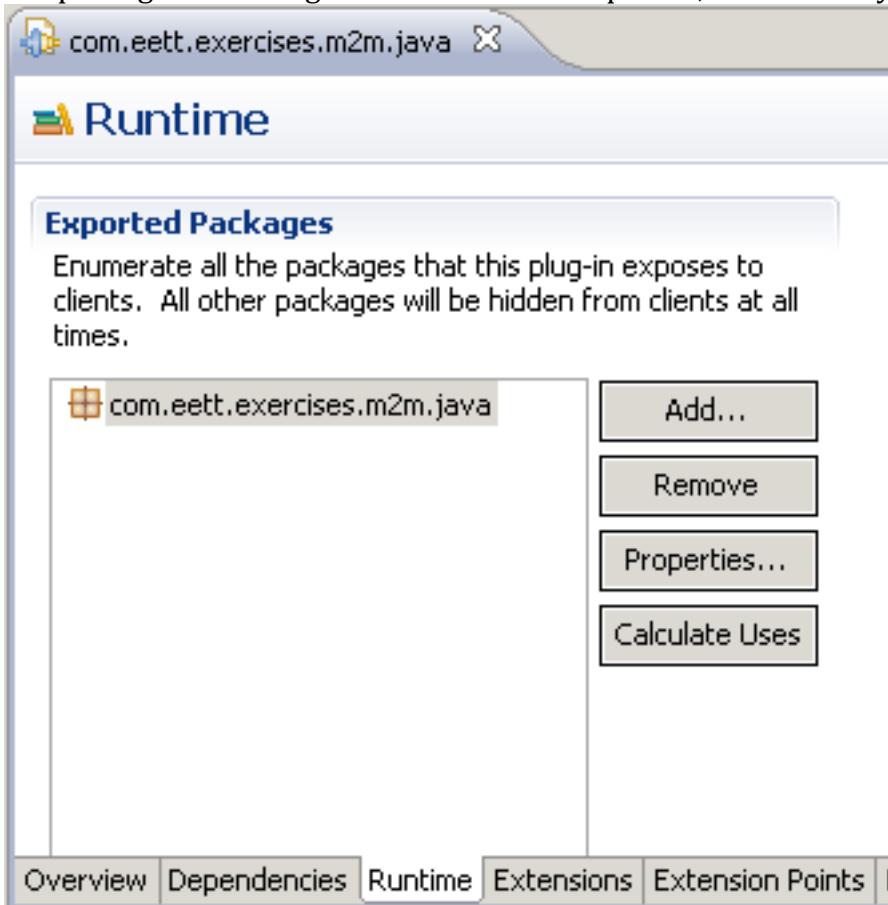
    }

    /**
     * A transformation written in Java that takes as input
     * a ROOM model and transforms it into a Java model.
     */
    public JModel transform(Model roomModel) {
        JModel target = JavaFactory.eINSTANCE.createJModel();

        target.setName(roomModel.getName());

        return target;
    }
}
```

The package containing this class must be exported, in order for you to access it.



We now need to invoke this transformation from the action we have created. This means that the com.eett.exercises.transforms plug-in needs a dependency on the Java M2M plug-in. Then we modify the run method in the action to be the following.

```
@Override
public void run(IAction action) {
    // get the selected object in the active editor
    EObject eobject =
        getSelectedObject();

    // this should be a ROOM model since we restricted
    // action to it but we will test it any way
    if(eobject instanceof Model){
        Model roomModel = (Model) eobject;

        // run the transformation and store the result
        JModel jModel = ROOM2JavaTransform
            .INSTANCE.transform(roomModel);

        // persist the model to a resource that has
        // the same name as the source ROOM model and in
        // the same folder
        if(roomModel.eResource() != null
            && roomModel.eResource().getURI() != null){
            // create the new URI and resource for the target
            // model by removing the source file extension and
```

```
// add the model.java file extension
URI jModelURI = roomModel.eResource()
                .getURI().trimFileExtension();
jModelURI = jModelURI.
                appendFileExtension("model.java");

Resource jModelResource =
        roomModel.eResource().getResourceSet()
                .createResource(jModelURI);
// add the result to the new model
jModelResource.getContents().add(jModel);

try {
    jModelResource.save(null);
} catch (IOException e) {
    e.printStackTrace();
    throw new RuntimeException("Trying to
transform an element and an exception was thrown:\n" +
e.getMessage());
}

// unload and remove the resource from the
// source models resource set
jModelResource.unload();
roomModel.eResource().getResourceSet().
        getResources().remove(jModelResource);
}
else {
    throw new RuntimeException("Trying to
transform an element that is not in a resource.");
}
}
```

Load a runtime workbench and tryout your transformation. Once you are sure that the action is working. Add rules to your m2m.java project that transform an Actor to a JCompilationUnit and a JClass within it. Add a JField to the JClass for each port on the Actor. The JCompilationUnit rule will be called by the transformation we have already created.

4.3 M2M with QVT

Before beginning this exercise it is best to export the projects for the metamodels as plug-ins and restart the Workbench. This requires using File → Export... and exporting them to a folder called dropins in the %RSA_INSTALL%/SDP directory.

In this section we will create a model-to-model transformation using QVT. Start by creating a QVT project com.eett.exercises.m2m.qvt. Add dependencies to the ROOM and Java model projects just as we did in the previous exercise.

Create a transformation ROOM2Java.qvto within the project. The first thing that needs to be done is to define the metamodels that will be used.

