



IBM Rational Software Conference 2009
As Real as It Gets!



Transformation Design Patterns

Sandeep Katoch
sakatoch@in.ibm.com

Rational. software

NMAC01

Agenda

■ Quick Review

- ▶ Model Driven Development
- ▶ Rational MDD Platform
- ▶ Model to Model Transformations
- ▶ Designing Model to Model Transformations with Mappings

■ Transformation Design Patterns

- ▶ Reference Filter
- ▶ One to Many
- ▶ Filling the Gaps
- ▶ Ask the User
- ▶ Copy a Reference
- ▶ Chain

Model-Driven Development

- What is model-driven development (MDD) ?
 - ▶ Technical Definition:
 - Development with models as the primary artifacts from which efficient implementations are generated by the application of transformations.
 - Models in the application domain are the primary focus when developing new software components.
 - Code, executables and other target domain artifacts are generated by using transformations that are designed by using input from modeling experts and target domain experts.
 - ▶ Another way to look at it?
 - When applied properly, MDD leverages abstractions to accelerate and improve the quality of individual and group software development.
 - To reap these benefits, use model-based technologies to provide abstractions and accelerators for solution delivery.



MDD Approach – Few Key Ideas

■ Abstraction

▶ Focus on relevant details

- In MDD, we use abstraction to enable us to work with a logical view of our application, focusing on the functionality of the system without being concerned with implementation details

▶ Abstraction can be used to model applications at

- Different Levels
 - Analysis / Design / Implementation
- Perspectives
 - Security / User Interface

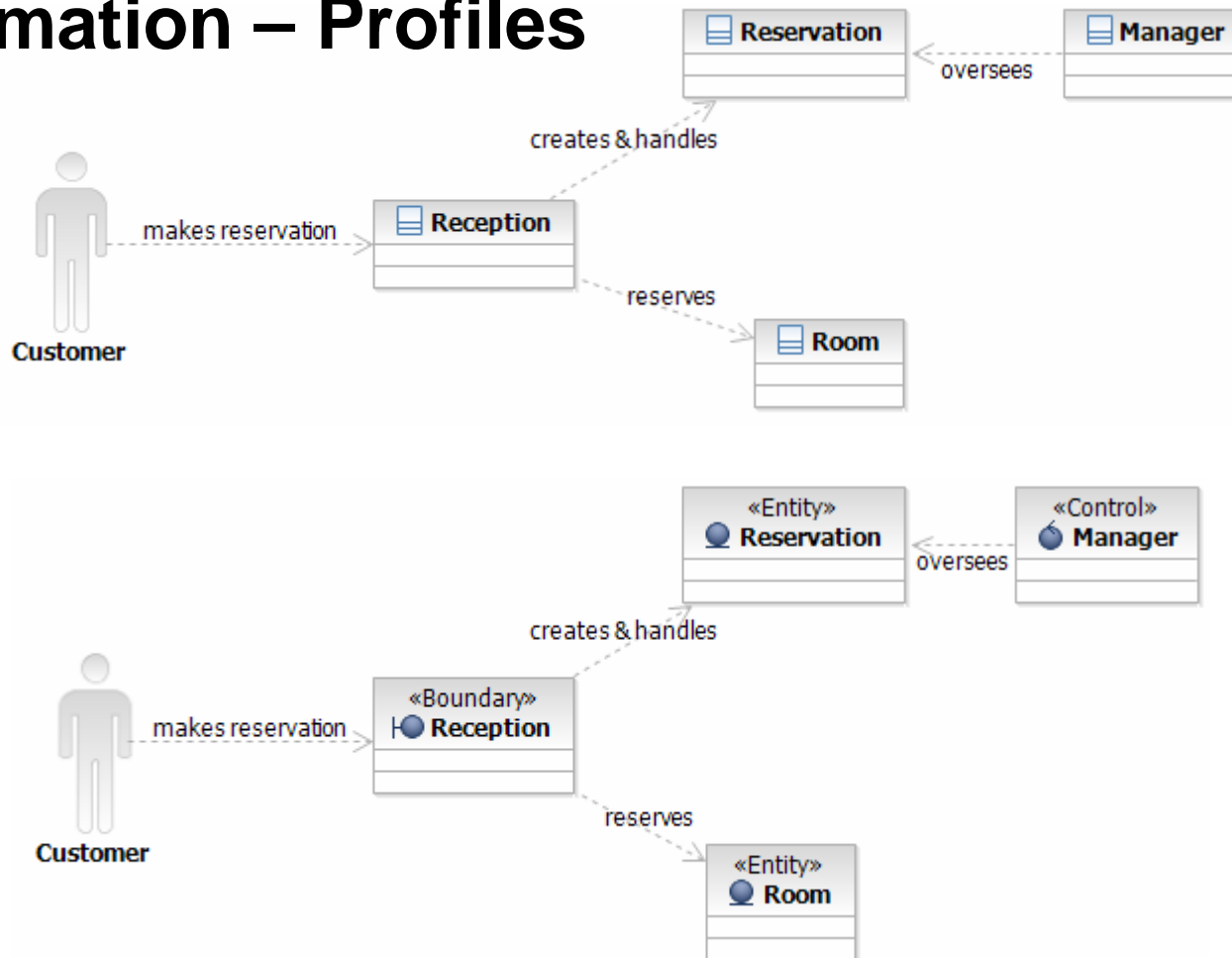
MDD Approach – Few Key Ideas

■ Automation

- ▶ Modeling is a valuable technique in itself, but manually synchronizing models and code is error prone and time consuming
- ▶ Automation is the main characteristic that distinguishes MDD from other approaches that use modeling
- ▶ MDD is concerned with automating the development process so that any artifact, which can be derived from information in a model, is generated

