Software Architecture & Design

6CCS3SAD / 7CCSMDAS

Dr Kevin Lano

Session 10:
Model-Driven Engineering 2

March 11th-18th, 2019

Learning Outcomes

At the end of this session you should be able to:

- Write simple model-to-text transformations in the Epsilon Generation Language (EGL)
- Design simple model-to-model transformations and express them in OCL.
- Understand the application of Model Driven Engineering to software architecture.

Reminder: What we wanted to do

- Automate implementations of architecture
  - By generating source code from architecture models
  - Using model-to-text transformations
    - Requires formal representation of models
    - Requires language for expressing transformations
    - Requires engine for executing transformations

Reminder: What we wanted to do

- Automate implementations of architecture
  - By generating source code from architecture models
  - Using model-to-text transformations
    - Requires formal representation of models
    - ✓ Requires language for expressing transformations
    - Requires engine for executing transformations

Model-to-Text Transformation Engines

- Quite a few exist
  - Java Emitter Templates
  - openArchitectureWare XPand
  - MOF2Script
  - Epsilon Generation Language
  - …

- All based on the same concept: Templates

Model-to-Text Transformation Engines

- Quite a few exist
  - Eclipse Modeling Framework
  - openArchitectureWare XPand
  - MOF2Script
  - Epsilon Generation Language
  - …

- All based on the same concept: Templates

Epsilon Generation Language (EGL)

- Part of Epsilon
  - "Model-Management Framework"
    - Set of languages and engines for manipulating models in various ways
  - Developed at University of York
  - Integrated into Eclipse IDE
    - Available from http://www.eclipse.org/epsilon/
- EGL is focused on Model-to-Text Transformations
  - Also called "code generation" – hence the name of EGL

Code Generation with EGL

- Code generation is specified using templates
  - Mix of target text and scripting code to evaluate model contents
  - Looks a bit like an ASP/JSP page
    - But scripting is specific to model evaluation
  - Scripting language:
    - OCL to navigate models
    - Additional control structure (if, foreach, ...)
- Templates are then evaluated for specific models or model elements

An EGL Example

Template

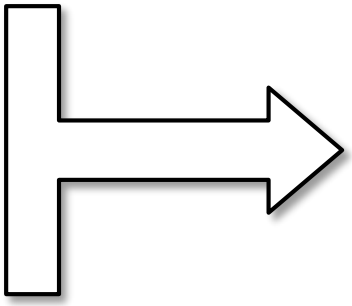
```
public class [%=c.name%]
    [%if (c.isCOMPlus()) {%] : ServicedComponent [%}%] {

[%if (c.isCOMPlus()) {%]
  public [%=c.name%]() {}
[%}%]

  // methods go here
}
```

- The [% %] delimit the script code, executable EGL
- Outside the [% %] is fixed text of the target language (here, C#)

An EGL Example

Template

```
public class [%=c.name%]
    [%if (c.isCOMPlus()) {%] : ServicedComponent [%}%] {

[%if (c.isCOMPlus()) {%]
  public [%=c.name%]() {}
[%}%]

  // methods go here
}
```



```
public class HolidayRes : ServicedComponent {

  public HolidayRes() {}

  // methods go here
}
```

Resulting Code

<<COM+>>
HolidayRes

A Closer Look at the Template

```
public class [%=c.name%]
    [%if (c.isCOMPlus()) {%] : ServicedComponent [%}%] {

[%if (c.isCOMPlus()) {%]
  public [%=c.name%]() {}
[%}%]

  // methods go here
}
```

A Closer Look at the Template

```
public class [%=c.name%]
    [%if (c.isCOMPlus()) {%] : ServicedComponent [%}%] {

[%if (c.isCOMPlus()) {%]
  public [%=c.name%]() {}
[%}%]

  // methods go here
}
```

Target text (code)

A Closer Look at the Template

```
public class [%=c.name%]
    [%if (c.isCOMPlus()) {%] : ServicedComponent [%}%] {

[%if (c.isCOMPlus()) {%]
  public [%=c.name%]() {}
[%}%]

  // methods go here
}
```

Scripting code with model evaluation

A Closer Look at the Template

c passed in as parameter, assumed to be a Component

```
public class [%=c.name%]
    [%if (c.isCOMPlus()) {%] : ServicedComponent [%}%] {

[%if (c.isCOMPlus()) {%]
  public [%=c.name%]() {}
[%}%]

  // methods go here
}
```

Scripting Code Blocks

- Two types
  - [%= <exp> %]
    - Evaluate **`<exp>`** and replace the scripting block with the result
  - [% <code-fragment> %]
    - Evaluate **`<code-fragment>`**
      - Can be partial code → all statements together must have appropriate block structure
    - Useful for wrapping control structures around parts of the generation (e.g., conditional generation)
    - **`[% if (c.isCOMPlus()) {%] : ServicedComponent [%}%]`**
      - Prints **`:ServicedComponent`** only for COM+ components

Control Structures Available

- Pretty much all that exist for programming languages
  - If-then-else

    ```
    if (<exp>) {<code>} else {<code>}
    ```

  - Foreach-loops

    ```
    for (<var> in <exp>) {<code>}
    ```

    - Evaluate **`<code>`** for every element in **`<exp>`** (accessible through **`<var>`**)
    - **`<exp>`** must evaluate to an OCL collection
  - ...

Expressions in EGL

- Based on OCL
  - All OCL constructs, especially model navigation
  - May directly refer to meta-classes by their name
    - **`Package.all`** returns all packages in the source model (i.e., all instances of meta-class **`Package`**)

A More Complex Template

```
public class [%=c.name%]
    [%if (c.isCOMPlus()) {%] : ServicedComponent [%}%] {

[%if (c.isCOMPlus()) {%]
  public [%=c.name%]() {}
[%}%]

  [%for (op in c.provided.operation) {%]
  public [%=op.returnType%] [%=op.name%](
      [%
      var first = true;
      for (p in op.params) {
        if (not first) {%], [%}
        first = false; %]
        [%=p.type%] [%=p.name%]
      [%}%]
      ) {}
  [%}%]
}
```

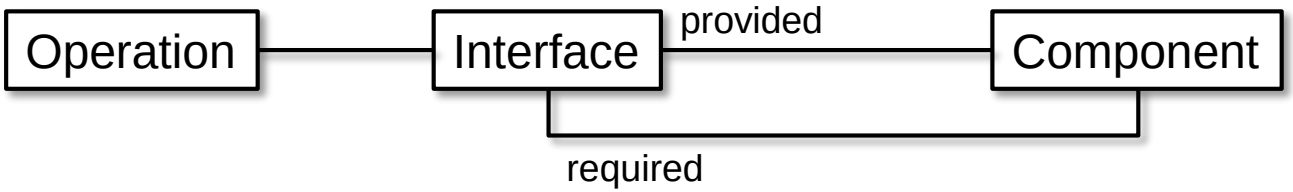
A More Complex Template

```
public class [%=c.name%]
    [%if (c.isCOMPlus()) {%] : ServicedComponent [%}%] {

[%if (c.isCOMPlus()) {%]
  public [%=c.name%]() {}
[%}%]

  [%for (op in c.provided.operation) {%]
  public [%=op.returnType%] [%=op.name%](
      [%
      var first = true;
      for (p in op.params) {
        if (not first) {%], [%}
        first = false; %]
        [%=p.type%] [%=p.name%]
      [%}%]
      ) {}
  [%}%]
}
```

A More Complex Template

```
public class [%=c.name%]
    [%if (c.isCOMPlus()) {%] : ServicedComponent [%}%] {

[%if (c.isCOMPlus()) {%]
  public [%=c.name%]() {}
[%}%]

  [%for (op in c.provided.operation) {%]
  public [%=op.returnType%] [%=op.name%](
      [%
      var first = true;
      for (p in op.params) {
        if (not first) {%], [%}
        first = false; %]
        [%=p.type%] [%=p.name%]
      [%}%]
      ) {}
  [%}%]
}
```

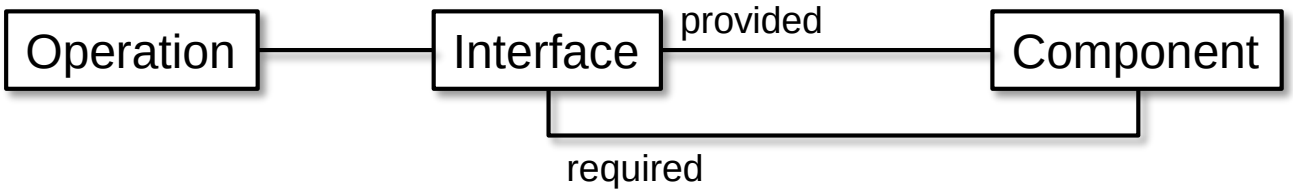
For every operation *op* in a provided interface of *c* generate...



A More Complex Template

```
public class [%=c.name%]
    [%if (c.isCOMPlus()) {%] : ServicedComponent [%}%] {

[%if (c.isCOMPlus()) {%]
  public [%=c.name%]() {}
[%}%]

  [%for (op in c.provided.operation) {%]
  public [%=op.returnType%] [%=op.name%](
      [%
      var first = true;
      for (p in op.params) {
        if (not first) {%], [%}
        first = false; %]
        [%=p.type%] [%=p.name%]
      [%}%]
      ) {}
  [%}%]
}
```

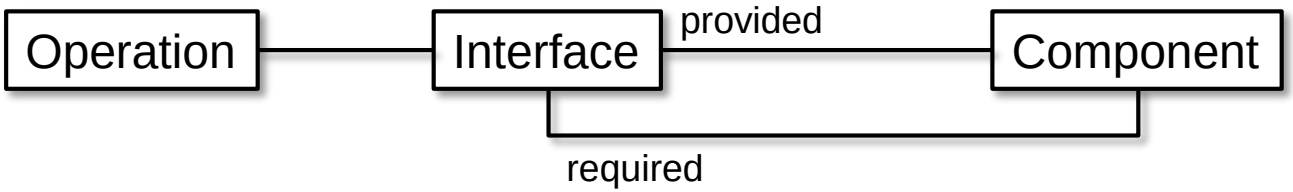
...a method header...



A More Complex Template

```
public class [%=c.name%]
    [%if (c.isCOMPlus()) {%] : ServicedComponent [%}%] {

[%if (c.isCOMPlus()) {%]
  public [%=c.name%]() {}
[%}%]

  [%for (op in c.provided.operation) {%]
  public [%=op.returnType%] [%=op.name%](
      [%
      var first = true;
      for (p in op.params) {
        if (not first) {%], [%}
        first = false; %]
        [%=p.type%] [%=p.name%]
      [%}%]
      ) {}
  [%}%]
}
```

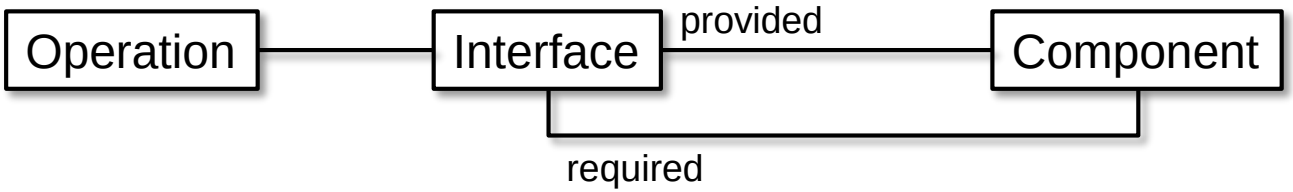
...with an empty body.



A More Complex Template

```
public class [%=c.name%]
    [%if (c.isCOMPlus()) {%] : ServicedComponent [%}%] {

[%if (c.isCOMPlus()) {%]
  public [%=c.name%]() {}
[%}%]

  [%for (op in c.provided.operation) {%]
  public [%=op.returnType%] [%=op.name%](
      [%
      var first = true;
      for (p in op.params) {
        if (not first) {%], [%}
        first = false; %]
        [%=p.type%] [%=p.name%]
      [%}%]
      ) {}
  [%}%]
}
```

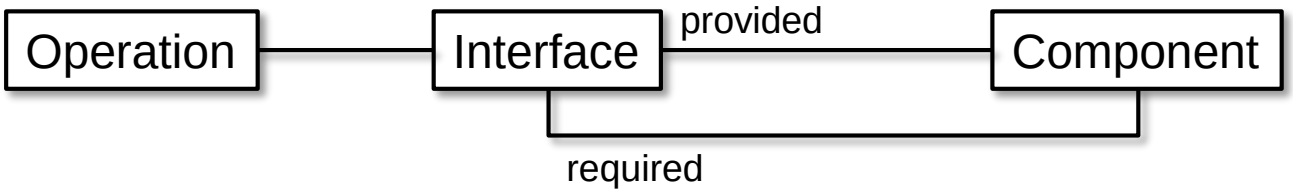
This loop generates the parameter list. Note the use of variable first to determine when to generate a comma.



