



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

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

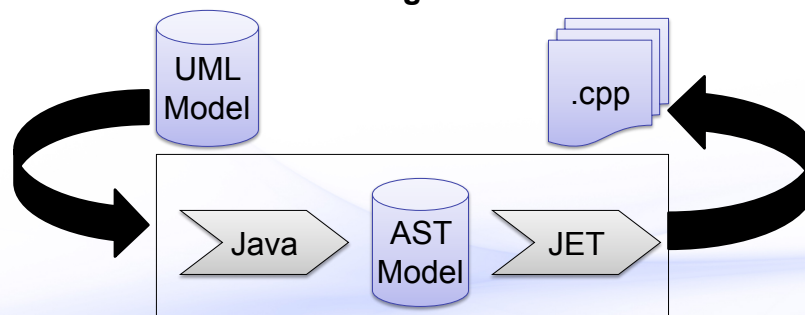
- Eclipse Technologies
 - Java
 - M2M Project - **QVT**, xTend and ATL
- RSA-RTE Technologies
 - Rational Transformation Engine

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

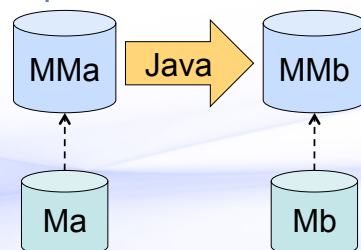
Model to Model Transformations

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

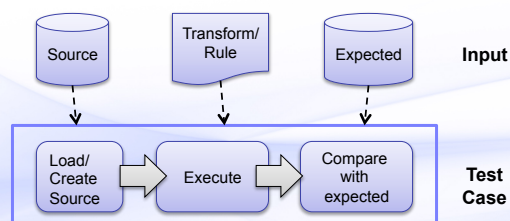


Developing a Transformation

- 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

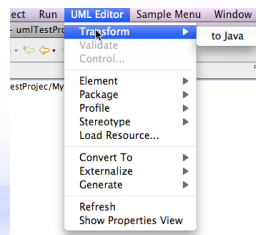
- Use workbench Java execution capabilities
 - We can use the Run As → Java Application
 - We can debug using Debug As → Java Application
 - To pass transformation parameter values
 - Use command line
 - Build a user interface
- Integrate with workbench
 - Provide an **action** to invoke from editor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

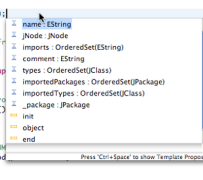
Developing QVTO Transformations

- 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

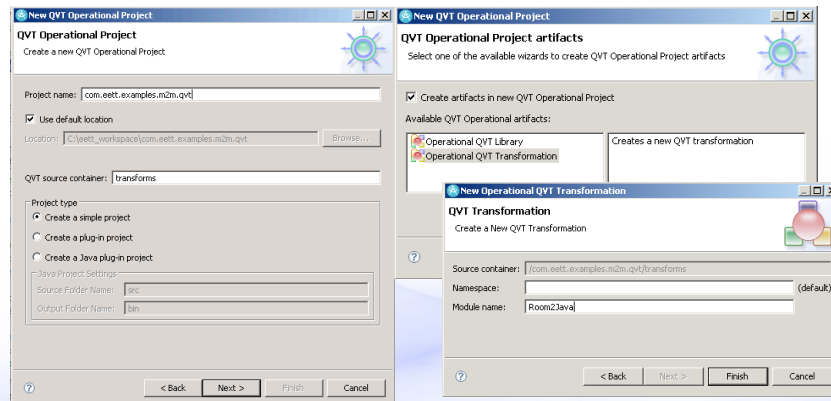
```

1::modeltype UML "strict" uses "http://www.eclipse.org/uml2/2.1.8/uml";
2::modeltype JAVA "strict" uses "http://www.eclipse.org/emf/2002/java";
3
4 transformation Room2Java(in source:UML, out target:JAVA);
5
6::main() {
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::toModel() : Sequence(JAVA::CompilationUnit) {
14  init {
15    result += self.packageElement[UML::Class]-->map toCompilationUnit();
16  }
17 }
18
19 -- Create a compilation unit from a UML class
20::mapping UML::Class::toCompilationUnit() : JCompilationUnit {
21  name := self.name;
22  types += self.map toClass();
23 }
24
25 -- Create a Java class object from
26::mapping UML::Class::toClass() : JClass {
27  name := self.name;
28  fields += self.attribute-->map
29 }
30
31 -- Create a Java field object from
32::mapping UML::Property::toField() : JField {
33  name := self.name;
34 }
35
36 -- Create a Java method from a UML
37::mapping UML::Operation::toMethod() : JMethod {
38  name := self.name;
39 }