MDD Approach – Few Key Ideas

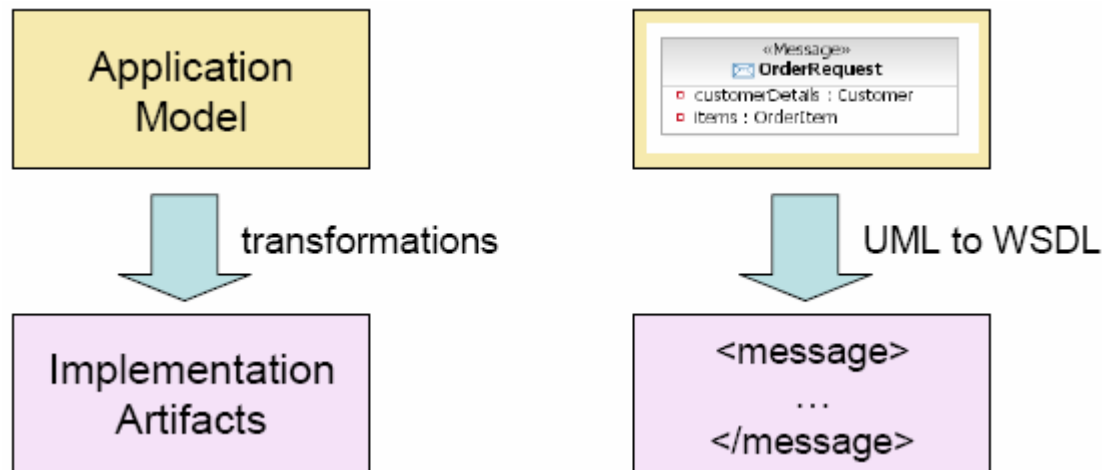
Automation – Profiles



MDD Approach – Few Key Ideas

■ Automation - Transformations

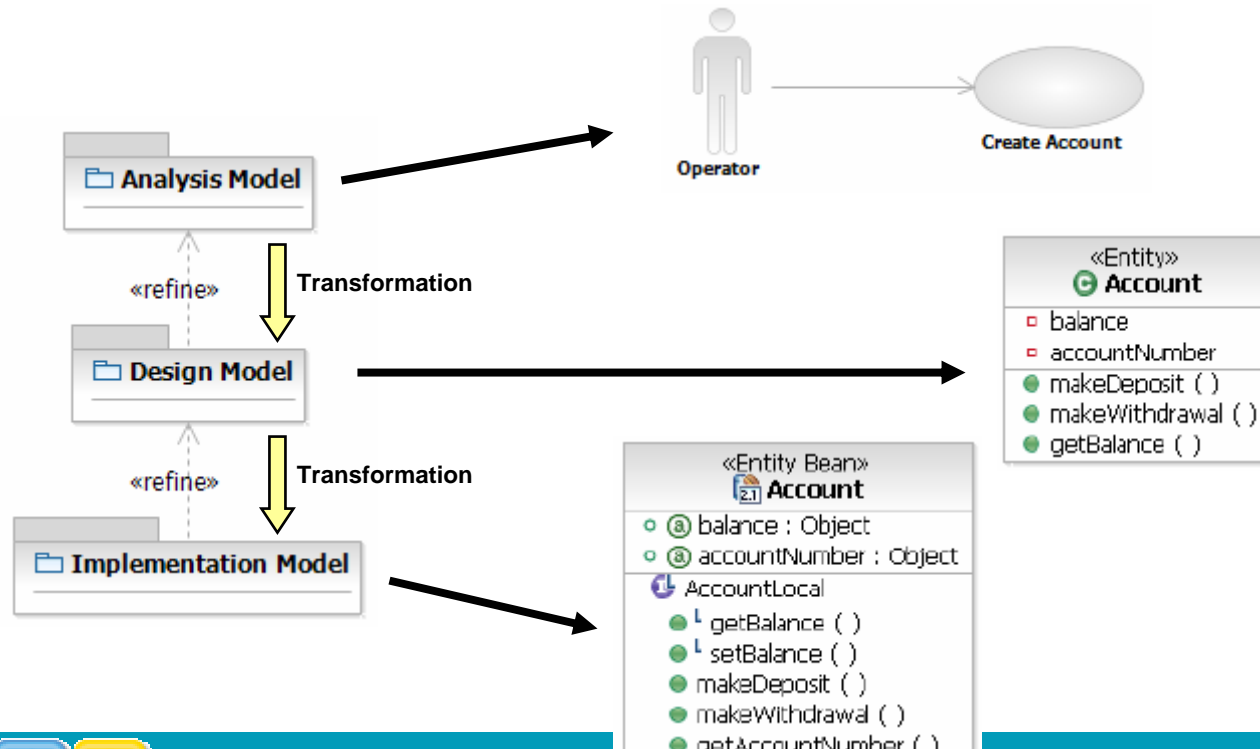
- ▶ Automate generation of Artifacts from Models
- ▶ This includes the generation of code and also the generation of more detailed models, for example generating a design model from an analysis model



MDD Approach – Few Key Ideas

■ Transformations - Layered Modeling

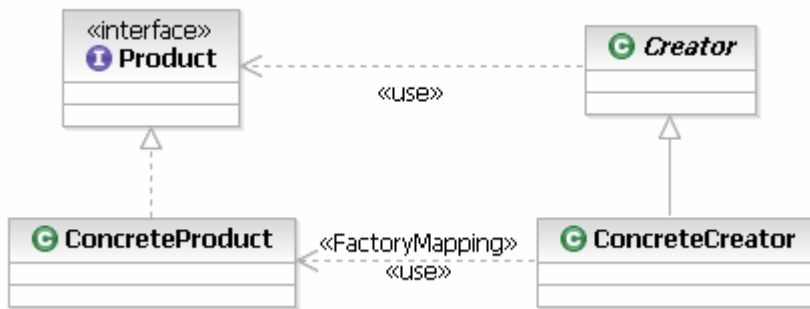
- ▶ Each successive layer adds further detail to the solution, answering questions that were left open in the layer above and constraining the implementation of the application



MDD Approach – Few Key Ideas

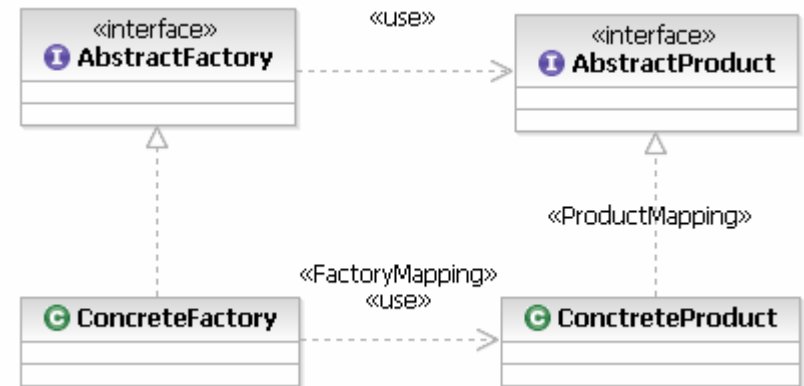
Automation - Patterns

- ▶ Best practice approaches to common design problems
- ▶ Automate the creation and the modification of model elements within a model to apply a given software pattern



Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate.

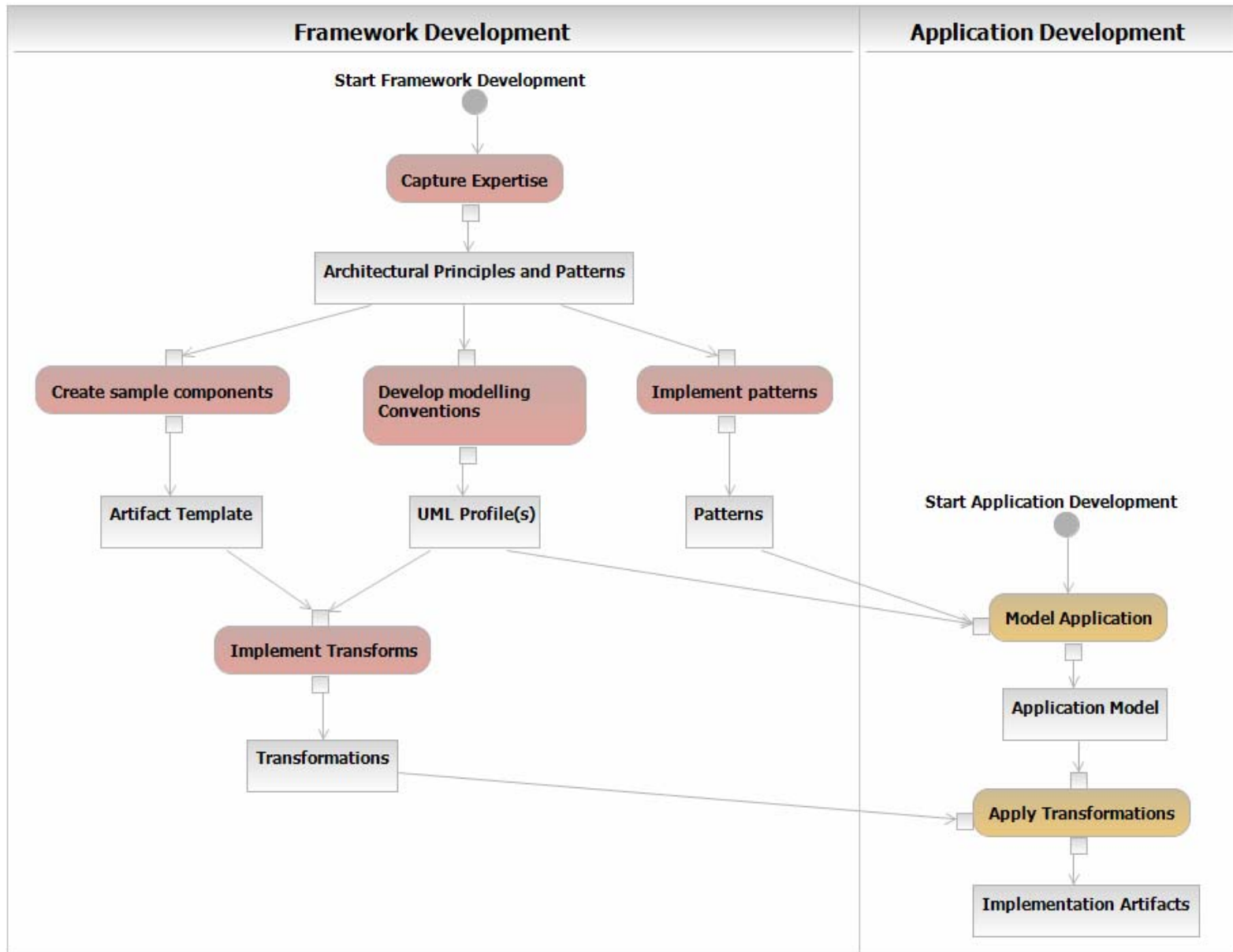


Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

MDD Approach – Process

- **There are the two distinct types of activities in the MDD process**
 - ▶ **Framework Development**
 - Develop Profiles, Patterns, Transformations
 - ▶ **Application Development**
 - Modeling the Application using Profiles & Patterns
 - Apply Transformations to generate artifacts
- **There is no magic to MDD**
 - ▶ Someone must come up with a set of modeling conventions that are suitable for the software under development
 - ▶ Someone must also develop transformations that can automate the generation of code from models that follow these conventions