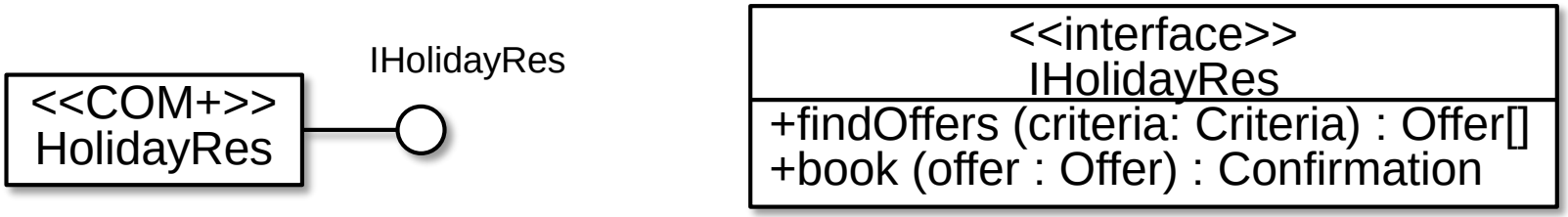
A More Complex Template

```
public class [%=c.name%]
    [%if (c.isCOMPlus()) {%] : ServicedComponent [%}%] {

[%if (c.isCOMPlus()) {%]
  public [%=c.name%]() {}
[%}%]

  [%for (op in c.provided.operation) {%]
  public [%=op.returnType%] [%=op.name%](
      [%
      var first = true;
      for (p in op.params) {
        if (not first) {%], [%}
        first = false; %]
        [%=p.type%] [%=p.name%]
      [%}%]
      ) {}
  [%}%]
}
```

Generating one parameter.



Generation Result




```
public class HolidayRes : ServicedComponent {

  public HolidayRes() {}

  public Offer[] findOffers(Criteria criteria) {}

  public Confirmation book(Offer offer) {}
}
```

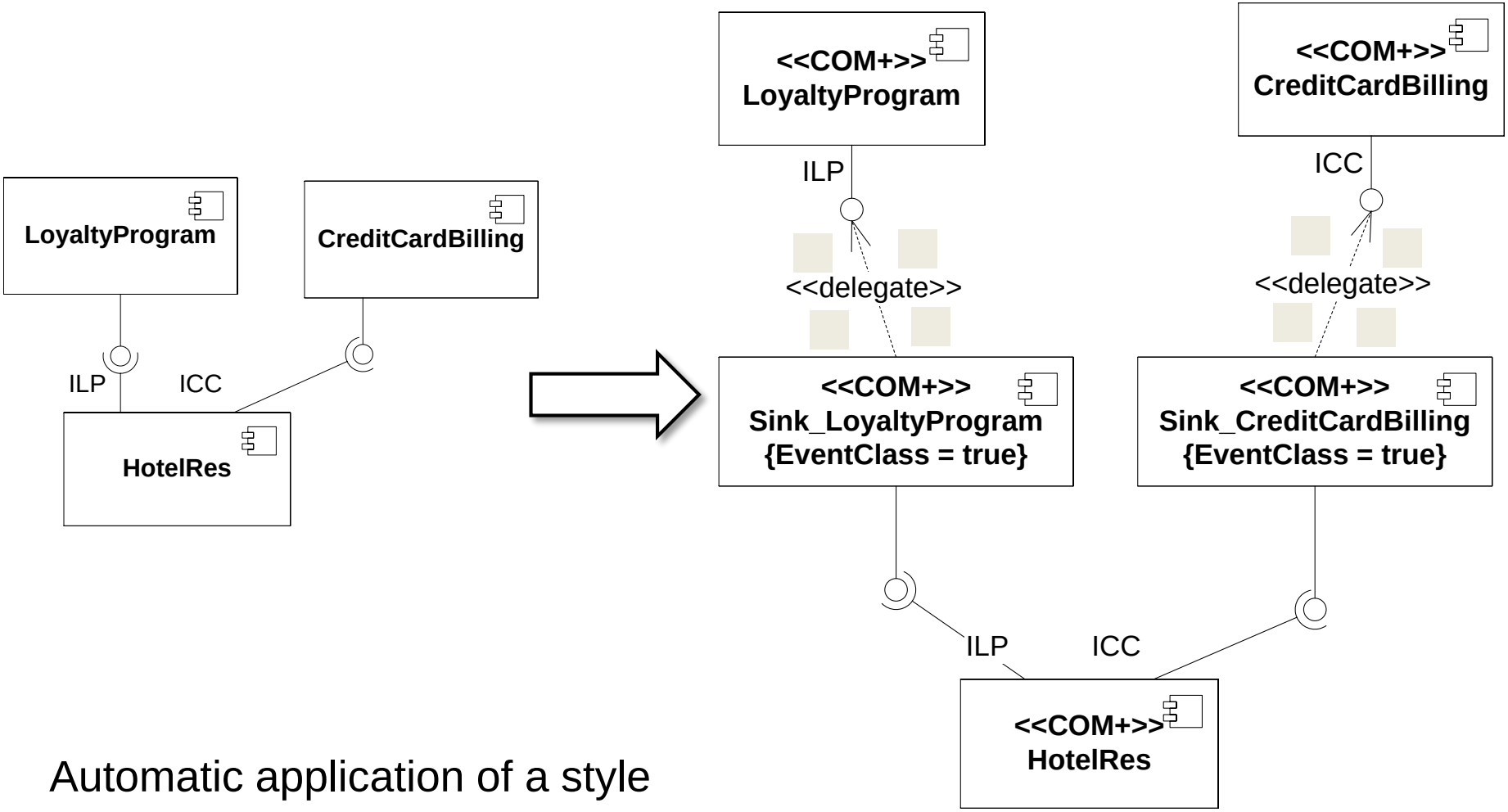
Model-Driven Engineering (MDE)

- Overview
  - Models as central artefacts of software development
    - Architecture, design, ...
  - Model transformations to encode and automate processing of models
    - Model-to-model: Transform models into other models
    - Model-to-text: Transform models to source code

Model-Driven Engineering (MDE)

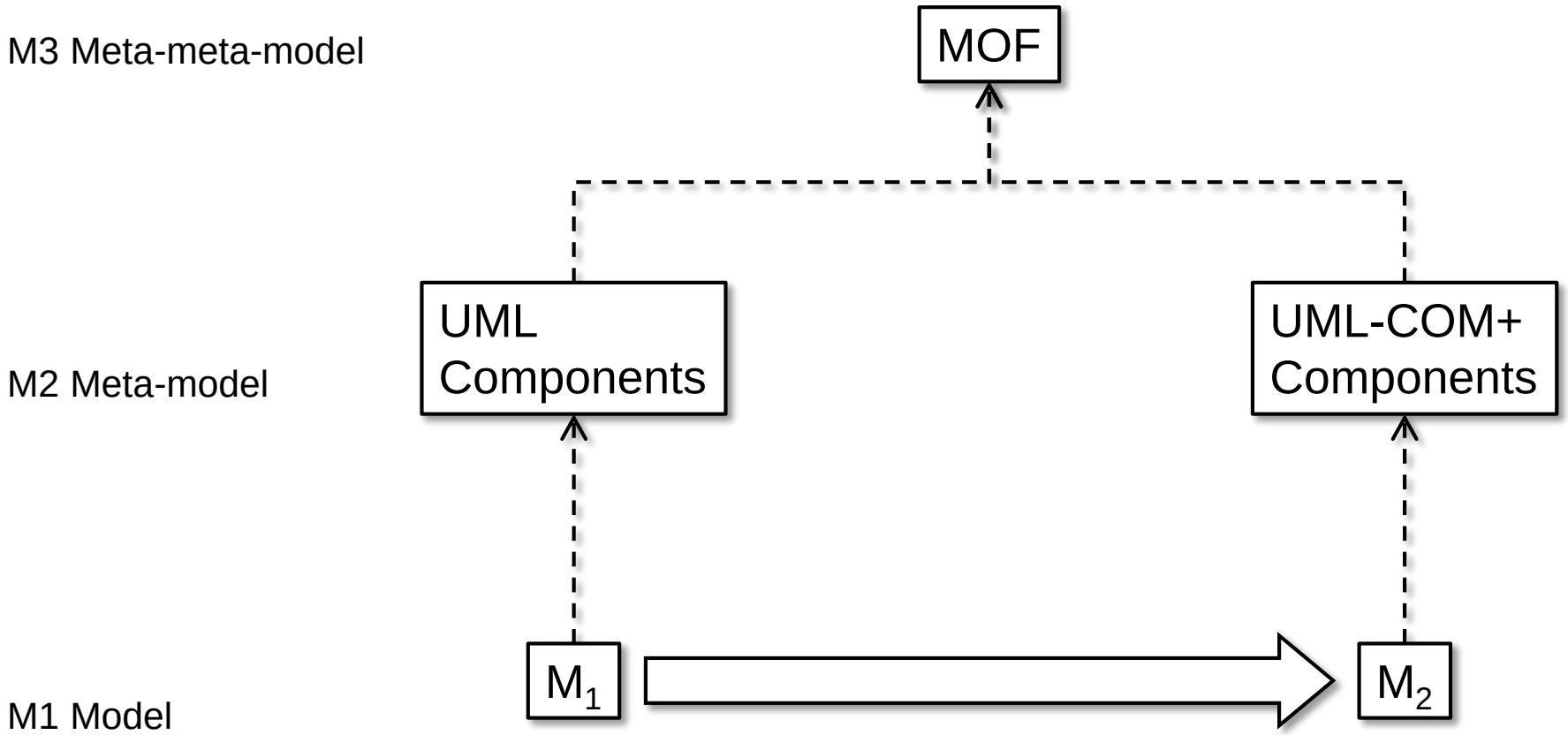
- Overview
  - Models as central artefacts of software development
    - Architecture, design, ...
  - Model transformations to encode and automate processing of models
    - Model-to-model: Transform models into other models
    - Model-to-text: Transform models to source code