```



Developing QVTO Transformations



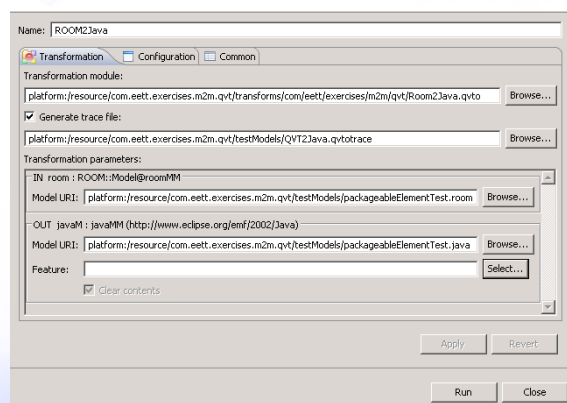
Testing QVTO Transformations

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

Executing QVTO Transformations

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

Executing QVTO Transformations



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

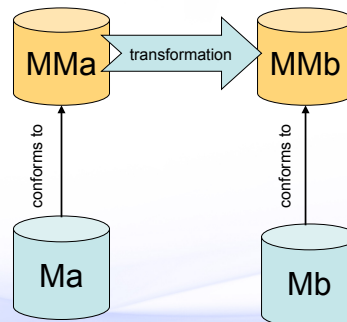
QVTO - Transformation

- 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 readwrite 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

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



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

### ▪ Query example

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

Implementation details

## 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

## 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



## 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

## 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

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

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

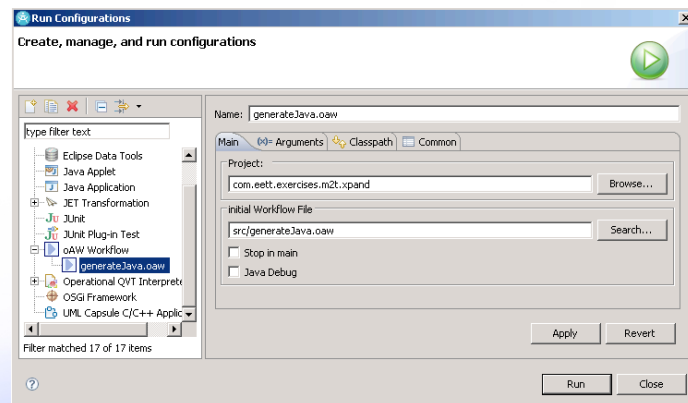
- Editor
 - Syntax highlighting
 - Code completion
- Model navigation
 - Java based syntax for model navigation
 - Has operators for working with collections
- Focus on transformation logic
 - Execution infrastructure left to the xPand runtime

```
1 Generations.qpt.12
2
3 <IMPORT java:
4 <IMPORT eclipse:
5 <ENDIMPORT
6
7 <DEFINE null FOR Object:
8 <ENDDEFINE
9
10 <DEFINE null FOR JModel:
11 <EXTEND writeCompilationUnit <FOR EACH this.elements>
12 <ENDDEFINE
13
14 <DEFINE writeCompilationUnit FOR JCompilationUnit:
15 <IF this.name != null>
16 <FILE this.name + ".java">
17 <ENDIF
18
19 <FOR EACH this.importedPackage AS i>
20 <IMPORT i.name>
21 <ENDFOR EACH>
22
23 <FOR EACH this.importedType AS i>
24 <IMPORT i.name>
25 <ENDFOR EACH>
26
27 <FOR EACH this.import AS i>
28 <IMPORT i>
29 <ENDFOR EACH>
```

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

Executing an xPand Transformation



Integrating with Workbench

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

Integrating with RSA-RTE

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

xPand - Metamodels

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

xPand - Extensions

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

xPand - Templates

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

xPand – 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»
«ENDFILE»
«ENDIF»
«ENDDEFINE»
```

Implementation details

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»  
«ENDDDEFINE»
```

- Iterate over elements

```
«DEFINE writeJField FOR JField»  
  «EXPAND writeFieldType FOR this.type»  
«ENDDDEFINE»
```