Rational MDD Platform

- Rational Modeling tools provide a set of flexible technologies that can be used to implement “MDD”
 - ▶ Set of supporting technologies for managing and manipulating models
 - ▶ Support authoring of various forms of patterns
 - Transformation framework
 - Model2Model transformations
 - Model2Text transformations
 - Model operative (in place) pattern (MoP) authoring for UML models
 - ▶ Support authoring of various forms of domain specific modeling languages
 - UML-based DSL tool authoring
 - Custom DSL tool authoring
 - ▶ Future technologies for executable models

- These technologies can be used together in various ways to address a huge variety of problems



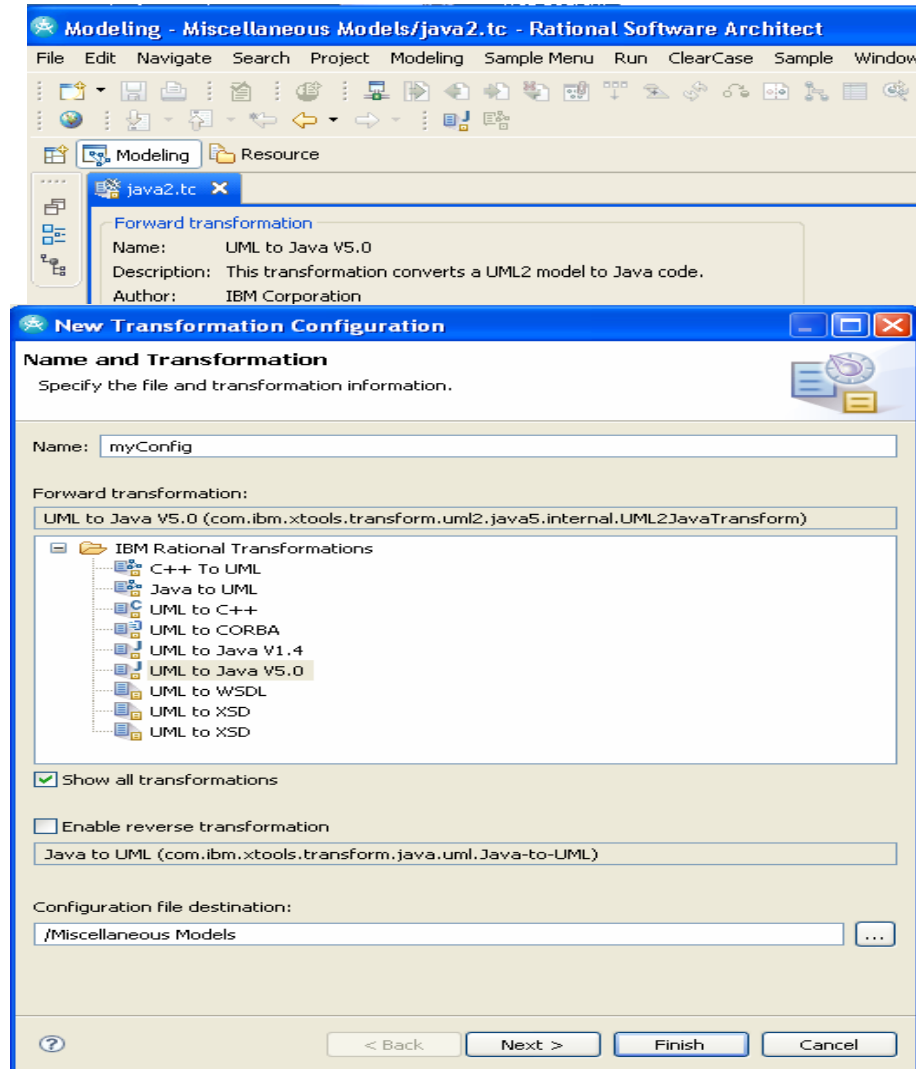
Model To Model Transformations

- **Model to model transformations** (from WIKIPEDIA)
 - ▶ The notion of model transformation is central to Model Driven Engineering. A model transformation takes as input a model conforming to a given metamodel and produces as output another model conforming to a given metamodel. If the source and target metamodels are identical the transformation is called endogeneous. If they are different the transformation is called exogeneous. A model transformation may also have several source models and several target models. One of the characteristics of a model transformation is that a transformation is also a model, i.e. it conforms to a given metamodel. This facilitates the definition of Higher Order Transformations (HOTs), i.e. transformations taking other transformations as input and/or transformations producing other transformations as output.

Rational Transformation Framework

The transformation framework has the following features:

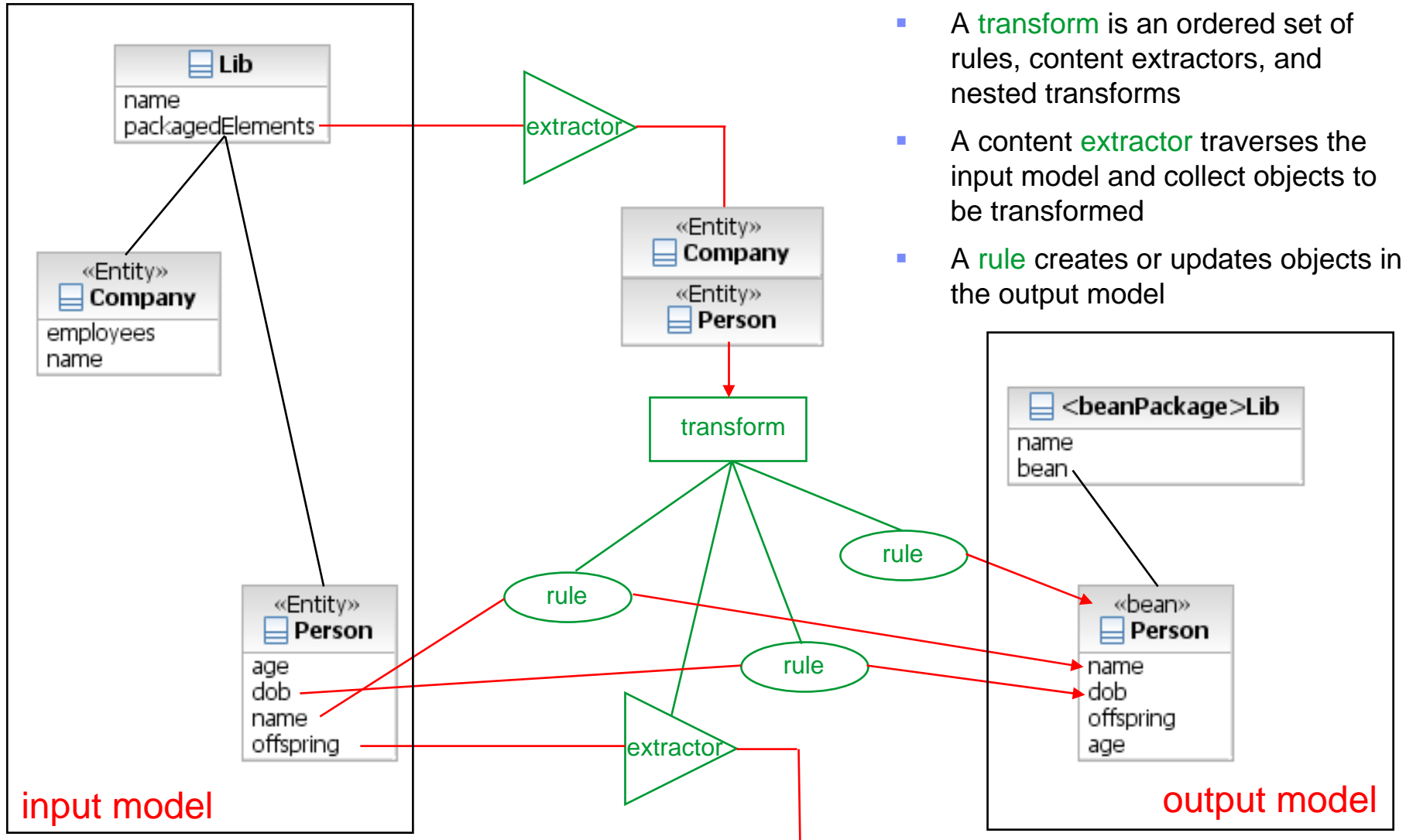
- It is a framework for managing and customizing transformations.
- It provides a default transformation engine.
- It supports a UI and multiple services:
 - ▶ Managing installed transformations
 - ▶ Managing transformation configurations
 - ▶ Managing transformation extensions
 - ▶ Chaining transformations; for example, Model2Model with Model2Text
 - ▶ Supporting headless transformations
 - ▶ Integrating transformations with Eclipse build