Example M2M Transformation

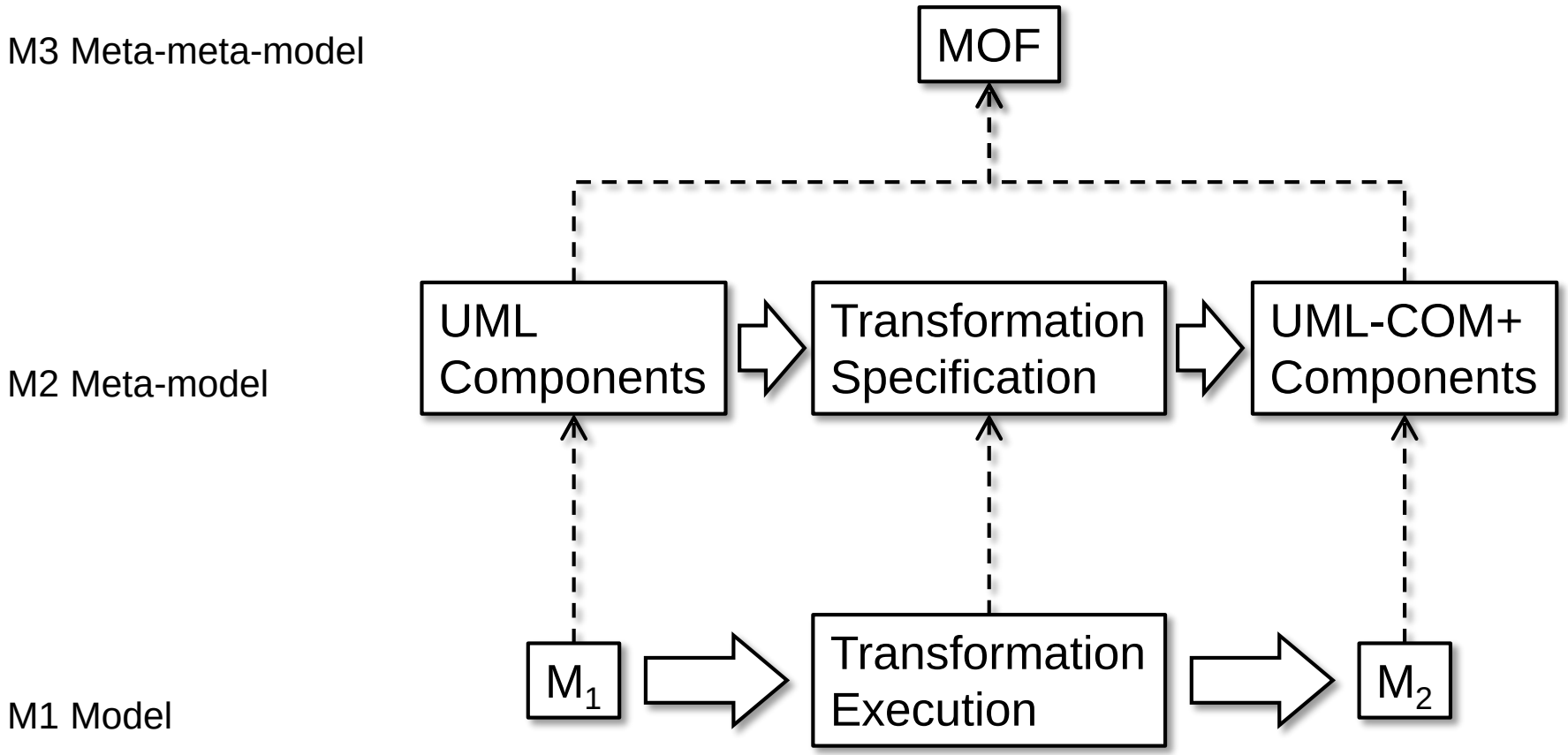


Automatic application of a style

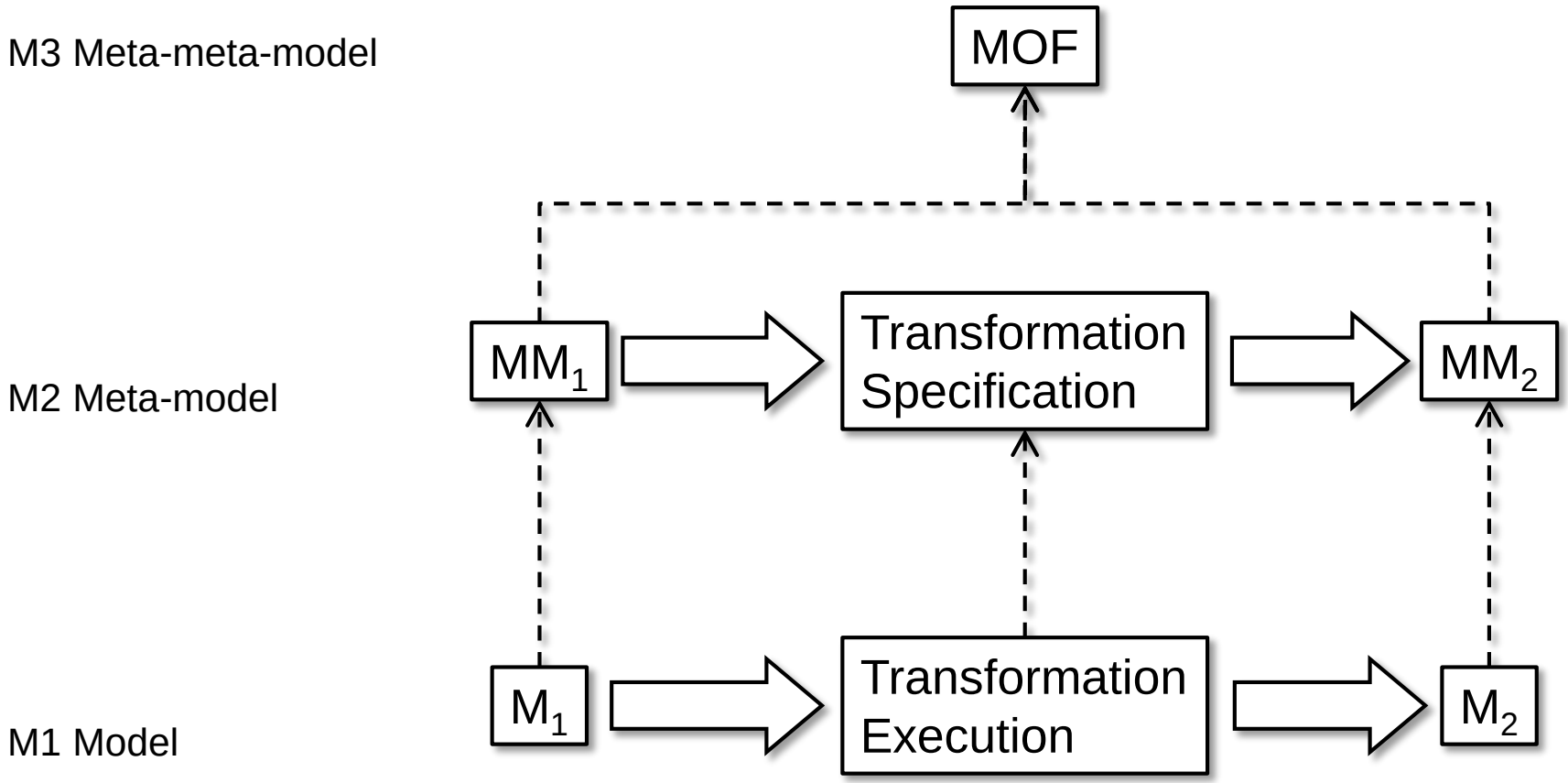
Model-to-Model Transformations



Model-to-Model Transformations



Model-to-Model Transformations

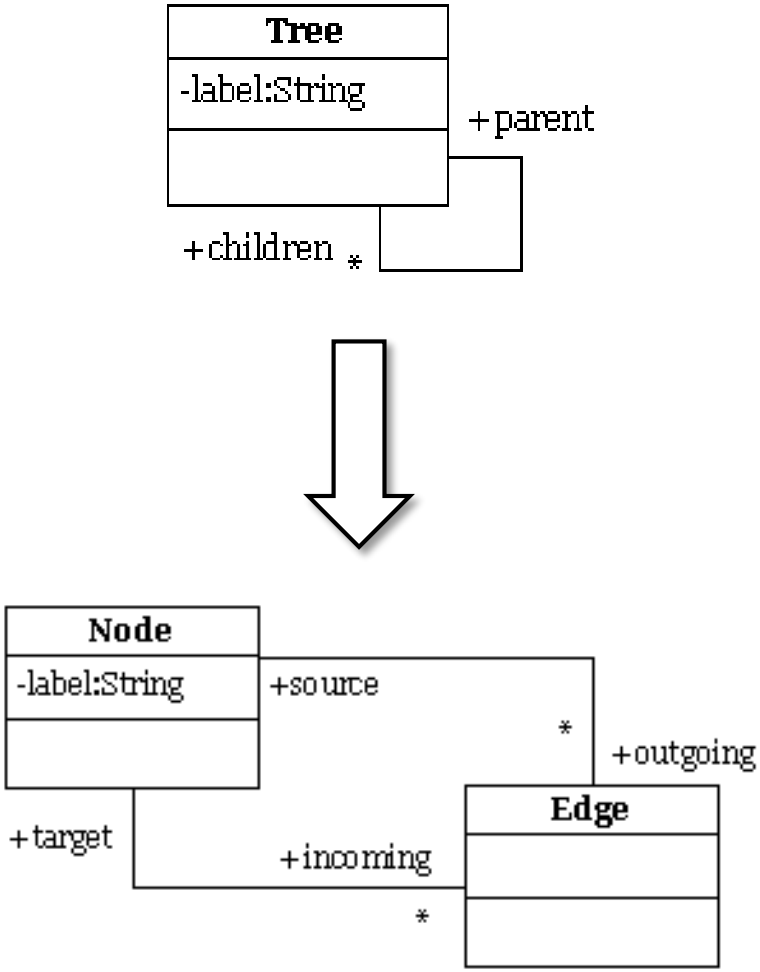


M2M Transformation Languages

- A number of different languages (and engines) exist
  - ATL
    - Widely used academic language and engine
  - **Epsilon Transformation Language (ETL)**
    - Similar to ATL, more program-like style
  - QVT-R, **UML-RSDS**
    - Declarative transformation languages
  - openArchitectureWare Xtend, KerMeta, QVT-O
    - Imperative transformation languages

Example ETL Transformation

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```



Example ETL Transformation

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

Example ETL Transformation

A rule named Tree2Node...

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

Example ETL Transformation

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

...taking an instance of meta-class Tree in model Tree...

Example ETL Transformation

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

...and producing an instance of meta-class Node in model Graph.

Example ETL Transformation

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

Imperative rule body.

Example ETL Transformation

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

Model navigation using OCL

Example ETL Transformation

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.paren
  }
}
```

Explicit creation of an instance of Edge
Instances of Node are implicitly created when the rule is executed.

Example ETL Transformation

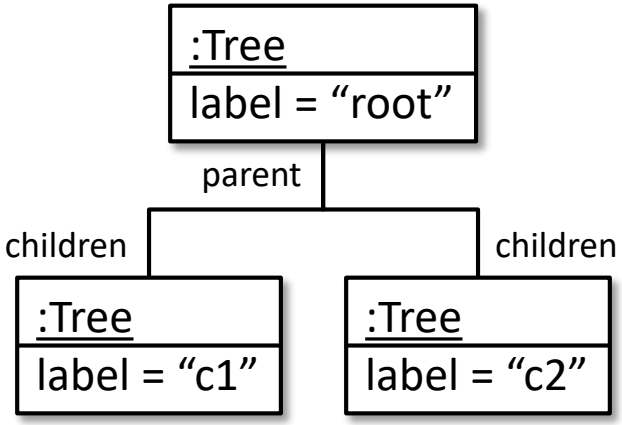
```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

Implicit rule invocation
Assign edge.target to reference whatever t.parent is translated to.

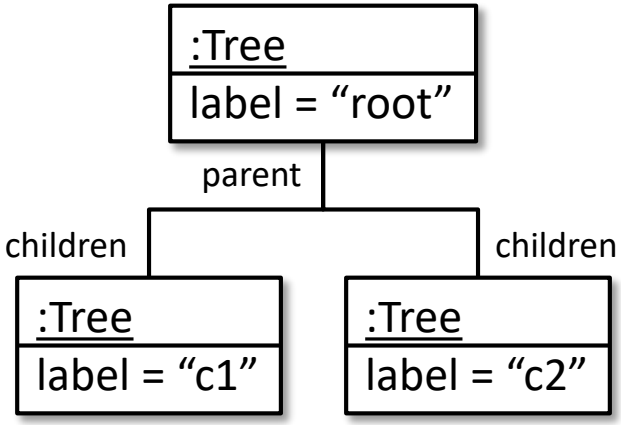
Key Concepts of ETL Model Transformations

- Rules
  - Are implicitly invoked as needed to ensure complete transformation
    - All model elements whose meta-class is the source of a rule are transformed by that rule
    - Rules may be implicitly invoked using **`::=`**
- OCL navigation
  - OCL applied to objects that are instances of the meta-model
- Implicit creation of target model elements
  - When appropriate rules are invoked
  - Explicit creation possible if needed (using **`new`** operator)