- Iterate over elements

```
«DEFINE main FOR JModel»  
  «EXPAND writeJCompilationUnit FOREACH this.elements»  
«ENDDDEFINE»
```

Implementation details

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

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

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

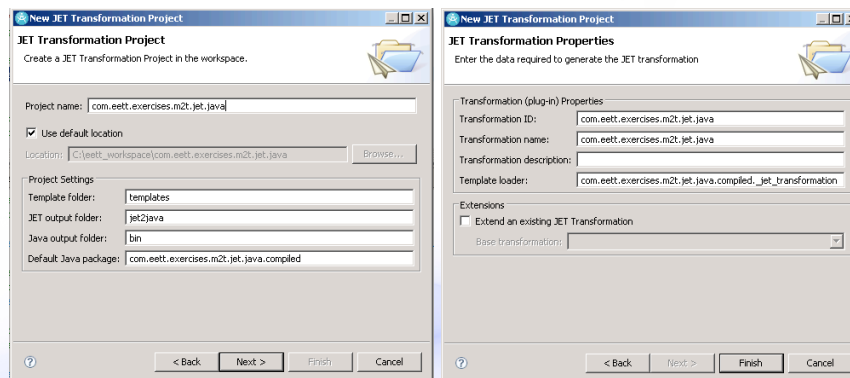
JET Overview

- 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

Developing a JET Transformation



New JET Transformation Project
JET Transformation Project
Create a JET Transformation Project in the workspace.

Project name:

☒ Use default location
Location:

Project Settings

Template folder:

JET output folder:

Java output folder:

Default Java package:

New JET Transformation Project
JET Transformation Properties
Enter the data required to generate the JET transformation

Transformation (plug-in) Properties

Transformation ID:

Transformation name:

Transformation description:

Template loader:

Extensions

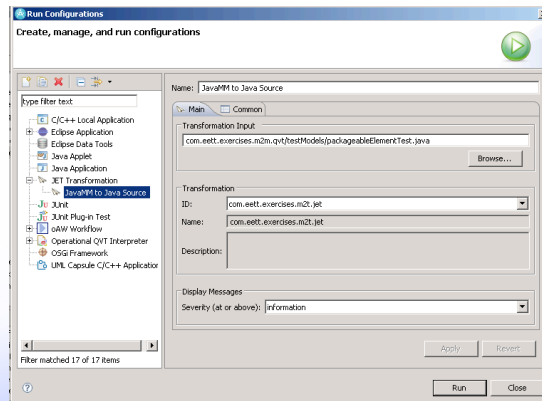
☐ Extend an existing JET Transformation

Base transformation:

Executing a JET Transformation

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

Executing a JET Transformation



Integrating with Workbench

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

Integrating with RSA-RTE

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

JET Concepts Overview

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

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

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

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:lc, f:uc
 - Convert the contents to lowercase/uppercase
- f:replaceAll
 - Replace all instances of a value within the contents to a new value
- f:xpath
 - Evaluate an XPath expression and writes it result

Implementation details

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 specified 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

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 it 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

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="<id>" passVariables="<variable list>" />`
 - `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

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



## 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 model,
    - manipulate Java model,
    - write Java model to source, and
    - format generated Java code
- In the process of becoming an Eclipse project called Model Workflow Engine (MWE)

## oAW Workflow Project?

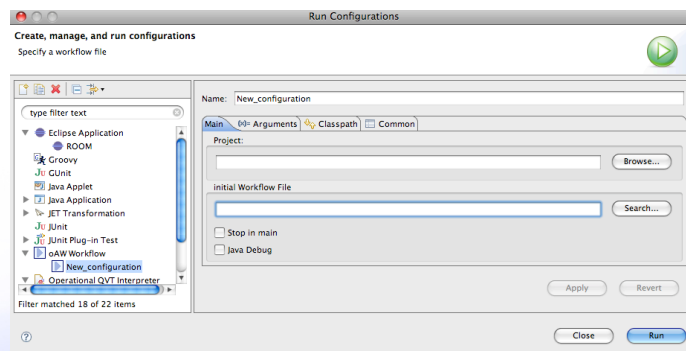
- Out of the box the oaw Workflow Project consists of
  - A workflow execution engine
  - Workflow components for reading and writing EMF models
  - API for integration with oAW Workflow
  - Workbench integration
    - Editor, Run as..., Debug as..., and ANT

## More oAW Workflow Project?

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

Executing an oAW Workflow

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



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