Model to Model Transformations

- Transformation frameworks
 - ▶ Rational Transformation Framework (RTF) from IBM Rational
 - Since 6.0
 - Forms the basis for built-in transformations
 - UML-to-Java, UML-to-C++, UML-to-C# ...
 - Allows users to author new transformations
 - ▶ Rational Transformation Mapping Framework (RTMF) from IBM Rational
 - Introduced in 7.0

Model to Model Transformations: RTF



- A **transform** is an ordered set of rules, content extractors, and nested transforms
- A content **extractor** traverses the input model and collect objects to be transformed
- A **rule** creates or updates objects in the output model

Model to Model Transformations: RTMF

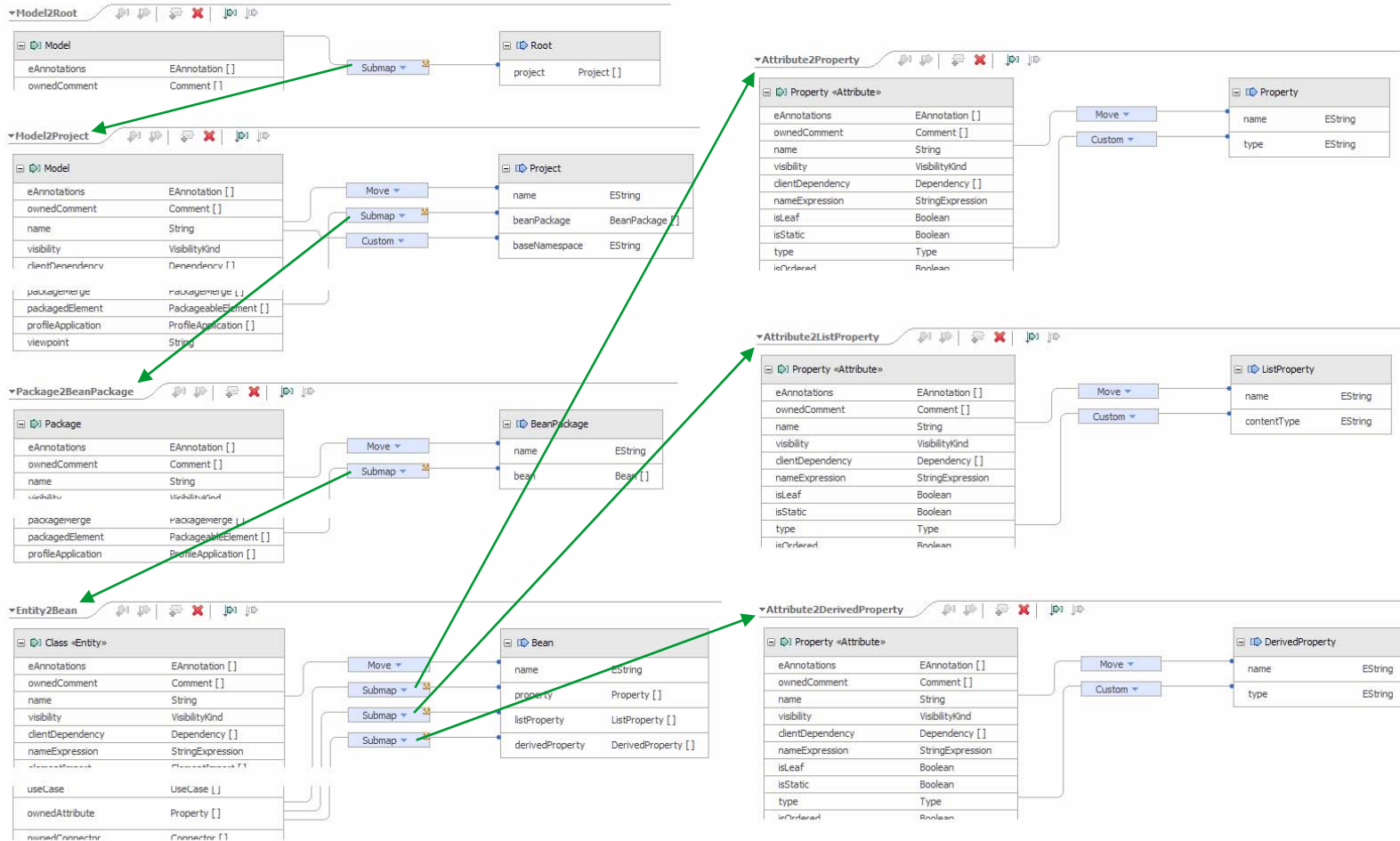
- **Visual construction of RTF transformations**
 - ▶ Design transformations by mapping between input and output metamodel types
 - ▶ Generation of Java source code for RTF transformations
- **Input models**
 - ▶ Ecore (includes UML models, profiles, and libraries) – typically loaded from workspace
- **Generated output models**
 - ▶ In-memory Ecore
- **Post processing options**
 - ▶ Merge model with existing model – the default behavior
 - ▶ Persist model (overwriting existing model if one with same uri exists)
 - ▶ Pass model as input to another transformation (not mutually exclusive with merge/persist)

Model to Model Transformations: Example

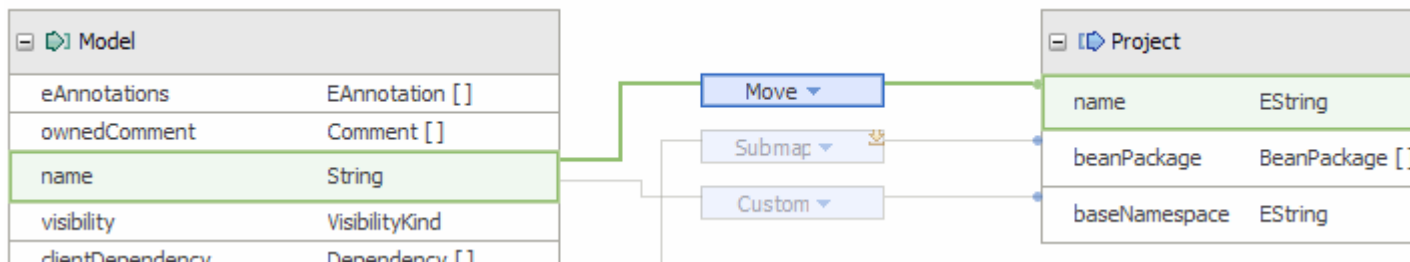


Mappings:

- Mapping Model = set of Mapping Declarations
- Mapping Declaration = set of Mappings
- Mapping = relationships between inputs and outputs
 - ▶ Type (move, submap, custom) conveys how the mapping should be implemented when transformation source code is generated