Running the transformation

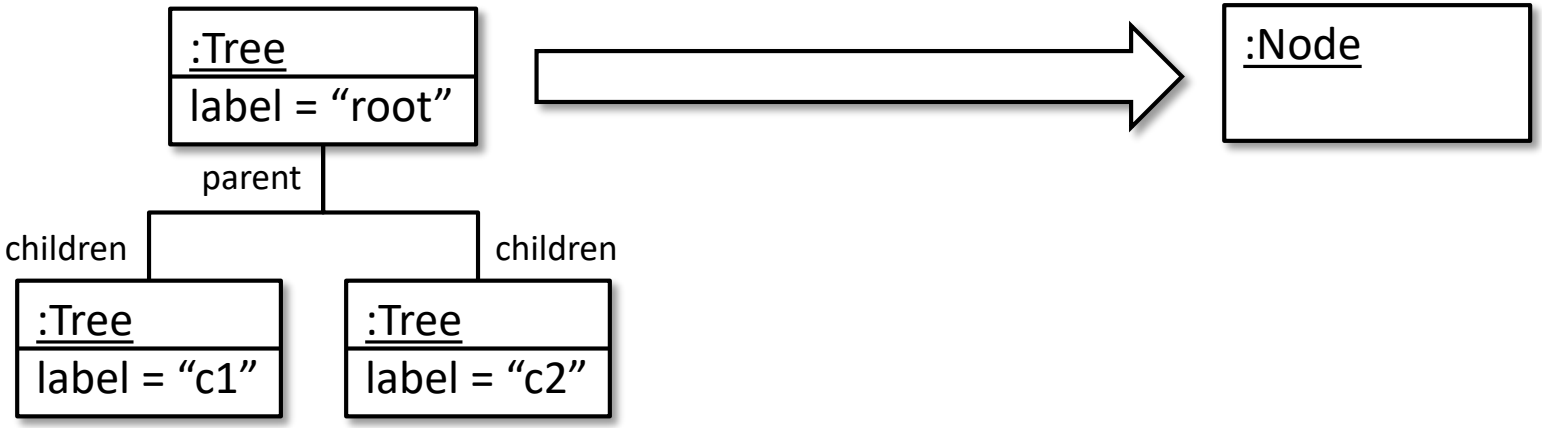


Running the transformation



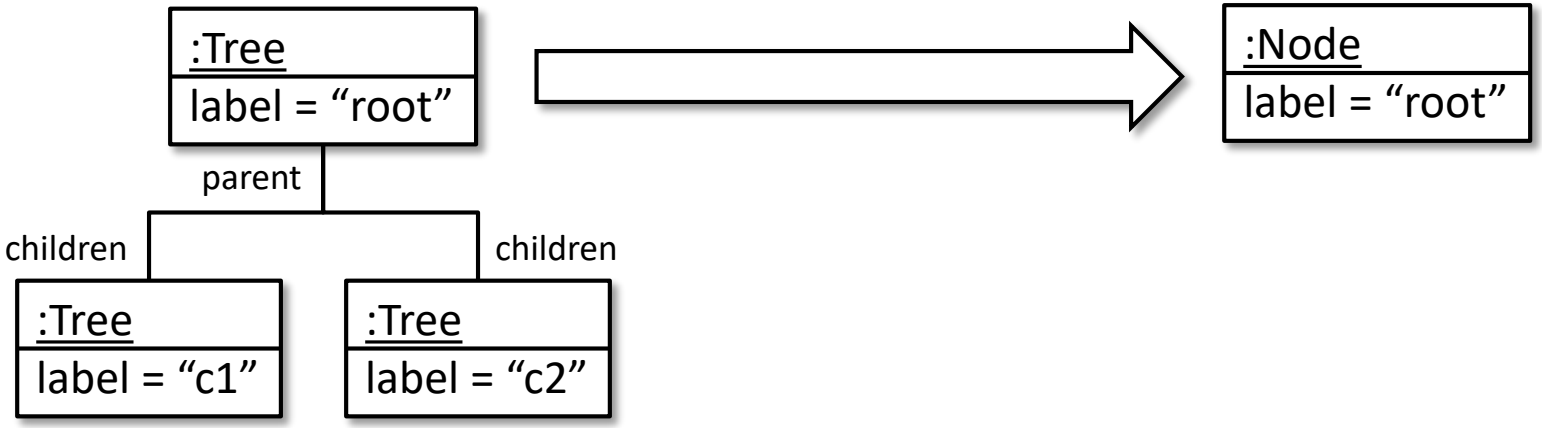

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

Running the transformation



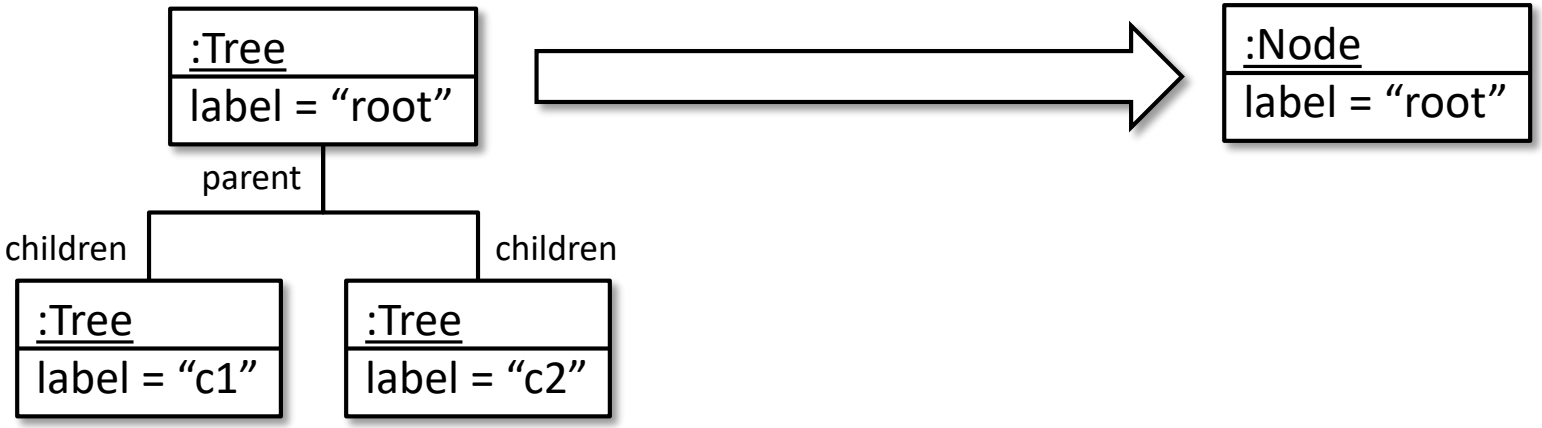

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

Running the transformation



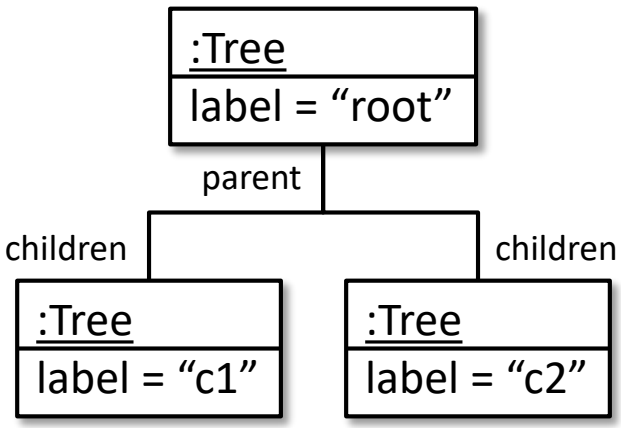

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

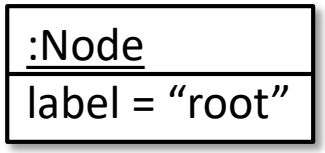
Running the transformation




```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

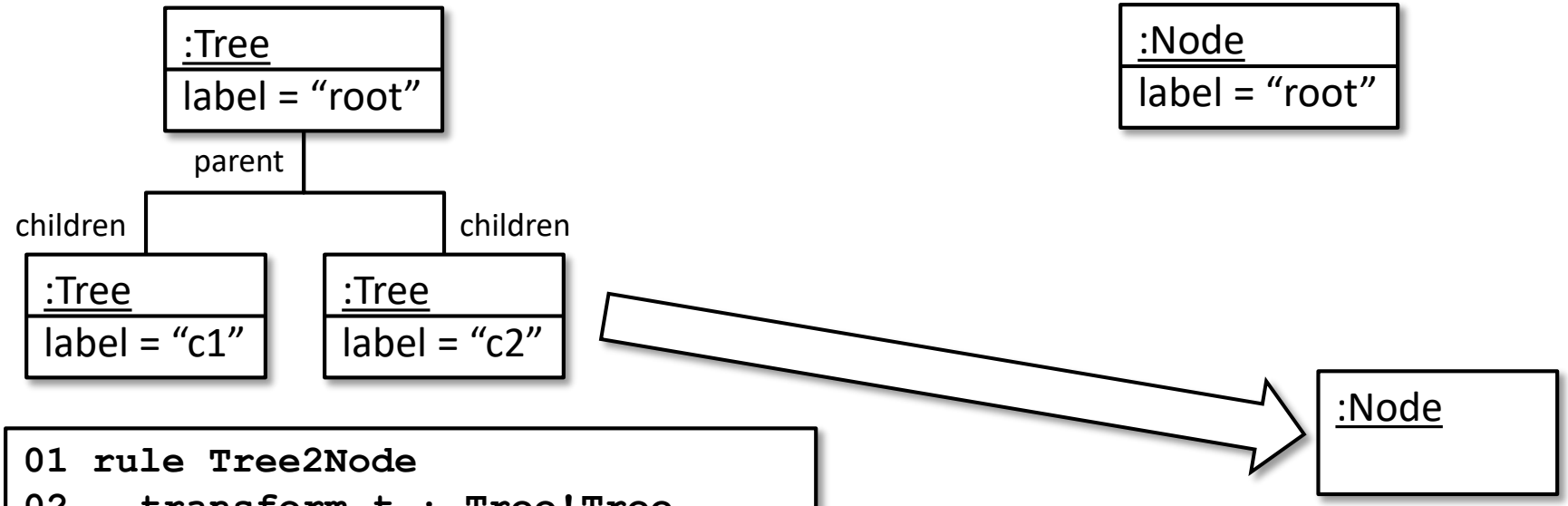
Running the transformation





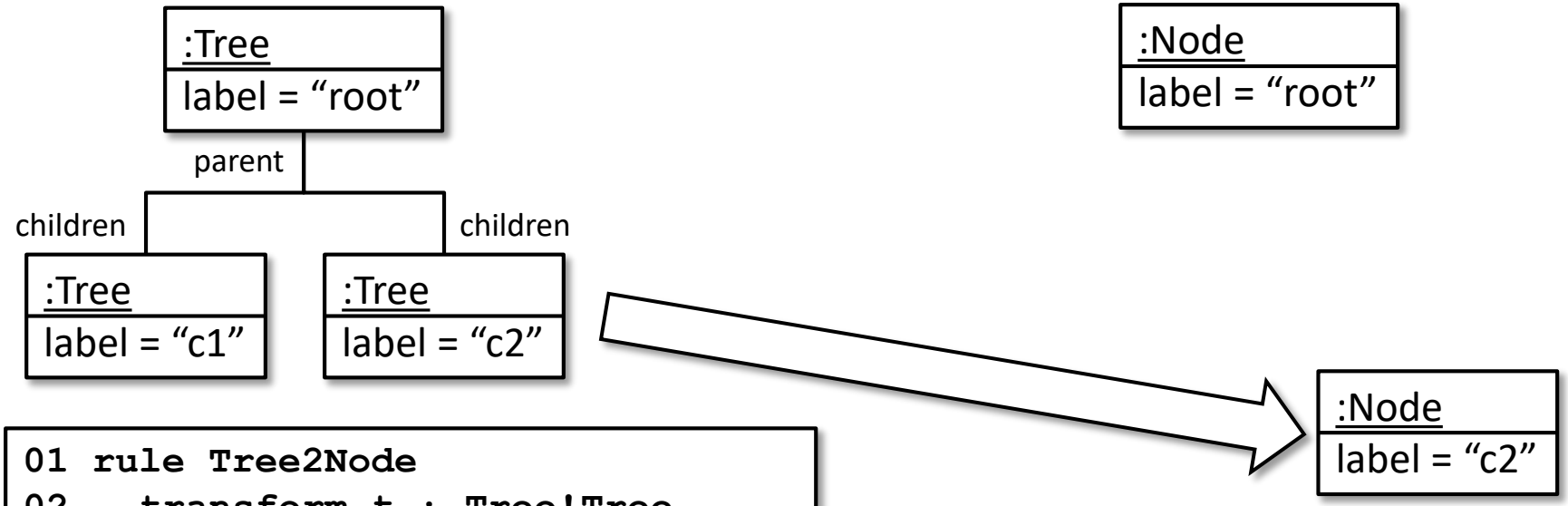

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

Running the transformation



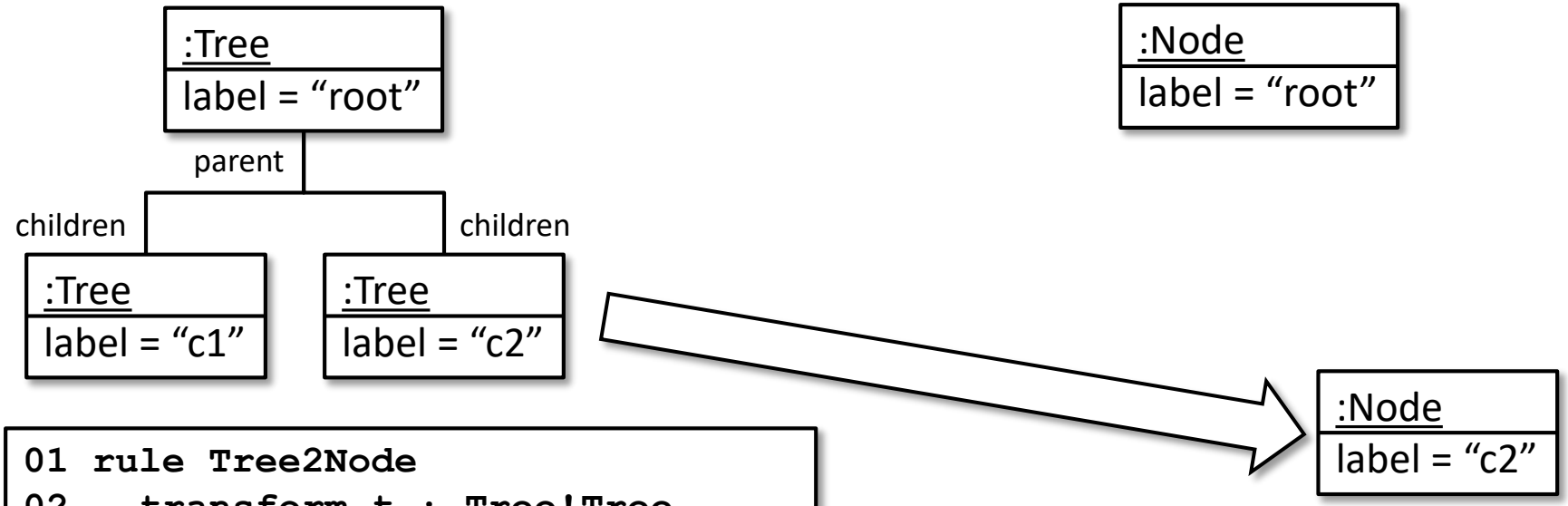

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

Running the transformation



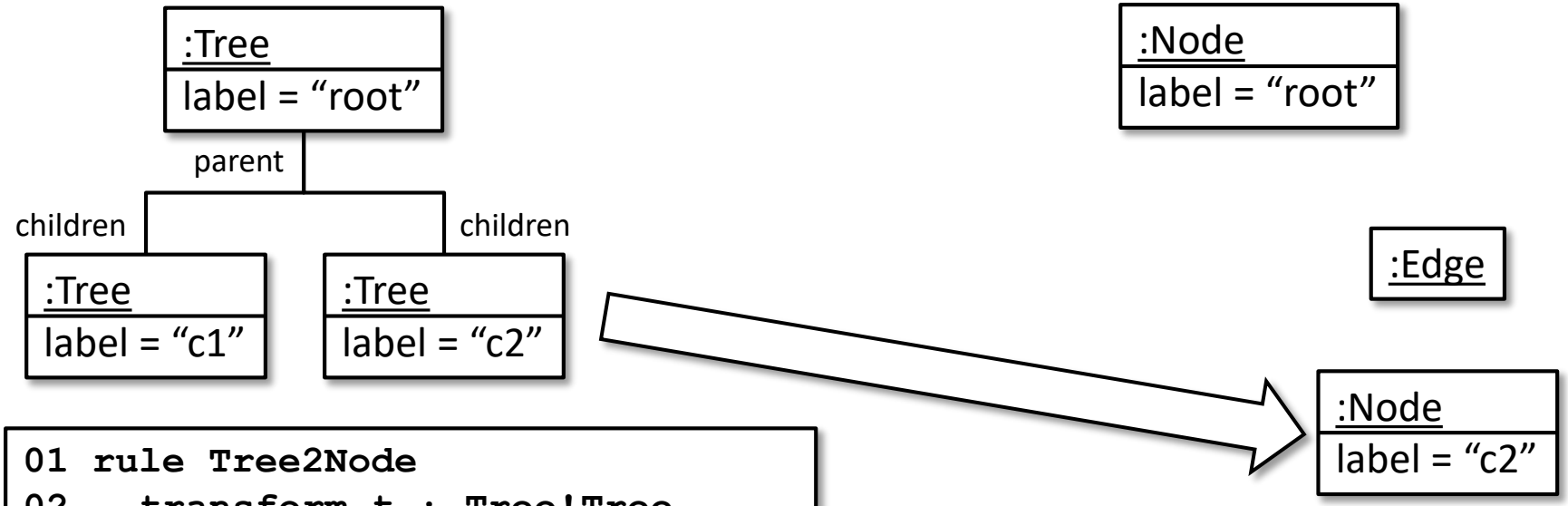