```
modeltype roomMM "strict" uses "http://www.eett.com/room/2008";
modeltype javaMM "strict" uses
"http://www.eclipse.org/emf/2002/Java";
```

Now define the transformation that has an in parameter of type roomMM and and out parameter of type javaMM.

```
transformation Room2Java (in room:roomMM, out javaM:javaMM);
```

Now we must define the entry point for the transformation, which will accept a Model element from ROOM and transform it into a target of type JModel. Set the name of the JModel to be the name of the ROOM model.

```
mapping main(in rModel:roomMM::Model, out jModel:javaMM::JModel) {
    init {
        jModel := object JModel{name:= rModel.name};
    }
}
```

Add a rule to the file that maps an Actor or Protocol to a JCompilation and a JClass as a member of the JCompilationUnit. The difference being that a Protocol ends up being an interface.

```

mapping PackageableElement::roomElement2JCompilationUnit()
    : JCompilationUnit
        disjuncts Actor::actor2JCompilationUnit,
                    Protocol::protocol2JCompilationUnit {
}

mapping Actor::actor2JCompilationUnit() : JCompilationUnit {
    name := self.name;
    type += self->map actor2JClass();
}

mapping Protocol::protocol2JCompilationUnit() : JCompilationUnit {
    name := self.name;
    types += self->map protocol2JClass();
}

mapping Protocol::protocol2JClass() : JClass {
    name := self.name;
    interface := true;
}

mapping Actor::actor2JClass() : JClass {
    name := self.name;
}

```

Modify the entry point to iterate through the members of the ROOM model and map. This will use the roomElement2JCompilationUnit rule we wrote.

```

mapping main(in rModel:roomMM::Model, out jModel:javaMM::JModel) {
    init {
        jModel := object JModel{name:= rModel.name};
        elements += rModel.elements->map
            roomElement2JCompilationUnit();
    }
}

```

To test the transformation create or use an existing ROOM model using the editor. Now create a Run Configuration to test the mapping, Run → Run Configurations... and create a new Operational QVT Interpreter, ROOM2Java. Set the Transformation Module to be the transformation we just developed and make sure that the Generate trace file is checked. Finally, specify the target of the transformation.

Now click on the Run button and examine the model that is created and the generated trace file.

Now extend the transformation to consider the ports on Actors and the signals on Protocols.

4.3.1 For the adventurous

Create a similar QVTO transformation that works on RSA-RTE models and thus against profile used by RSA-RTE.

4.4 M2T with xPand

In this exercise we will map a JModel into Java source. To do this we will use xPand and a workflow. Start by creating a new openArchitectureWare Project, com.eett.exercises.m2t.xpand. Make sure that the Create a sample is checked.

Now modify the MANIFEST.MF to have a dependency on org.eclipse.emf.java.

We won't need the files in src/metamodel so they can be deleted. The xPand template should also be renamed to GenerateJava.xpt and an import for the Java metamodel. Which will allow us to build an M2T for our Java models.

```
«IMPORT java»
```

We now need to define the transformation entry point, which is defined for a JModel.

```
«DEFINE main FOR JModel»  
«ENDDEFINE»
```

Add a rule for writing each JCompilationUnit to a file with the .java extension.

```
«DEFINE writeJCompilationUnit FOR JCompilationUnit»  
    «IF this.name != null»  
        «FILE this.name + ".java"»  
  