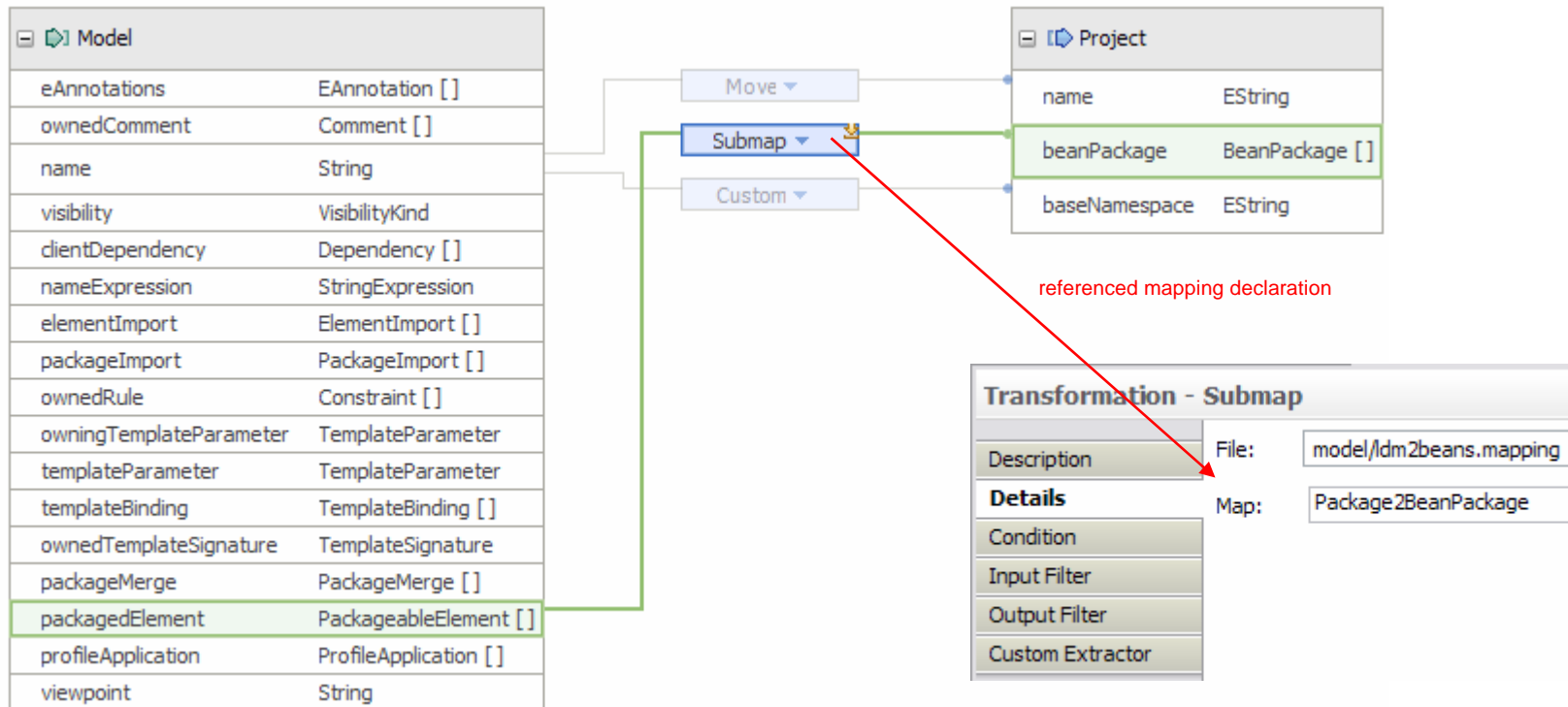


Mappings: Move



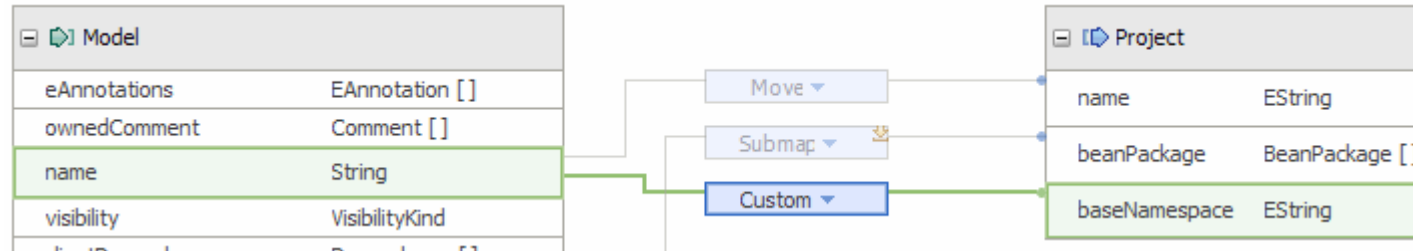
- Create a move mapping when the input and output attributes are compatible primitive types; for example, String, int, boolean
- When transformation source code is generated a rule is defined; this rule copies the contents of the specified input attribute of the current input object to the specified output attribute of the current output object
- A move mapping can have a guard condition associated with it, thus making the rule execution optional

Mappings: Submap



- Create a submap mapping when the input and output attributes are complex types; for example, Model, Package, Class.
- When transformation source code is generated a content extractor is defined; this content extractor passes the contents of the specified input attribute of the current input object to the transform that was generated from the referenced mapping declaration.

Mappings: Custom



- Create a custom mapping when a move or submap mapping is insufficient.
- The transformation author supplies the code that computes the value for the output feature by using the values of input feature(s) of the current input object.
- You can specify Java source code directly in the mapping model or you can provide the name of a Java class that contains the custom code.

Transformation - Custom

Description Code: In-line External

Details

```
protected void executeNameToBaseNamespace_Rule ( Model Model_src, Project Project_tgt ) {
    final String DEFAULT_NAMESPACE = "com.ibm.rsd2008";
    String name = Model_src.getName();
    int lastDelimiter = name.lastIndexOf('.');
    if (lastDelimiter >= 0)
        Project_tgt.setBaseNamespace(name.substring(0, lastDelimiter));
    else
        Project_tgt.setBaseNamespace(DEFAULT_NAMESPACE);
}
```

Mappings: Submap Filters and Customizations

■ Input Filter

- ▶ Subset the collection of input objects that the referenced mapping declaration will process
- ▶ Default: all objects in the input collection

■ Output Filter

- ▶ Decide which generated object will be used to resolve a reference
- ▶ Default: first compatible object generated from input object by specified mapping declaration

■ Custom Extractor

- ▶ Generate the collection of objects that the referenced mapping declaration will use as input
- ▶ Default: all objects in mapped input feature of current input object

■ Custom Feature

- ▶ Specify the feature of the output object that the referenced mapping declaration will modify
- ▶ Default: mapped output feature

■ Custom Output

- ▶ Specify the output object that the referenced mapping declaration will modify
- ▶ Default: current output object

Model to Model Mappings: Example

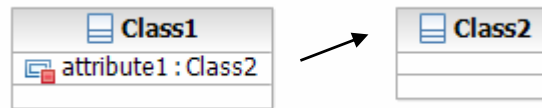


Pattern: Reference Filter

■ Problem

- ▶ Need to create a reference to an output object of a particular type that was generated from a specific input object, but there's multiple output objects of different types to choose from

■ Example



Design

Generated Implementation



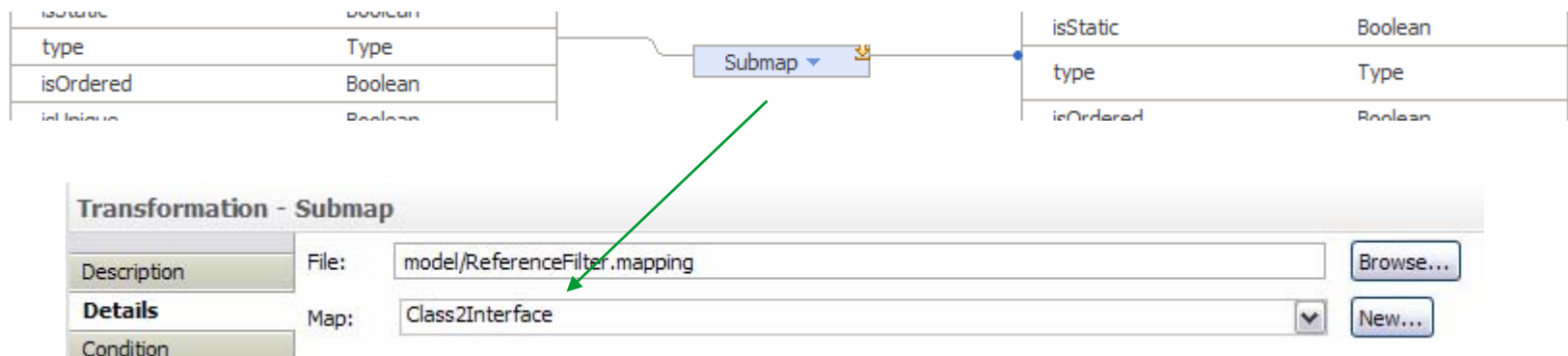
Pattern: Reference Filter

■ Solution

- ▶ Filter types via mapping declarations

■ Strategy

- ▶ Submap mapping specifies mapping declaration that generates a particular type of object



■ Consequences

- ▶ Filtering is symmetric with mapping solution for object creation
- ▶ Second level of filtering might be required if more than one object of same type created

Pattern: Reference Filter Example

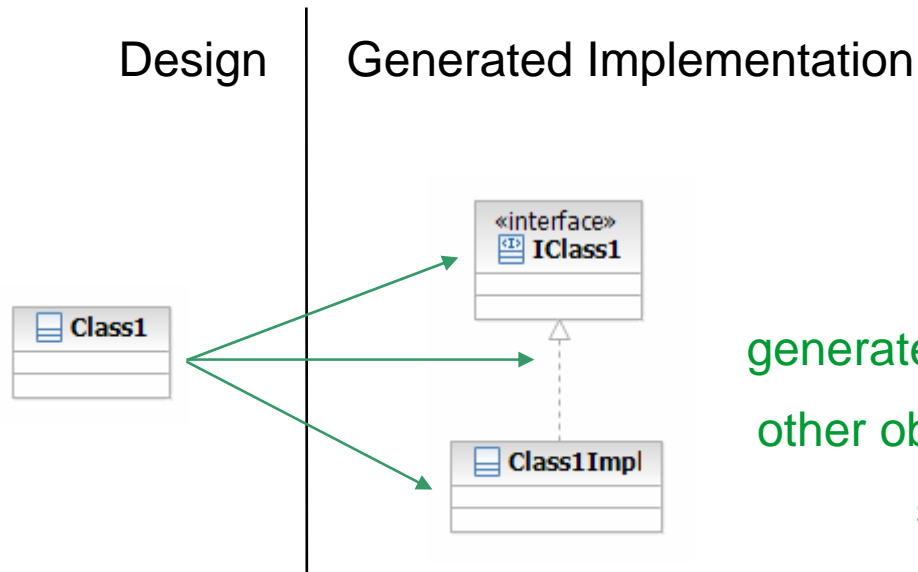


Pattern: One to Many

■ Problem

- ▶ Multiple related output objects must be generated from a single input object

■ Example



generated relationship references
other objects generated from the
same input object