```
01 rule Tree2Node
02   transform t : Tree!Tree
03   to n : Graph!Node {
04
05   n.label = t.label;
06
07   if (t.parent.notEmpty()) {
08     var edge = new Graph!Edge;
09     edge.source = n;
10     edge.target ::= t.parent;
11   }
12 }
```

Running the transformation



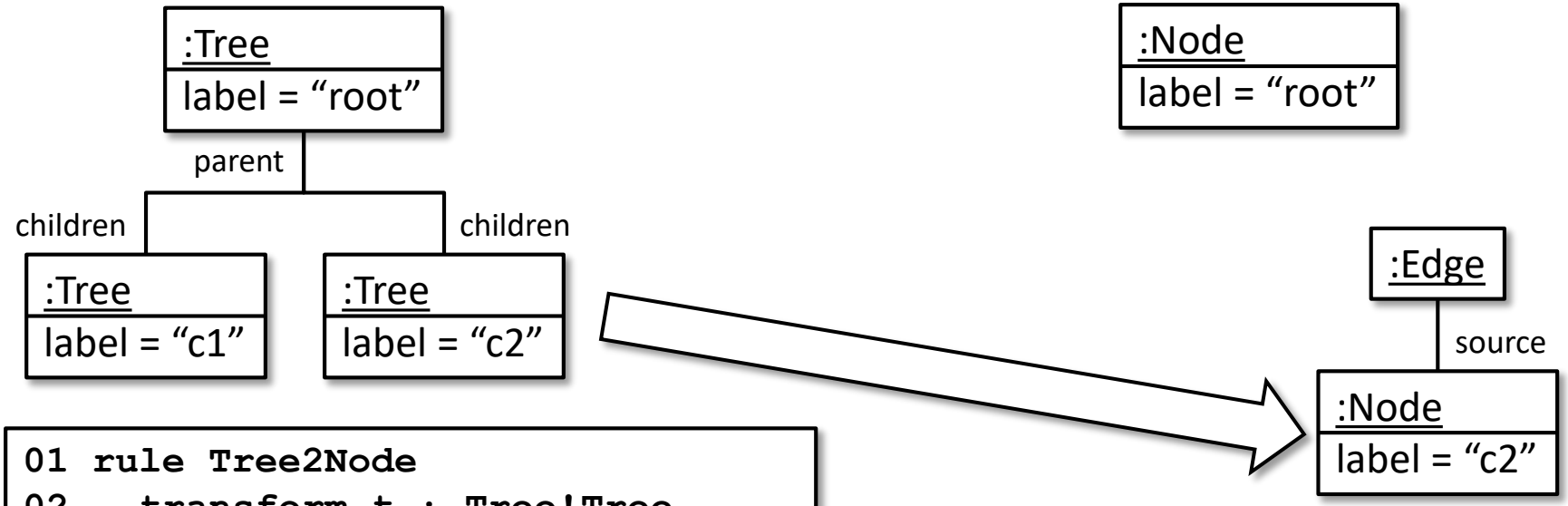

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

Running the transformation



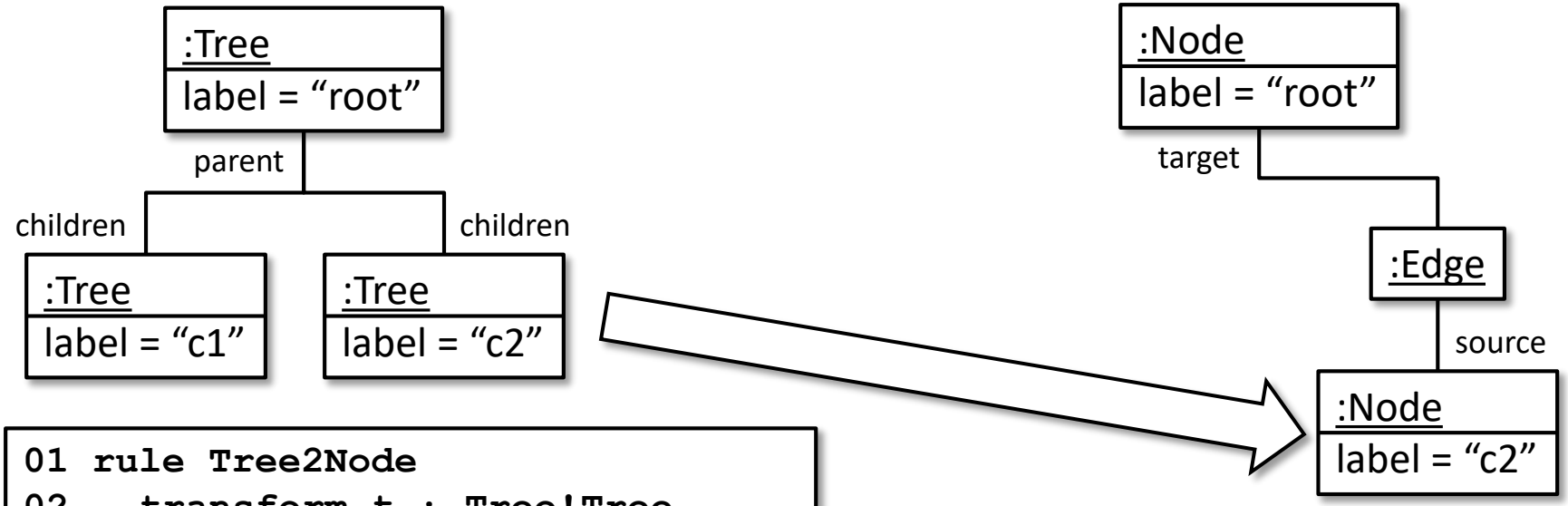

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

Running the transformation



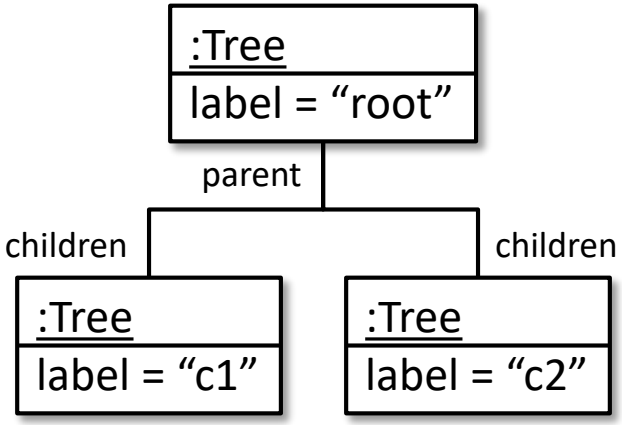

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

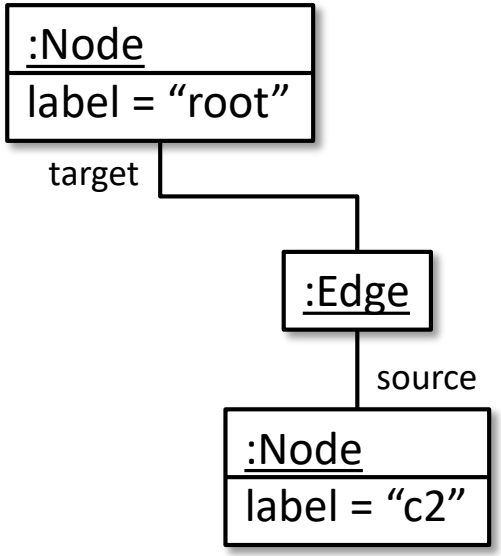
Running the transformation