        «ENDIF»  
    «ENDIF»  
«ENDDEFINE»
```

Modify the workflow file that was generated. So that it is named generateJava.oaw and that it works with our models and metamodels.

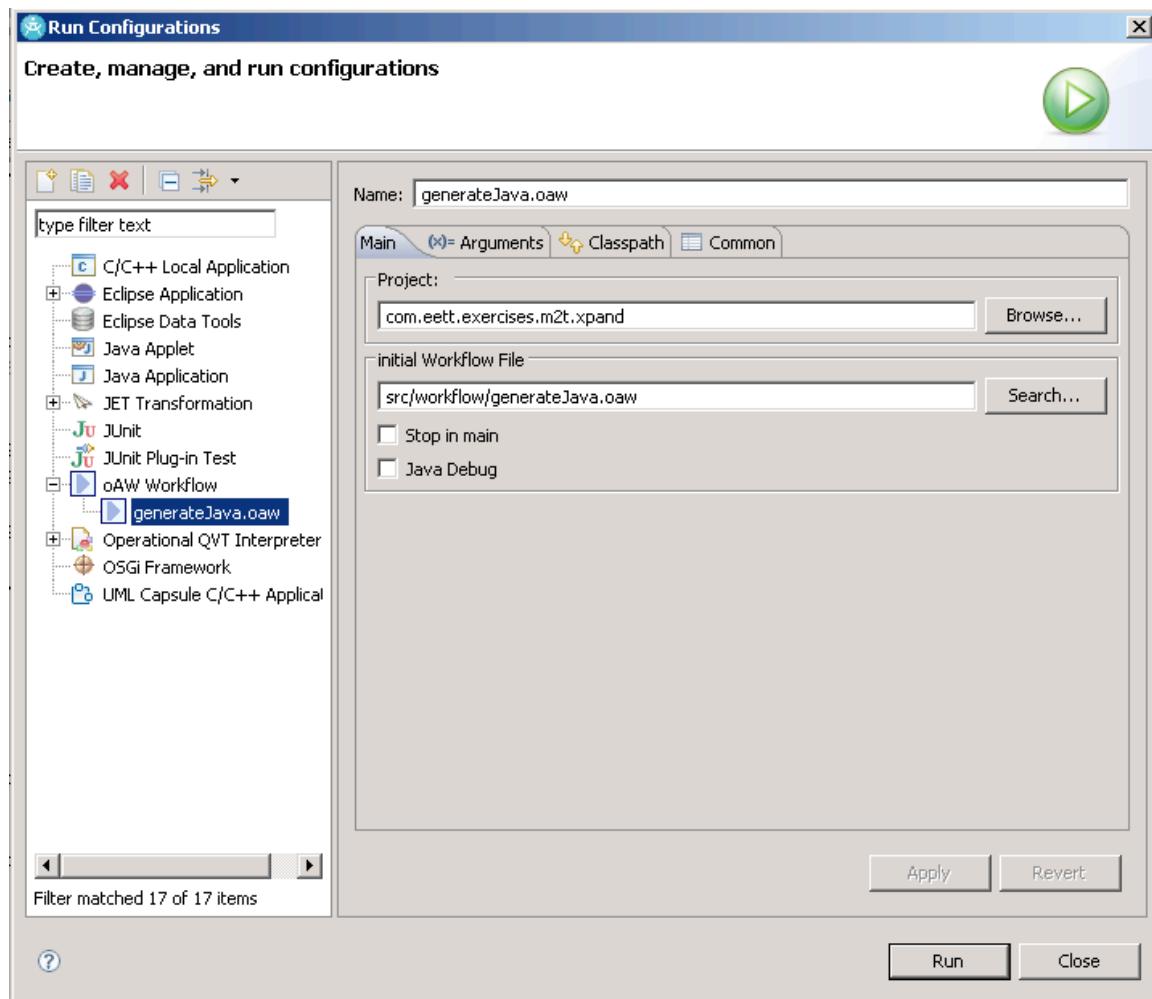
Change the model property to be the model you will test with. Note that you can also define these properties in a separate file.

```
<property name='baseDir' value='.' />  
<property file='${baseDir}/my.properties' />
```

Change <component class="org.openarchitectureware.xpand2.Generator"> element to be as shown below. This points to our Java metamodel and the transformation file that we renamed. The last part is a post-processing step that makes the generated Java look good.