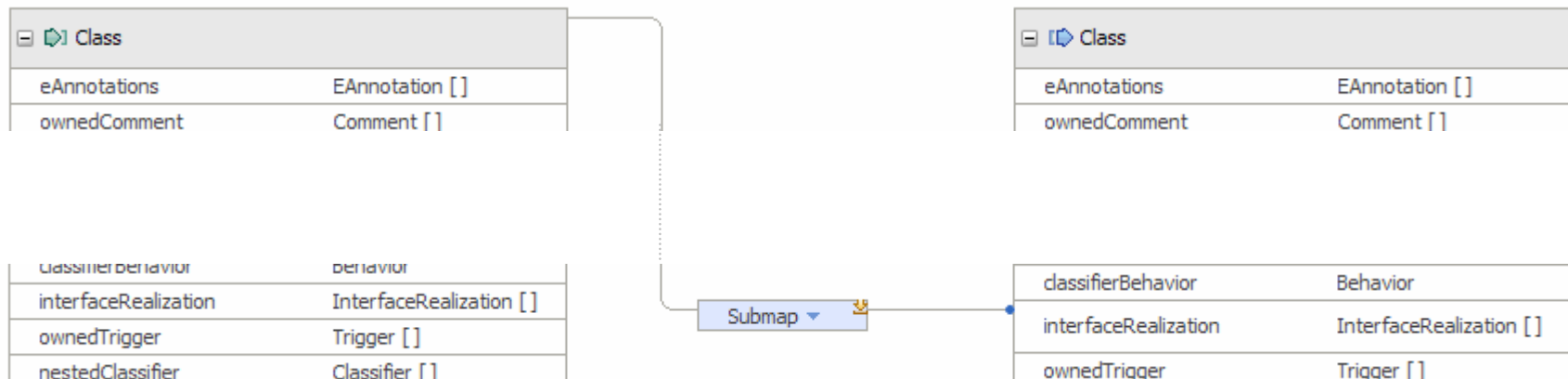
Pattern: One to Many

■ Solution

- ▶ Propagate input object to multiple mapping declarations

■ Strategy

- ▶ Submap mappings specify an element, instead of a feature of element, as their input



■ Consequences

- ▶ Separation of concerns (into multiple mapping declarations) simplifies mapping solution
- ▶ Recursive application of the pattern can make the solution harder to maintain

Pattern: One to Many Example

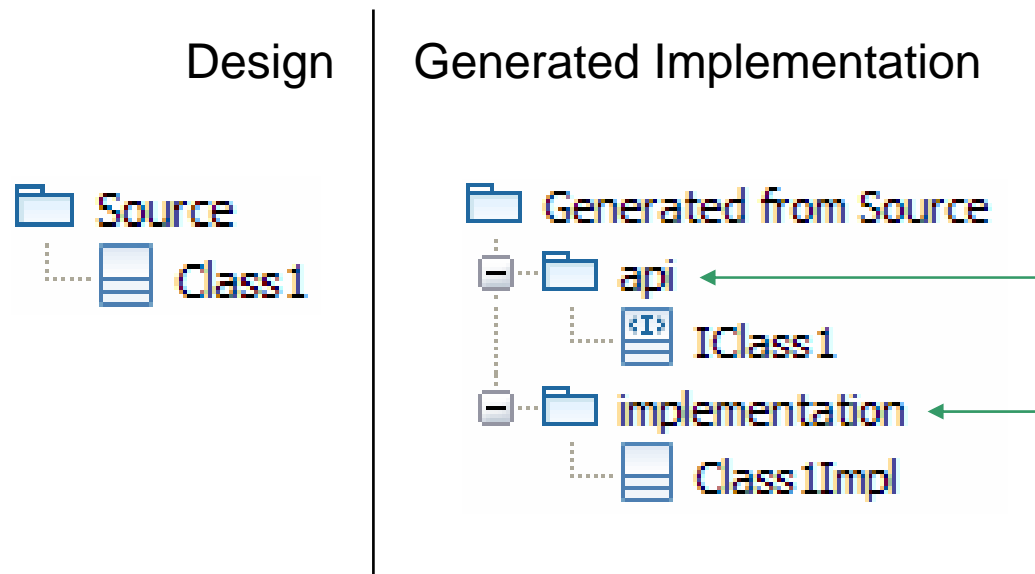


Pattern: Filling the Gaps

■ Problem

- ▶ Output model has structures that don't directly correspond to anything in the input model

■ Example



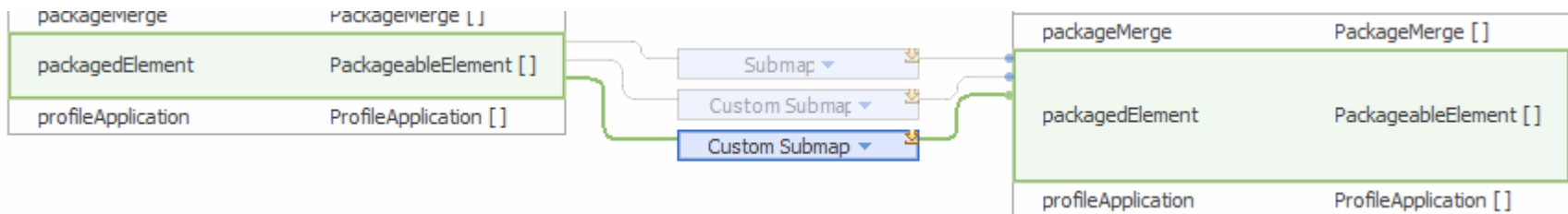
Pattern: Filling the Gaps

■ Solution

- ▶ Redirect output to objects other than the current target container

■ Strategy

- ▶ Custom submap mappings can specify the output object that will be used (by the transform that is generated from the referenced mapping declaration)



Transformation - Custom Submap	
Description	File: <input type="text" value="model/FillingTheGaps.mapping"/> <input type="button" value="Browse..."/>
Details	Map: <input type="text" value="Class2Interface"/> <input type="button" value="New..."/>
Condition	
Input Filter	
Output Filter	
Custom Extractor	
Custom Feature	
Custom Output	

Pattern: Filling the Gaps

Transformation - Custom Submap

Description	<input checked="" type="checkbox"/> Specify a custom output
Details	Code: <input checked="" type="radio"/> Inline <input type="radio"/> External
Condition	<pre>protected EObject extendPackagedElementToPackagedElement_UsingClass2Interface_Output (Package Package_tgt) {</pre>
Input Filter	<pre> final String API_PACKAGE_NAME = "api";</pre>
Output Filter	<pre> Package apiPackage = Package_tgt.getNestedPackage(API_PACKAGE_NAME);</pre>
Custom Extractor	<pre> if (apiPackage == null) {</pre>
Custom Feature	<pre> apiPackage = Package_tgt.createNestedPackage(API_PACKAGE_NAME);</pre>
Custom Output	<pre> } return apiPackage;</pre>

■ Consequences

- ▶ Specification of the target container is isolated from the referenced mapping declaration, and so the population of the target container can still be specified in the normal way with mappings (via the referenced mapping declaration)
- ▶ Custom code must be provided to create instances of some objects