```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

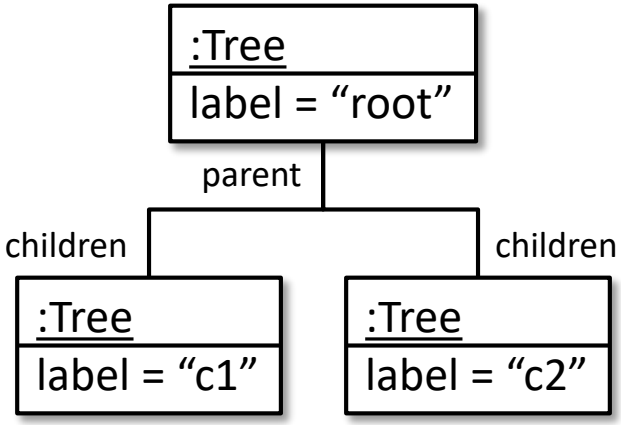
Running the transformation

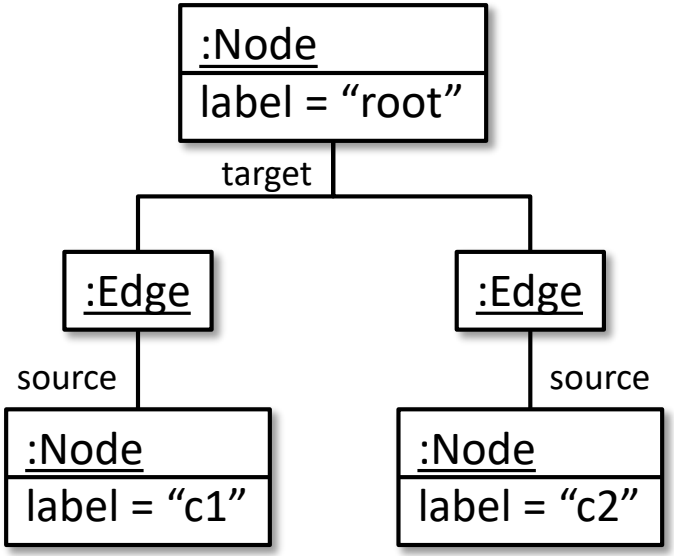





```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

Running the transformation






```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node {

  n.label = t.label;

  if (t.parent.notEmpty()) {
    var edge = new Graph!Edge;
    edge.source = n;
    edge.target ::= t.parent;
  }
}
```

Simpler approach, using UML-RSDS (essentially, OCL)

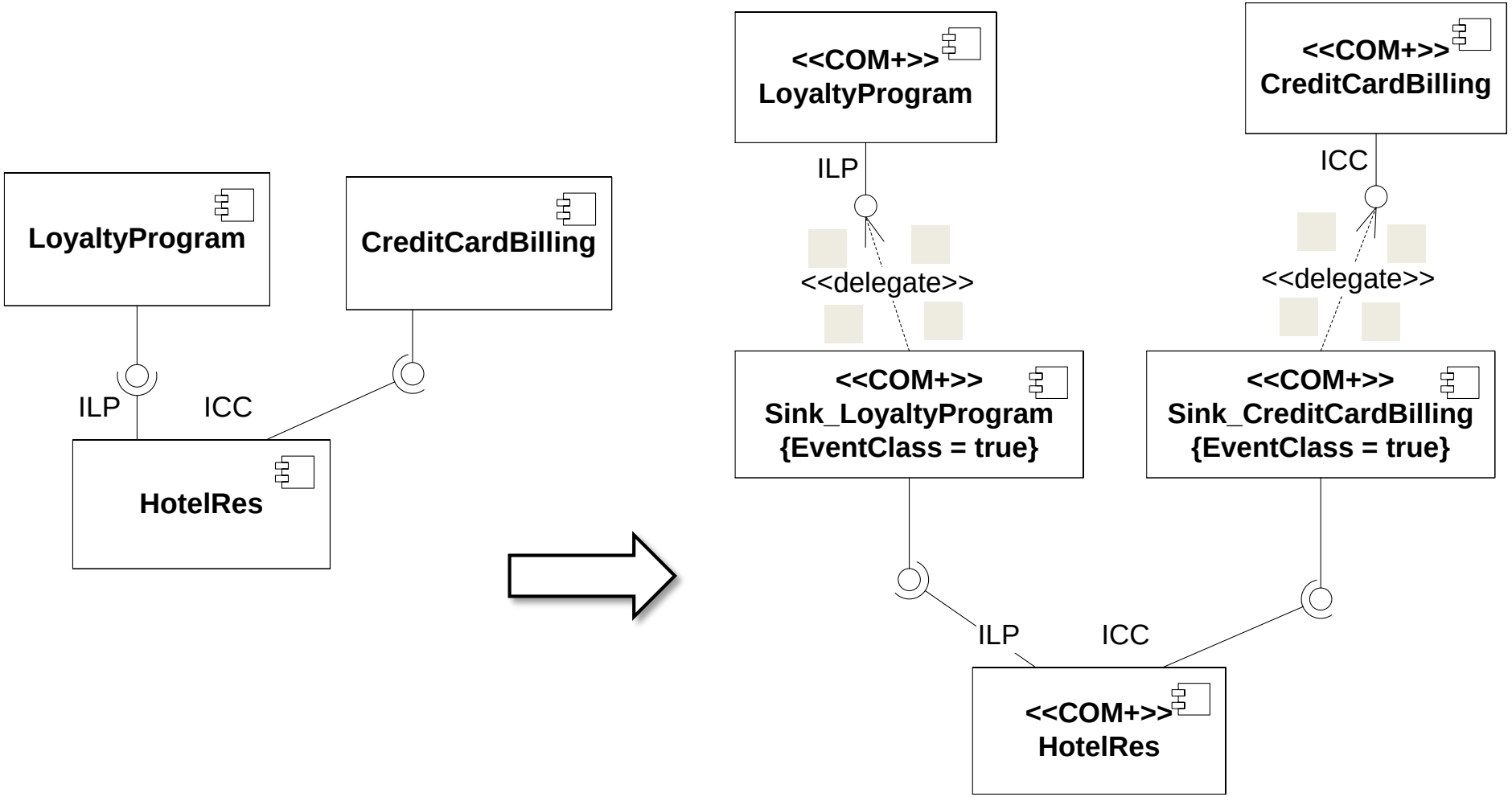
```
Tree::
        Node->exists( n | n.label = label )
```

/ For each tree, create a graph node with the same label (assume labels are unique) */*

```
Tree::
        p : parent   implies
                Edge->exists( edge | edge.source = Node[label]
                                     and
                                     edge.target = Node[p.label] )
```

/ For each tree with a parent p, create an edge from the node corresponding to the tree to the node corresponding to p. E[id] is E-object with key value id */*

More Complex Transformation

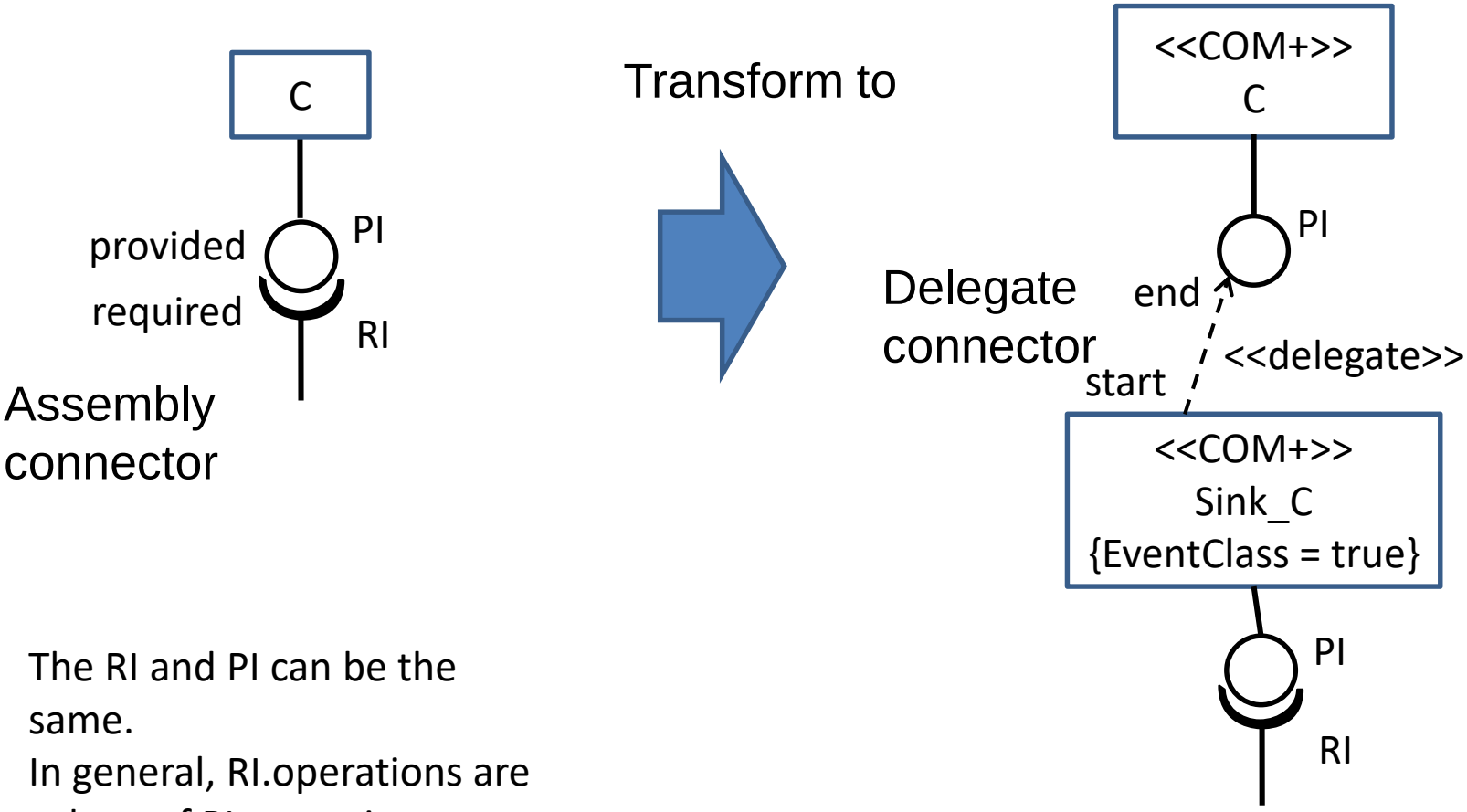


More Complex Transformation

For such transformations, need to think carefully about what the mappings are.

- *UML Components are mapped to COM Components. Gain a "COM+" stereotype*
- *UML Interfaces are mapped to COM Interfaces, with same operations*
- *The COM Components have as required and provided interfaces, the mapped COM Interfaces of the required and provided interfaces of their source UML Components*
- *Each AssemblyConnector is copied, & also mapped to a DelegateConnector and a new Component, which is a sink for the component at the provider end of the original connector. The delegate connects the new component to the mapped provider interface. The new component has the same provided interface.*

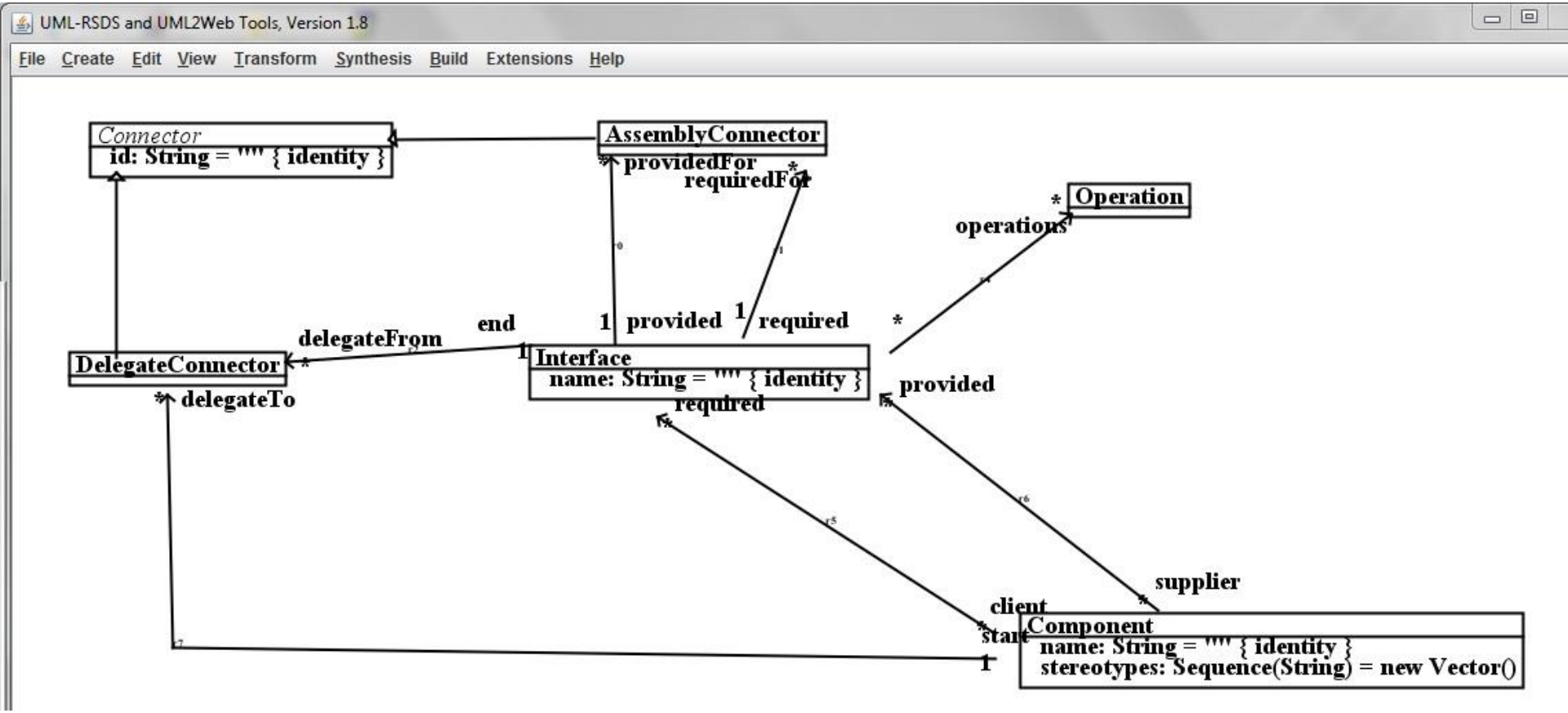
More Complex Transformation



The RI and PI can be the same.
In general, RI.operations are subset of PI.operations

UML Components metamodel (MM1).

COM Components metamodel (MM2) is similar (classes have names COMConnector, etc).



Translating UML architecture to COM

- *UML Components are mapped to COM Components. Gain a "COM+" stereotype*

Component::