```
<!-- generate code -->
<component class="org.openarchitectureware.xpand2.Generator">
    <metaModel id="javaMM" class="oaw.type.emf.EmfMetaModel">
        <metaModelPackage
            value="org.eclipse.emf.java.JavaPackage" />
    </metaModel>
    <expand
        value="template::GenerateJava::main FOR model" />
    <outlet path="${src-gen}" >
        <postprocessor
            class="org.openarchitectureware.xpand2.output.JavaBeautifier" />
    </outlet>
</component>
```

You can now create an oaw Workflow Run Configuration to execute the transformation. Select the workflow that we modified in the previous step as your initial Workflow File using the Search... button. Apply and run the workflow.



There should be a src-gen folder in your project with the generated files in it.

Augment the transformation to include generating the types contained in the JCompilationUnit, which should be JClass. This can be done by adding a new rule to the transformation and calling it with the EXPAND FOREACH expression.

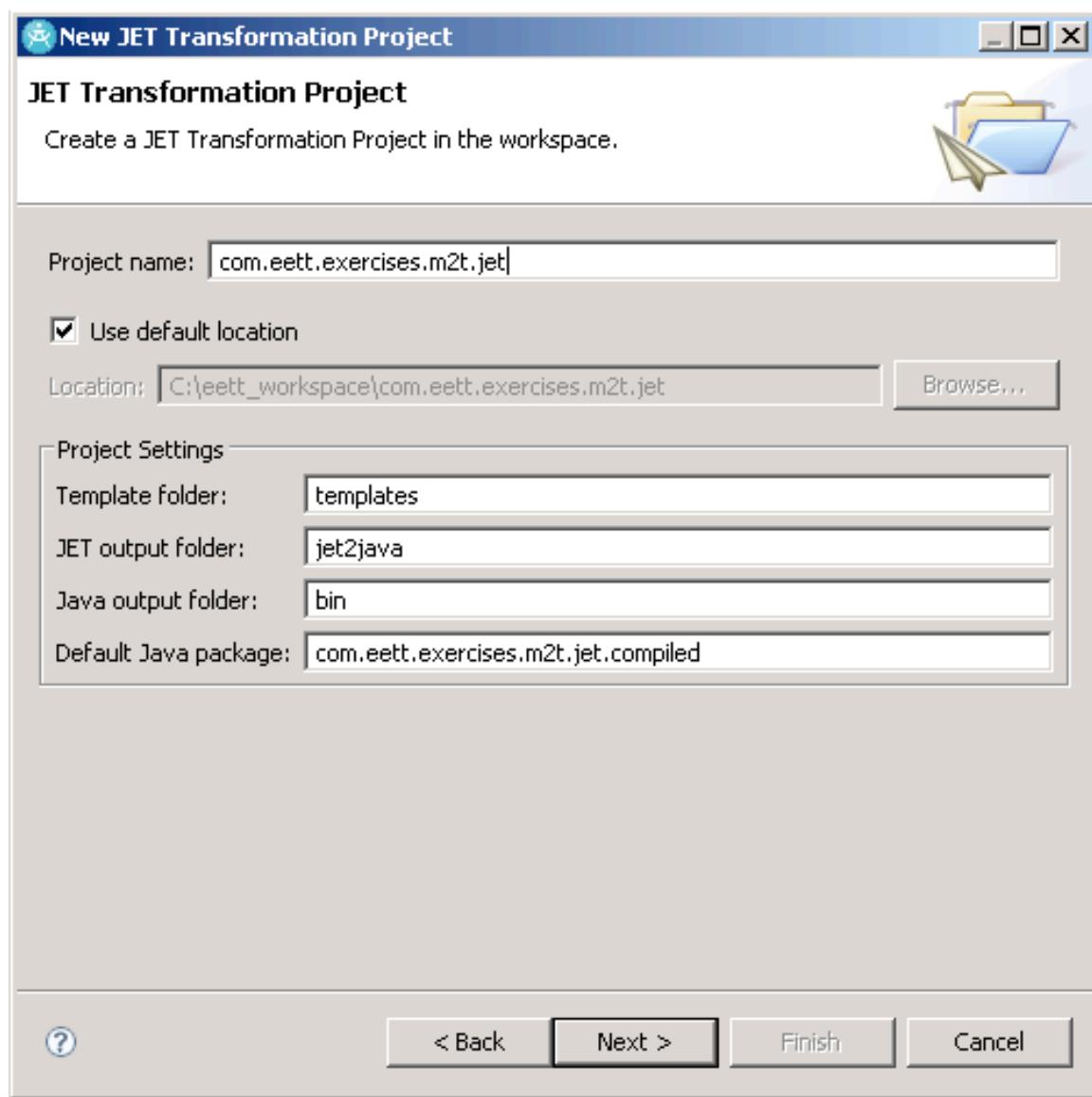