Pattern: Filling the Gaps Example

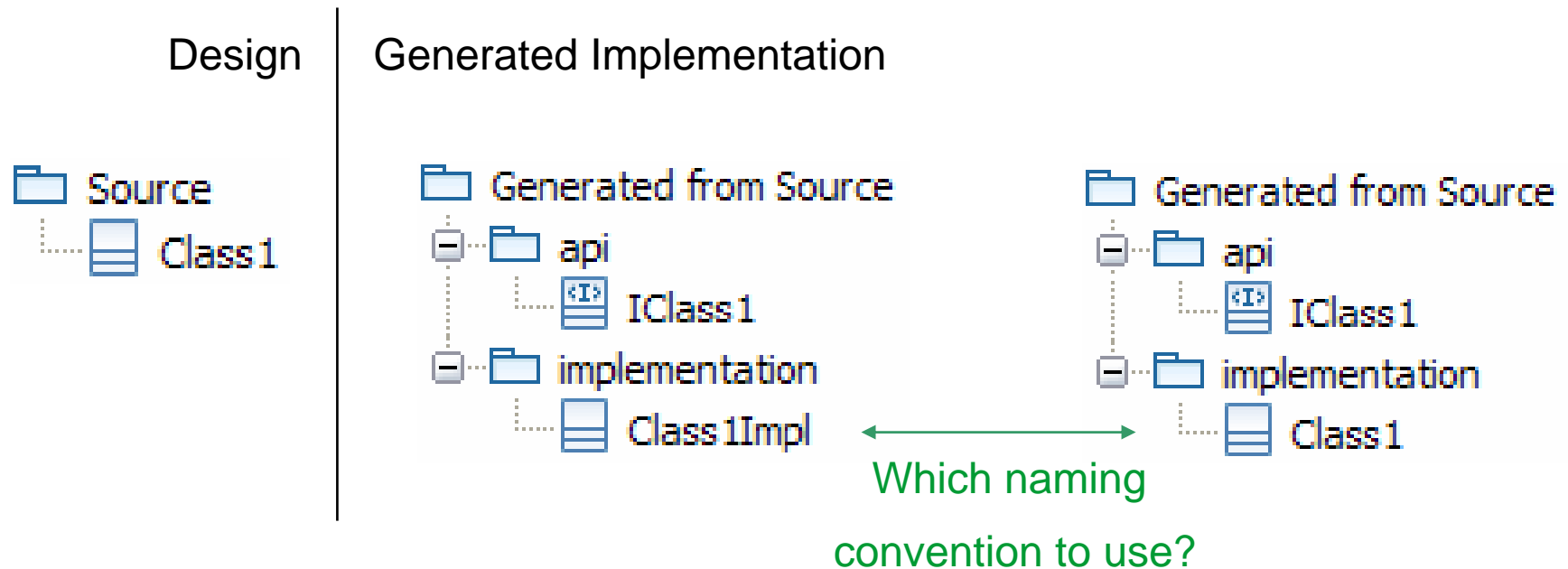


Pattern: Ask the User

■ Problem

- ▶ Output model is dependant on information that's not contained in the input model

■ Example



Pattern: Ask the User

■ Solution

- ▶ Obtain the additional information from the transformation context object

■ Strategy

- ▶ Define transformation specific properties and have the user assign values to those properties programmatically or via the Transformation Configuration Editor

All Extensions

Define extensions for this plug-in in the following section.

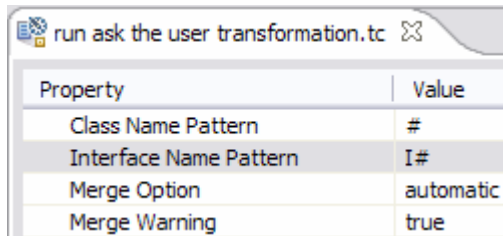
- [-] com.ibm.xtools.transform.core.transformationProviders
 - [X] com.ibm.xtools.transform.authoring.patterns.ask_the_user.A
 - [X] Highest (Priority)
 - [X] Ask the User Transformation (Transformation)
 - [X] Merge Option (Property)
 - [X] Merge Warning (Property)
 - [X] Class Name Pattern (Property)
 - [X] Interface Name Pattern (Property)

Extension Element Details

Set the properties of "Property". Required fields are denoted by "**".

name*:	<input type="text" value="Interface Name Pattern"/>
id*:	<input type="text" value="interface_name_pattern"/>
description:	<input type="text" value="Pattern for forming interface names ('#' is source class name)"/>
value:	<input type="text" value="I#"/>
metatype:	<input type="text" value="String"/>
metatypeData:	<input type="text"/>
maxValues:	<input type="text"/>
delimiters:	<input type="text"/>
readonly:	<input type="checkbox" value="false"/>

Pattern: Ask the User



Property	Value
Class Name Pattern	#
Interface Name Pattern	I#
Merge Option	automatic
Merge Warning	true

Transformation Configuration Editor

← Properties tab

Retrieve property

value from

transformation context

```

public void execute(EObject arg0, EObject arg1) {
    if (arg0 instanceof NamedElement && arg1 instanceof NamedElement) {
        String name = ((NamedElement)arg0).getName();
        String pattern = null;
        if (arg1 instanceof Class) {
            pattern = (String)context.getPropertyValue(CLASS_PATTERN_PROPERTY_ID);
        } else if (arg1 instanceof Interface) {
            pattern = (String)context.getPropertyValue(INTERFACE_PATTERN_PROPERTY_ID);
        }
        if (pattern != null && pattern.contains(NAME_PLACE HOLDER_TOKEN)) {
            name = pattern.replaceAll(NAME_PLACE HOLDER_TOKEN, name);
        }
        ((NamedElement)arg1).setName(name);
    }
}

```

■ Consequences

- ▶ Supplying information via properties is simpler than supplying via auxiliary models
- ▶ Complex properties can require custom GUI for specifying values

Pattern: Ask the User Example

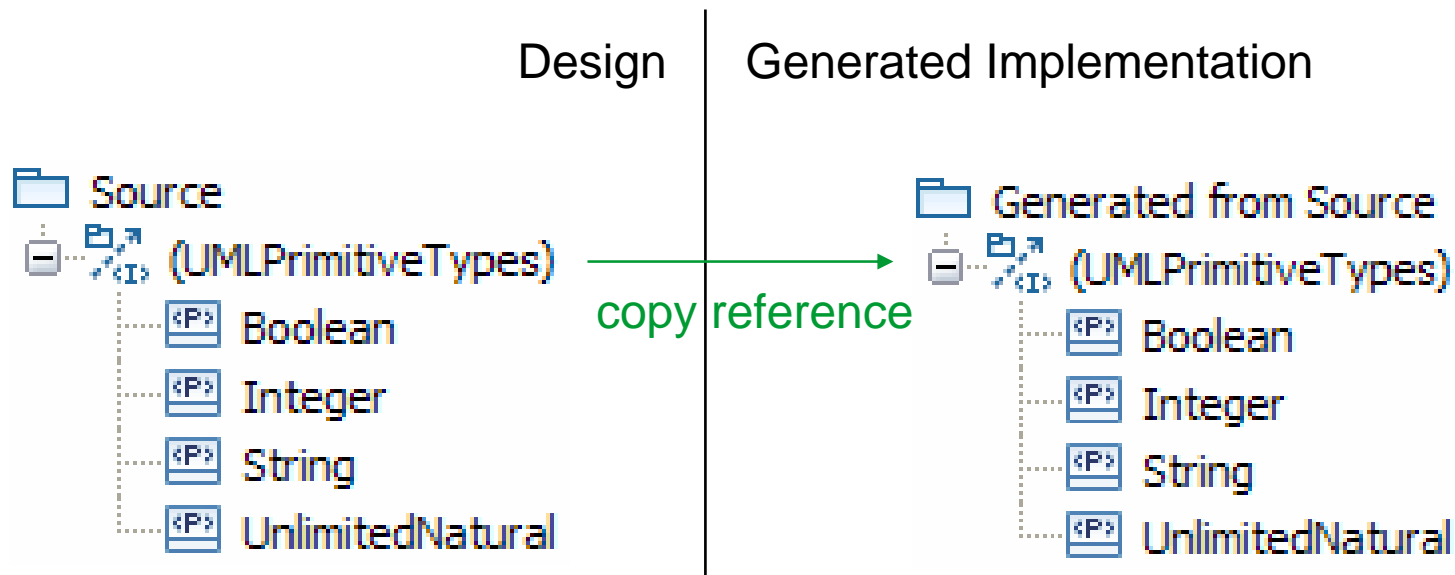


Pattern: Copy a Reference

■ Problem

- ▶ A reference to an external object (an object that is contained in a model that is not being transformed) needs to be propagated from the input model to the output model

■ Example



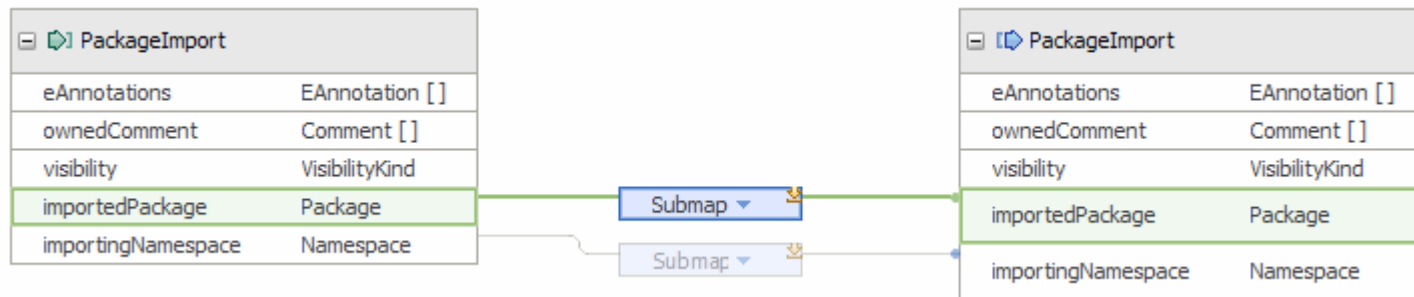
Pattern: Copy a Reference

■ Solution

- ▶ Use a submap mapping that specifies an empty (or ignorable) mapping declaration

■ Strategy

- ▶ When the target of a submap mapping is a reference, the specified mapping declaration is only used (at transformation development time and transformation execution time) to ensure type safety and assist with reference resolutions; consequently, any mappings in the specified mapping declaration will be ignored. Intra-model references are copied if no corresponding object was generated.