```
COMComponent->exists( t | t.name = name and
                          t.stereotypes->includes("COM+"))
```

- *UML Interfaces are mapped to COM Interfaces, with same operations*

Interface::

```
COMInterface->exists(t | t.name = name and
                          t.operations = operations)
```

Translating UML architecture to COM

- *The COM Components have as required and provided interfaces, the mapped COM Interfaces of the required and provided interfaces of their source UML Components*

Component::

c = COMComponent[name] implies

c.provided = COMInterface[provided.name] and

c.required = COMInterface[required.name]

- With these 3 rules, we have a transformation which simply copies the UML components & their interfaces, adding a stereotype to the components.

Translating UML architecture to COM

c1 : Component
c1.name = "HotelRes"
c2 : Component
c2.name = "LoyaltyProgram"
c3 : Component
c3.name = "CreditCardBilling"

i1 : Interface
i1.name = "ILP"
i2 : Interface
i2.name = "ICC“

i1 : c1.required
i2 : c1.required
i1 : c2.provided
i2 : c3.provided

Input model

comix0 : COMInterface
comix0.name = "ILP"
comix_1 : COMInterface
comix_1.name = "ICC"
comcx_0 : COMComponent
comcx_0.name = "HotelRes"
"COM+" : comcx_0.stereotypes
comcx_1 : COMComponent
comcx_1.name = "LoyaltyProgram"
"COM+" : comcx_1.stereotypes
comcx_2 : COMComponent
comcx_2.name = "CreditCardBilling"
"COM+" : comcx_2.stereotypes

comix_0 : comcx_0.required
comix_1 : comcx_0.required
comix_0 : comcx_1.provided
comix_1 : comcx_2.provided

Output model

Translating UML architecture to COM

- *Each AssemblyConnector is mapped to a DelegateConnector and a new Component, which is a sink for the component at the provider end of the original connector. The delegate connects the new component to the mapped provider interface. The new component has the same provided interface.*

```
AssemblyConnector::
        COMDelegateConnector->exists( d |
                COMComponent->exists( s |
                        s.name = “Sink_” + provided.supplier.name  and
                        s.provided = COMInterface[provided.name]  and
                        d.start = s  and
                        d.end = COMInterface[provided.name] ) )
```

Translating UML architecture to COM

- *Also attach stereotypes to the new component*

```
AssemblyConnector::
        COMDelegateConnector->exists( d |
                COMComponent->exists( s |
                        s.name = “Sink_” + provided.supplier.name  and
                        s.provided = COMInterface[provided.name]  and
                        s.stereotypes->includes(“COM+”)  and
                        s.eventClass = true  and
                        d.id = “delegateTo” + provided.supplier.name  and
                        d.start = s  and
                        d.end = COMInterface[provided.name]) )
```

Translating UML architecture to COM

- *Finally, copy the assembly connector*

```
AssemblyConnector::
        COMAssemblyConnector->exists( ac |
                ac.id = id  and
                ac.provided = COMInterface[provided.name]  and
                ac.required = COMInterface[required.name] )
```

Translating UML architecture to COM

ac1 : AssemblyConnector
ac1.id = "ac1"
ac1.provided = i1
ac1.required = i1

ac2 : AssemblyConnector
ac2.id = "ac2"
ac2.provided = i2
ac2.required = i2

Input Model

comdx_0 : COMDelegateConnector
comdx_0.id = "delegateToSink_LoyaltyProgram"

comax_0 : COMAssemblyConnector
comax_0.id = "ac1“

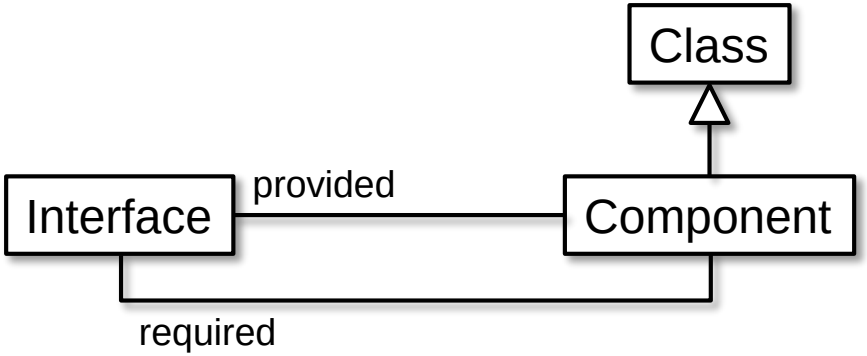
comcx_3 : COMComponent
comcx_3.name = "Sink_LoyaltyProgram"
"COM+" : comcx_3.stereotypes

comdx_0.end = comix_0
comdx_0.start = comcx_3

Output Model

Translating Components using ETL

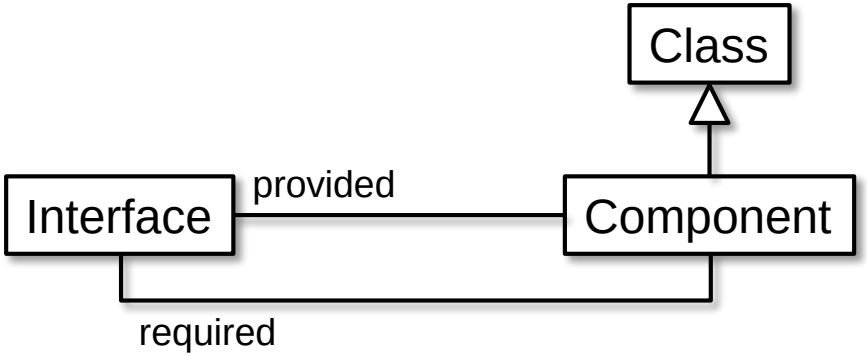
```
rule Component2COMPlus
  transform s : Abstract!Component
  to t : Concrete!Component {

  t.name = s.name;

  t.required ::= s.required;

  -- Mark Component with COM+ stereotype
  t.applyStereotype ('COM+');
}
```



Translating Components using ETL

```
rule Component2COMPlus
  transform s : Abstract!Component
  to t : Concrete!Component {

  t.name = s.name;

  t.required ::= s.required;

  -- Mark Component with COM+ stereotype
  t.applyStereotype ('COM+');
}
```

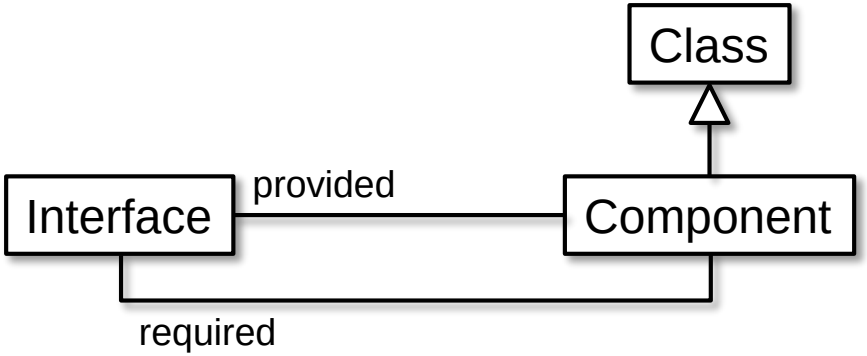
This rule transfers abstract components to concrete COM+ components



Translating Components using ETL

```
rule Component2COMPlus
  transform s : Abstract!Component
  to t : Concrete!Component {

  t.name = s.name;

  t.required ::= s.required;

  -- Mark Component with COM+ stereotype
  t.applyStereotype ('COM+');
}
```

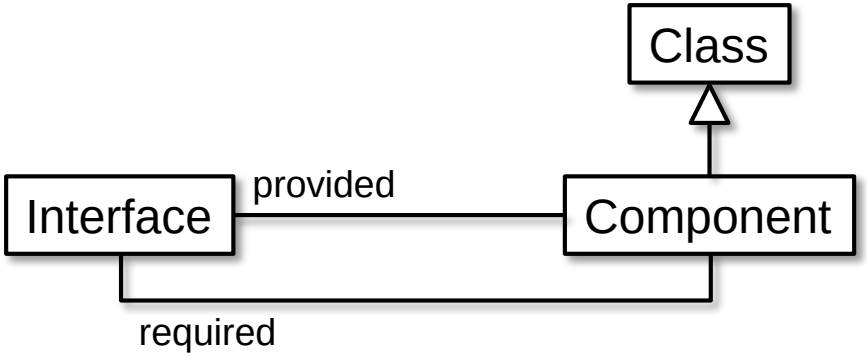
Transform a component from model 'Abstract' to a component in model 'Concrete'



Translating Components using ETL

```
rule Component2COMPlus
  transform s : Abstract!Component
  to t : Concrete!Component {

  t.name = s.name;

  t.required ::= s.required;

  -- Mark Component with (
  t.applyStereotype ('COM-
}
```

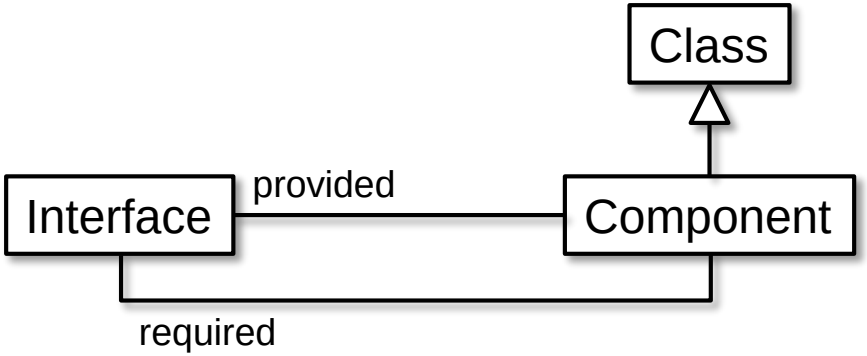
Copy over component name and all required interfaces (note use of ::=)
Provided interfaces are transformed specially



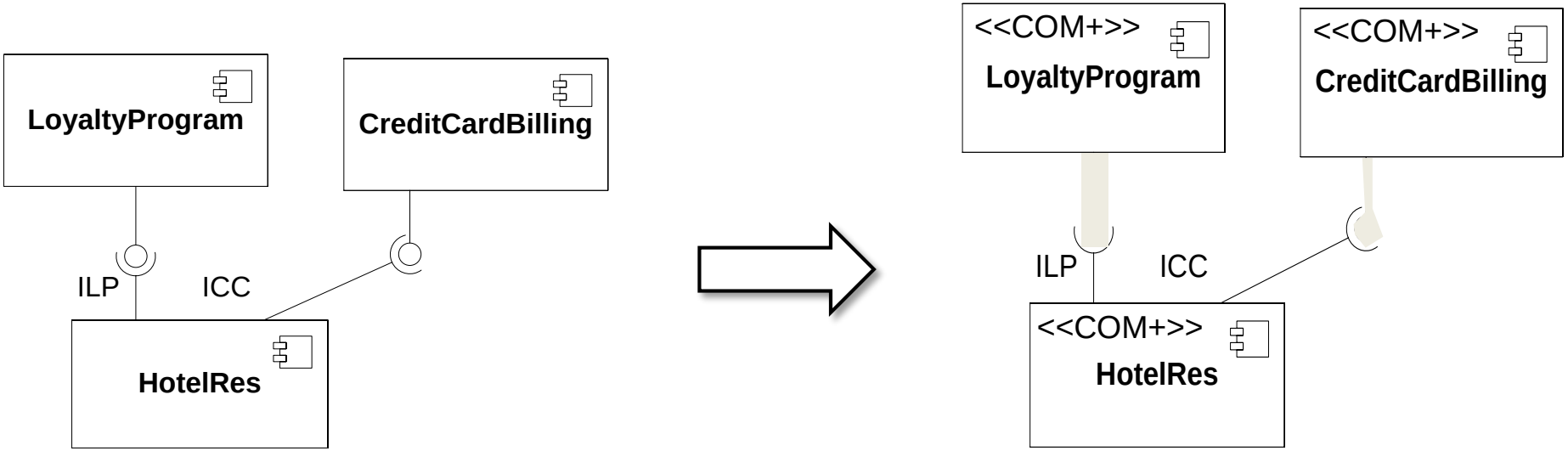
Translating Components