Re-run the transformation and observe the impact of your new rules.

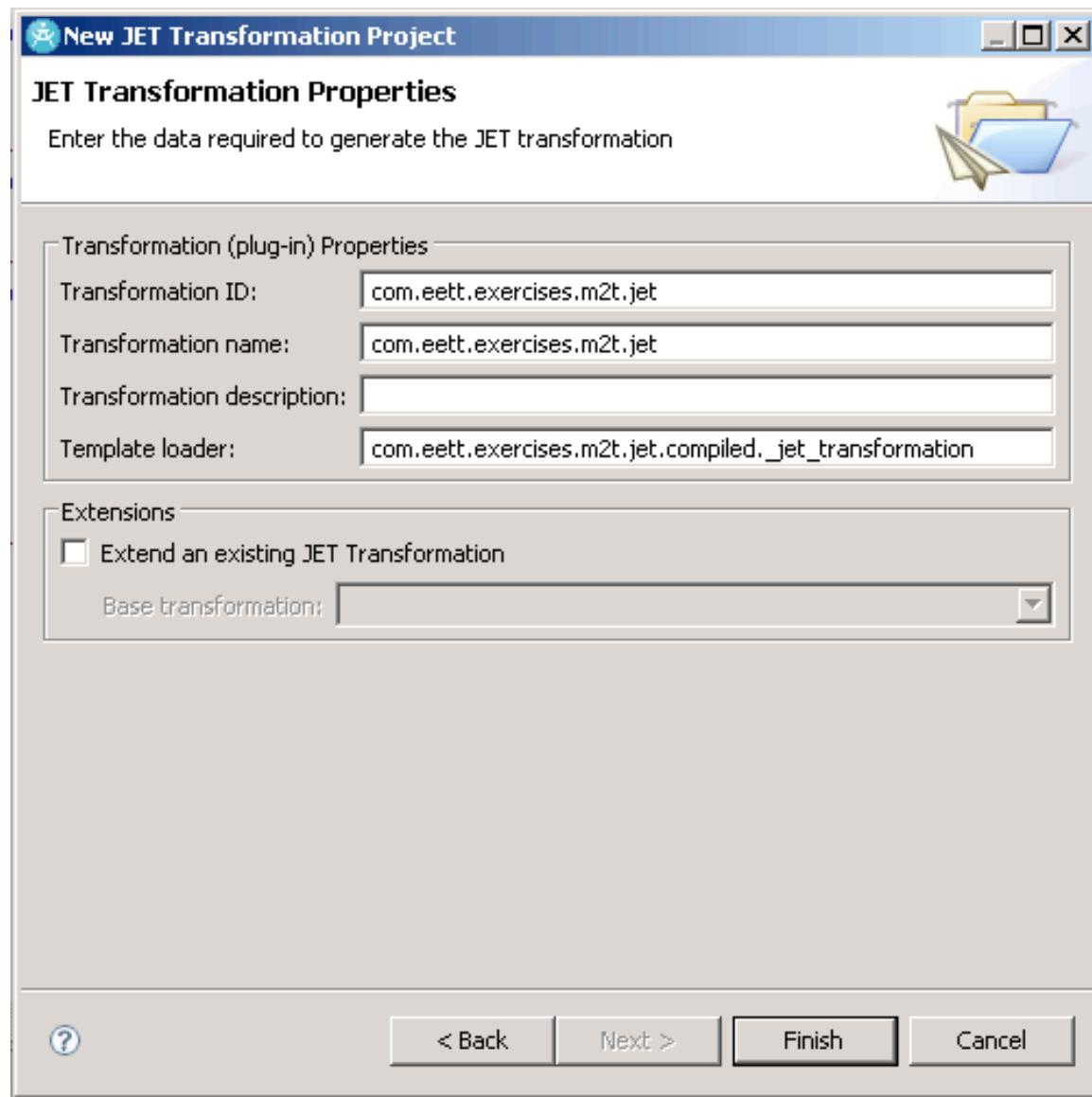
For the adventurous

- Extend the transformation even further to include JField and JOperation for a JClass.

4.5 M2T with JET

In this exercise we will map a JModel into Java source. To do this we will use JET. Start by creating a new EMFT JET Transformation Project, com.eett.exercises.m2t.jet. Keep the default values for the other fields in the project wizard.

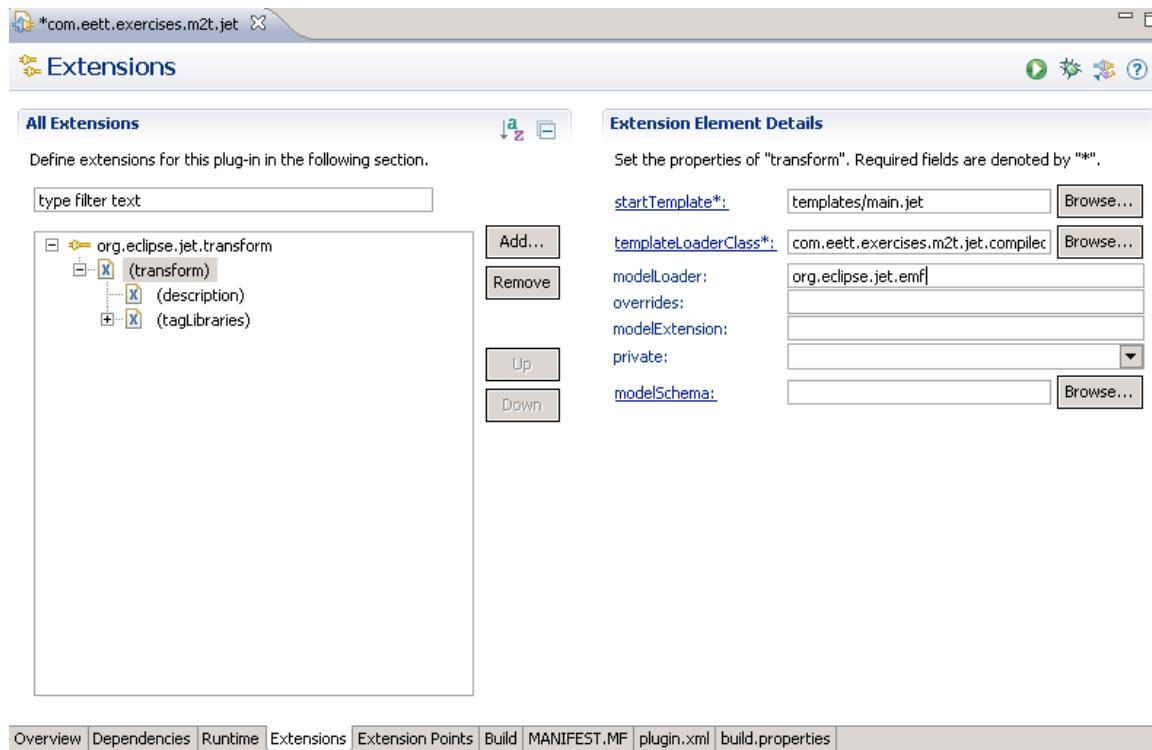




Now modify the MANIFEST.MF to have a dependency on org.eclipse.emf.java.

The screenshot shows the Zeligsoft Eclipse Transformation interface. The top navigation bar includes 'File', 'Edit', 'Transform', 'Help', and 'About'. Below the navigation bar, the title 'Zeligsoft Eclipse Transformation' is displayed. The main content area features a large 'Dependencies' tab, which is currently selected. The 'Dependencies' tab has a sub-section titled 'Required Plug-ins' with a note: 'Specify the list of plug-ins required for the operation of this plug-in.' A list box contains two entries: 'org.eclipse.jet' and 'org.eclipse.emf.java (2.4.0)'. To the right of the list box are five buttons: 'Add...', 'Remove', 'Up', 'Down', and 'Properties...'. At the bottom of the list box, the text 'Total: 2' is visible. Below the 'Dependencies' tab, there is another tab labeled 'Automated Management of Dependencies' with a sub-section titled 'MANIFI'. The 'MANIFI' section has tabs for 'Overview', 'Dependencies', 'Runtime', 'Extensions', 'Extension Points', 'Build', and 'MANIFI'. The 'Dependencies' tab is currently selected in this section.

We also need to modify the org.eclipse.jet.transform extension to specify an EMF model loader. Change to the Extensions page of the Plug-in Editor and select the (transform) element of the extension. Change its modelLoader field to be org.eclipse.jet.emf.



The JET template is named main.jet and can be found in the templates folder. We will modify it to use our metamodel. At the same time define the entry point, which sets a variable jModel, to be the root of the JModel passed in.