■ Consequences

- ▶ No need to define mappings for types that will be referenced, but not contained
- ▶ Transformation will create empty objects for instances of types with empty mapping declarations when the occurrence is by containment rather than reference

Pattern: Copy a Reference

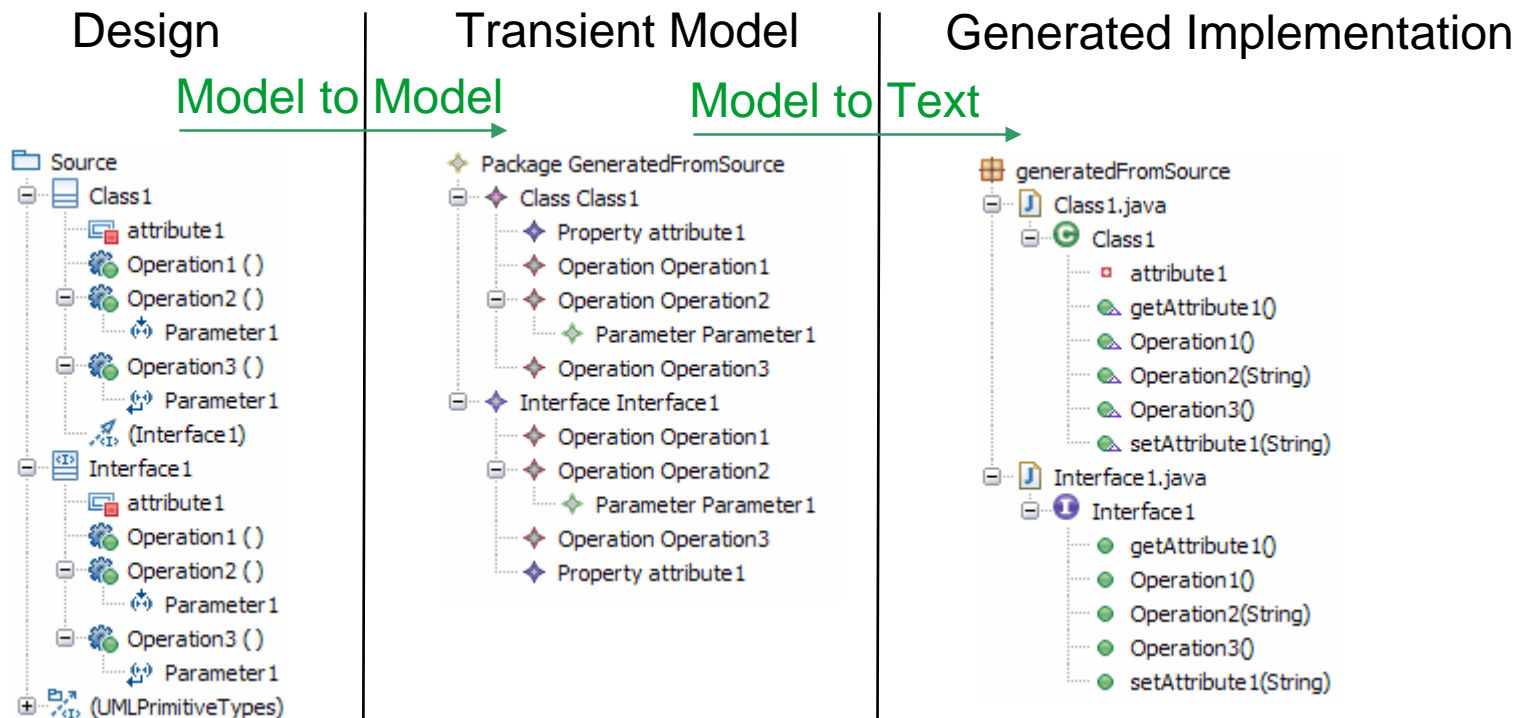


Pattern: Chain

■ Problem

- ▶ Output model is input for other transformations

■ Example



Pattern: Chain

■ Solution

- ▶ Add rules that chain to other transformations

■ Strategy

- ▶ Add instances of ChainRule or JETRule to PostProcessing rules of the RootTransformation - a class which instantiates the top level (Main) transform of the generated transformation

```
protected RootTransformation createRootTransformation(ITransformationDescriptor descriptor) {  
    return new RootTransformation(descriptor, new MainTransform()) {  
        @Override  
        protected void addPostProcessingRules() {  
            add(new JETRule("com.ibm.xtools.transform.authoring.patterns.chain.jet"));  
        }  
    };  
}
```

■ Consequences

- ▶ Generated in-memory (optionally persisted) model can easily be passed to other transformations, usually with no special adaptation for those transformations required
- ▶ Custom, transformation specific, properties can require extensions to standard chain rules

Questions

Thank You

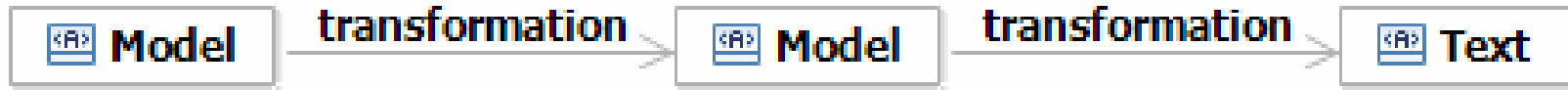
© Copyright IBM Corporation 2009. All rights reserved. The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. Product release dates and/or capabilities referenced in these materials may change at any time at IBM's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way. IBM, the IBM logo, Rational, the Rational logo, Telelogic, the Telelogic logo, and other IBM products and services are trademarks of the International Business Machines Corporation, in the United States, other countries or both. Other company, product, or service names may be trademarks or service marks of others.





Backup

Example solution: LogicalDataModel to Java beans



- Create two transformations, and transform in two steps:
 - ▶ From a UML model that has the LogicalDataModel profile applied to it, create an instance of an Ecore model for Java beans.
 - ▶ From an Ecore model for Java beans, create a Java project and populate it with Java bean source code.
- Why do this in two steps?
 - ▶ The best practices for authoring model-to-text transformations suggest the following steps:
 - Harvest the templates from an exemplar project. For example, harvest the templates from a Java bean reference project.
 - Construct a model that contains the information that is required to instantiate the templates. For example, construct an Ecore model for Java beans.
 - ▶ Different requirements were used to design the LogicalDataModel profile and the Ecore Java bean model.
 - ▶ These different requirements often lead to disparities between the models, which are typically resolved by inserting an additional transformation step.
 - ▶ You can reuse the transformations independently. For example, you can reuse the Ecore-model-to-Java-beans transformation in many other contexts.

Designing Model to Model Transformations: Mappings

- The following factors contribute to the complexity of transformation authoring:
 - ▶ The inherit complexity in the input and output metamodels and their APIs
 - ▶ The relationships of interest between the input and output metamodels
 - ▶ The transformation framework elements, including the engine, languages, supporting tools, and APIs
- Learning the transformation framework is not the goal of the transformation author, but it often becomes the main task
 - ▶ The framework complexity and abstraction level impacts the productivity of experts and novices
 - ▶ Complexity is a common problem for transformation frameworks:
 - XSLT with XPath
 - Query/View/Transformation (QVT) operational or relational transformation language with Object Constraint Language (OCL)
- Mappings simplify transformation authoring so that the author can focus on the **problem** and not the tooling required to implement the **solution**



Designing Model to Model Transformations: Example

- Sketch of a design: mapping LogicalDataModel things to Java bean model things