```
<%@jet imports="org.eclipse.emf.java.*" %>
<%@taglib prefix="ws" id="org.eclipse.jet.workspaceTags" %>
<%-- Main entry point for com.eett.exercises.m2t.jet --%>

<%-- Define variables the root element in the model --%>
<c:setVariable select="/" var="jModel"/>
```

Now we will add an iterator over the JCompilationUnits (the elements attribute) in the model and write them to a file in a src-gen directory in the com.eett.exercises.m2t.jet (Note this could be a parameter to the template).

```

<c:iterate select="$jModel/elements" var="jCompilationUnit">
    <ws:project name="com.eett.exercises.m2t.jet">
        <ws:folder path="src-gen">
            <java:package name="com.eett.exercises.emf.m2t.jet">
                <java:class name="${jCompilationUnit/@name}">
                    template="templates/JCompilationUnit.java.jet"/>
                </java:class>
            </java:package>
        </ws:folder>
    </ws:project>
</c:iterate>

```

Notice that this calls another template JCompilationUnit.java.jet that we have not created yet. So add a new JET transformation, JCompilationUnit.java.jet to the templates folder. Set the contents of the file to be the following.

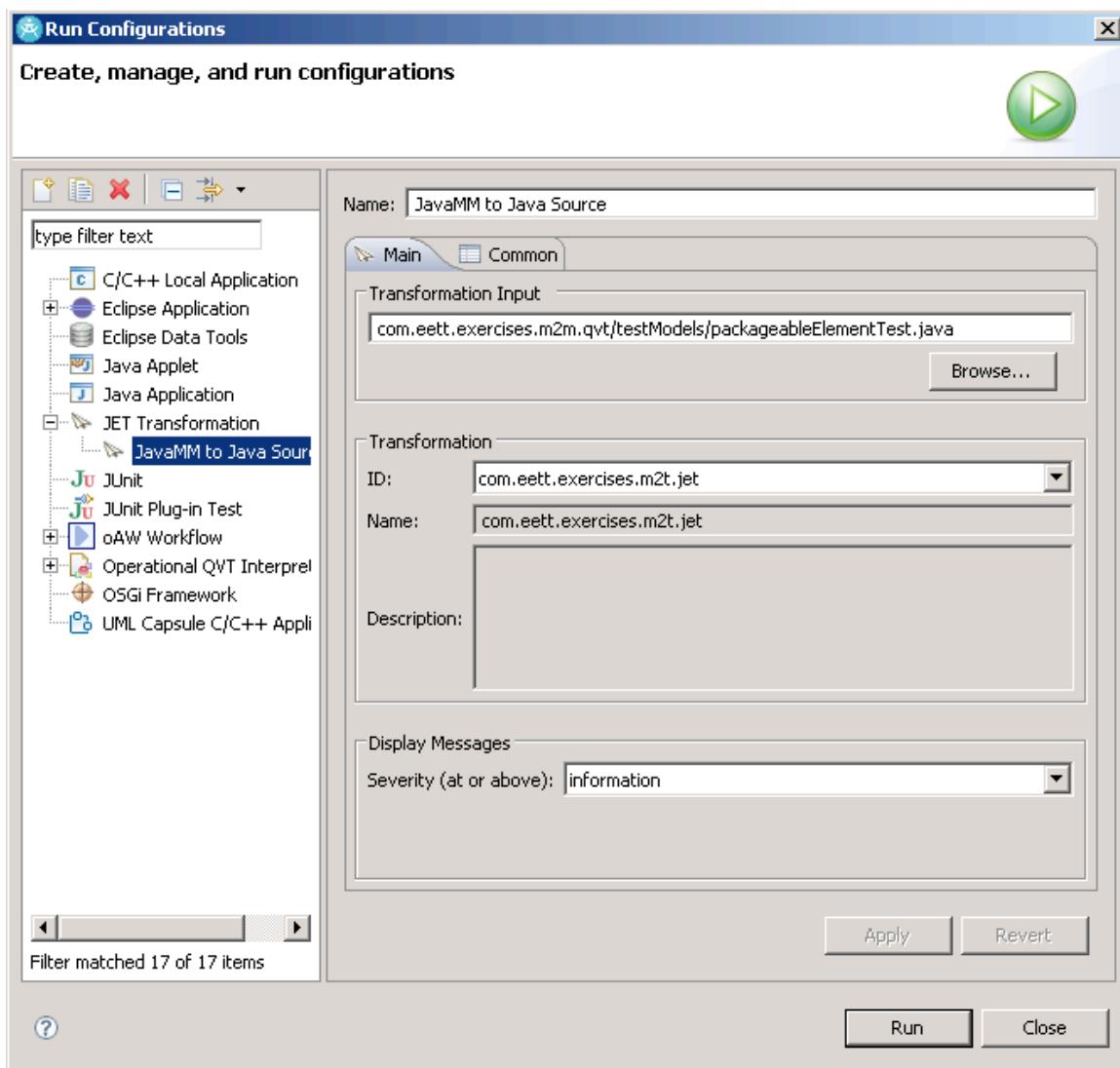
```

<%@taglib prefix="ws" id="org.eclipse.jet.workspaceTags" %>
<%@taglib prefix="c" id="org.eclipse.jet.controlTags" %>

<c:iterate select="$jCompilationUnit/types" var="jClass">
public class <c:get select="$jClass/@name"/> {
}
</c:iterate>

```

You can now create a JET Transformation Run Configuration to execute the transformation. But before doing so create a new src folder called src-gen in the project. Select the transformation we just created in the ID field. Also select a Java model to test our transformation as the Transformation Input. Apply and run the transformation. The files should be generated to the src-gen folder.



Augment the transformation to include generating the types contained in the JCompilationUnit, which should be JClass. Remember that a JClass has a field that indicates whether it is an interface. Hint use the c:if tag.

Re-run the transformation and observe the impact of your new rules.

For the adventurous

- Extend the transformation even further to include JField and JOperation for a JClass.

5 Validation Exercises

5.1 Working with Validation Framework

This exercise will perform model validation on our ROOM metamodel using the EMF validation framework. More specifically, it will create an EValidator implementation that delegates to the validation framework, to provide user-demand “batch mode” validation from an EMF editor.

5.2 Constructing a Batch Model Constraint

A model constraint is a subclass of the AbstractModelConstraint that overrides the validate() method. The validate method has the task of taking the input from the validation context returning either a ctx.createSuccessStatus() or ctx.createFailureStatus().

Create a new Class NonEmptyNamesConstraint that specializes AbstractModelConstraint, in the com.eett.exercises.validation package.

Override the validate method and add the following logic to it.

```
public IStatus validate(IValidationContext ctx) {
    EObject eObj = ctx.getTarget();
    EMFEventType eType = ctx.getEventType();

    // In the case of batch mode and not live mode.
    if (eType == EMFEventType.NULL) {
        String name = null;
        if (eObj instanceof NamedElement) {
            name = ((NamedElement)eObj).getName();
        }
        if (name == null || name.length() == 0) {
            return ctx.
                createFailureStatus(new Object[]
                    {eObj.eClass().getName()});
        }
    }
    return ctx.createSuccessStatus();
}
```

We now need to add the constraint provider extension to our Plug-in Manifest. Open the plug-in editor and add an org.eclipse.emf.validation.constraintProviders extension. Add a category to the extension and set its properties to.

name	ROOM Constraints
id	com.eett.exercises.validation.constraints

Add a constraintProvider to the extension setting the package element that was created to <http://www.eett.com/room/2008>.

Add a constraints element to the constraintProvider and set its categories to com.eett.exercises.validation.constraints. Now add a constraint element and set its properties to.

id	com.eett.exercises.validation.NonEmptyNamesConstraint
name	Non-empty names
mode	Batch
severity	ERROR
lang	Java
class	com.eett.exercises.validation.NonEmptyNamesConstraint
statusCode	1

Modify the message element to have the following:

```
A {0} has been found to have no name specified.
```

Add three target elements to the constraint for NamedElement classes. These classes are qualified by the EPackage we specified in the package element previously.

We now need to add a context in which our constraint will execute. The primary reason for doing this is avoid constraints from different vendors being executed in the wrong context. Start by adding another extension for org.eclipse.emf.validation.constraintBindings. and add a clientContext element to it. Setting its id to com.eett.exercises.validation.roomContext and default to false.

Add a selector element to the clientContext setting its class to be com.eett.exercises.validation.ValidationDelegateClientSelector. Then create a new class in the com.eett.exercises.validation package called ValidationDelegateClientSelector and add the following to the file.

```
//NOTE: This is _NOT_ a recommended approach to writing a client selector.  
//Suggested approaches:  
// -Check the resource of the EObject either by identity or by URI  
//   as long as this resource is somehow unique to this application  
// -Check the identity of the resource set to ensure that it is some  
//   private object  
// -Check the identity of the EObject itself to see if it belongs to  
//   some private collection  
// -Check the EClass of the EObject but only if the metamodel is private  
//   to this application and will not be used by other contexts  
public class ValidationDelegateClientSelector  
    implements IClientSelector {  
  
    public static boolean running = false;  
  
    public boolean selects(Object object) {  
        return running;  
    }  
}
```

Finally add a binding element to the constraintBindings extension settings its id to be com.eett.exercises.validation.roomContext the id of the context we defined in the previous step, and the category to com.eett.exercises.validation.constraints, which we defined on the constrainProviders extension.

Your plugin.xml file should now look like the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
    <extension
        point="org.eclipse.emf.validation.constraintProviders">
        <category
            id="com.eett.exercises.validation.constraints"
            name="ROOM Constraints">
        </category>
        <constraintProvider
            cache="true">
            <package
                namespaceUri="http://www.eett.com/room/2008">
            </package>
            <constraints>
                <constraint

id="com.eett.exercises.validation.NonEmptyNamesConstraint"
                    lang="Java"
                    mode="Batch"
                    name="Non-empty names"
                    severity="ERROR"
                    statusCode="1">
                    <message>
                        message body text
                    </message>
                    <target
                        class="NamedElement">
                    </target>
                </constraint>
            </constraints>
        </constraintProvider>
    </extension>
    <extension
        point="org.eclipse.emf.validation.constraintBindings">
        <clientContext
            default="false"
            id="com.eett.exercises.validation.roomContext">
            <selector
                class="com.eett.exercises.validation.ValidationDelegateClientSelector
">
                </selector>
            </clientContext>
            <binding
                category="com.eett.exercises.validation.constraints"
                context="com.eett.exercises.validation.roomContext">
            </binding>
        </extension>
    </plugin>
```

3.3 Executing the constraint

To be able to execute the constraint add an action to the ROOM Editor to invoke constraints. Refer to the Query and Transformation for guidance on doing this.

The code to execute the is the following:

```
ValidationDelegateClientSelector.running = true;

IBatchValidator validator =
(IBatchValidator)ModelValidationService.getInstance()
    .newValidator(EvaluationMode.BATCH);
validator.setIncludeLiveConstraints(true);

IStatus status = validator.validate(selectedEObjects);
ValidationDelegateClientSelector.running = false;
```

The first part of this code snippet enables the latch so that the validation service will determine that the provided EObjects belong to our client context. We requested a batch validation and asked that the batch validator include live validation constraints because live validation constraints are often written to handle the batch validation case. Finally, we validate the selected EObjects and are given back the status of the validation.