```
rule Component2COMPlus
  transform s : Abstract!Component
  to t : Concrete!Component {

  t.name = s.name;

  t.required ::= s.required;

  -- Mark Component with COM+ stereotype
  t.applyStereotype ('COM+');
}
```

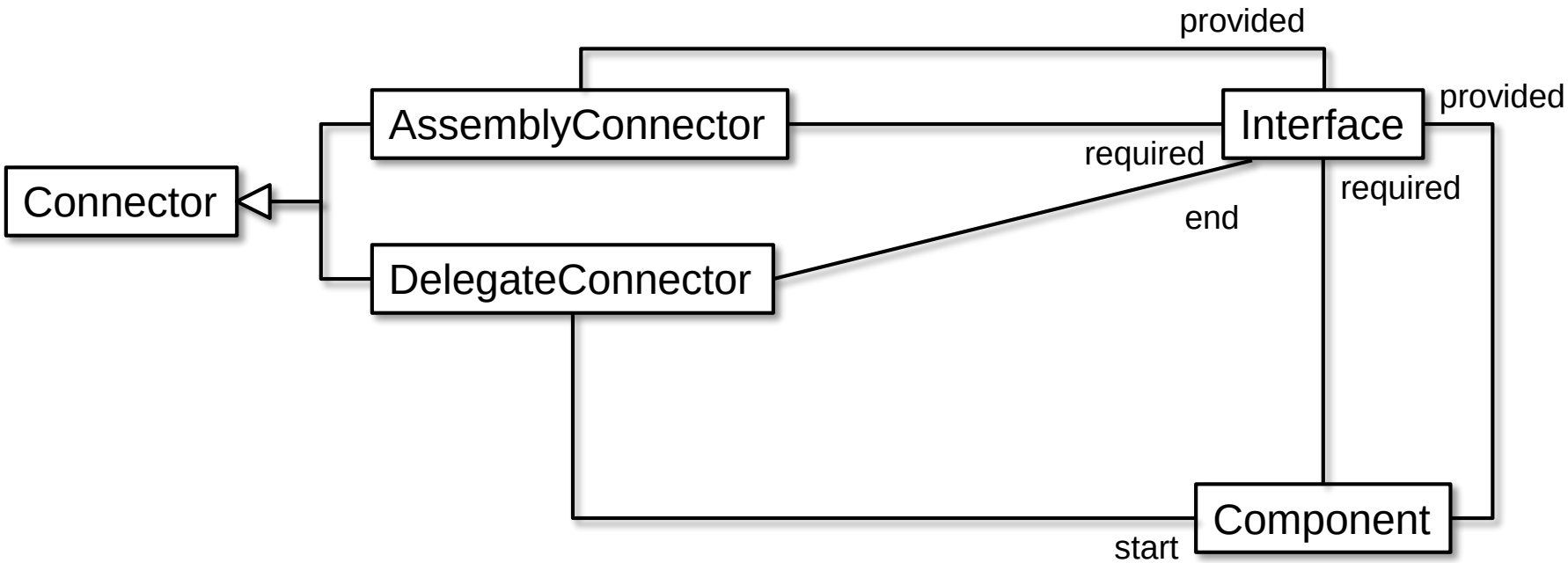
Apply <<COM+>> stereotype
(There is actually a little more technical detail.)



Translating Components (2)



Translating Interfaces



Translating Interfaces (2)

```
rule RequiredInterfaces
  transform s : Abstract!Interface
  to t : Concrete!Interface {

  guard: s.component.required->includes(s);

  t.name = s.name;

  if (s.connector->notEmpty()) {
    -- There is an assembly connector
    var con = new Concrete!AssemblyConnector;
    con.required = t;
    con.provided ::= s.connector.provided;
  }
}
```

Translating Interfaces (2)

```
rule RequiredInterfaces
  transform s : Abstract!Interface
  to t : Concrete!Interface {

  guard: s.component.required->includes(s);

  t.name = s.name;

  if (s.connector->notEmpty()) {
    -- There is an assembly connector
    var con = new Concrete!AssemblyConnector;
    con.required = t;
    con.provided ::= s.connector.provided;
  }
}
```

We use a guard to only transfer required interfaces in this way.

Translating Interfaces (2)

```
rule RequiredInterfaces
  transform s : Abstract!Interface
  to t : Concrete!Interface {

  guard: s.component.required->includes(s);

  t.name = s.name;

  if (s.connector->notEmpty()) {
    -- There is an assembly connector
    var con = new Concrete!AssemblyConnector;
    con.required = t;
    con.provided ::= s.connector.provided;
  }
}
```

We create a new connector.
Provided interfaces will be transformed in a special way.

Translating Interfaces (3)

```
rule ProvidedInterfaces
  transform s : Abstract!Interface
  to t : Concrete!Interface {

  guard: s.component.provided->includes(s);

  t.name = s.name;

  var eventSink = new Concrete!Component;
  eventSink.applyStereotype ('COM+');
  eventSink.setTaggedValue ('EventClass', 'true');

  var intf = new Concrete!Interface;
  intf.name = s.name;

  ...
```

Translating Interfaces (3)

```
rule ProvidedInterfaces
  transform s : Abstract!Interface
  to t : Concrete!Interface {

  guard: s.component.provided->includes(s);

  t.name = s.name;

  var eventSink = new Concrete!Component;
  eventSink.applyStereotype ('COM+');
  eventSink.setTaggedValue ('EventClass', 'true');

  var intf = new Concrete!Interface;
  intf.name = s.name;

  ...
```

We use a guard to only transfer provided interfaces in this way.

Translating Interfaces (3)

```
rule ProvidedInterfaces
  transform s : Abstract!Interface
  to t : Concrete!Interface {

  guard: s.component.provided->includes(s);

  t.name = s.name;

  var eventSink = new Concrete!Component;
  eventSink.applyStereotype ('COM+');
  eventSink.setTaggedValue ('EventClass', 'true');

  var intf = new Concrete!Interface;
  intf.name = s.name;

  ...
```

We create an event sink component.

Translating Interfaces (3)

```
rule ProvidedInterfaces
  transform s : Abstract!Interface
  to t : Concrete!Interface {

  guard: s.component.provided->includes(s);

  t.name = s.name;

  var eventSink = new Concrete!Component;
  eventSink.applyStereotype('COM+');
  eventSink.setTaggedValue('EventClass', 'true');

  var intf = new Concrete!Interface;
  intf.name = s.name;

  ...
```

And another provided interface.

Translating Interfaces (4)

```
...

  var delegate = new Concrete!DelegateConnector;

  delegate.start = eventSink;
  delegate.end = intf;

  s.component.equivalent().provided->append(intf);

  eventSink.provided = Set{t};
}
```

Translating Interfaces (4)

We wire the delegate connector between event sink and the new interface.

```
...

  var delegate = new Concrete!DelegateConnector;

  delegate.start = eventSink;
  delegate.end =  intf;

  s.component.equivalent().provided->append(intf);

  eventSink.provided = Set{t};
}
```

Translating Interfaces (4)

```
...

  var delegate = new Concrete!DelegateConnector;

  delegate.start = eventSink;
  delegate.end =  intf;

  s.component.equivalent().provided->append(intf);

  eventSink.provided = Set{t};
}
```

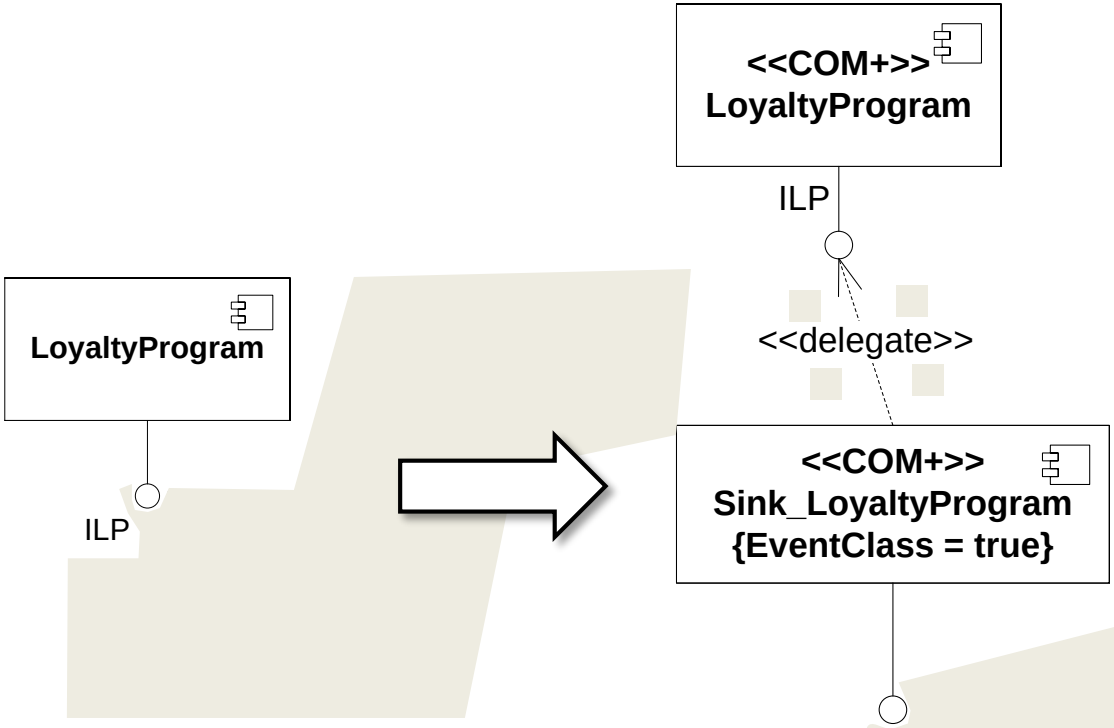
Add new interface to the result of transforming the original component (s.component.equivalent())

Translating Interfaces (4)

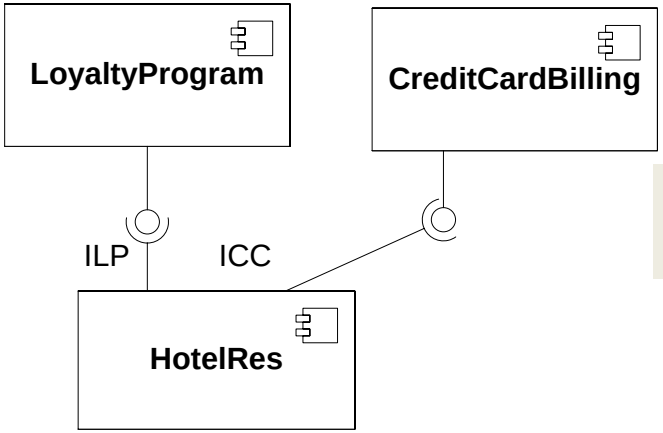
```
...

  var delegate = new Concrete!DelegateConnector;

  delegate.start = eventSink;
  delegate.end = intf;

  s.component.equivalent().provided->append(intf);

  eventSink.provided = Set{t};
}
```

Add the event sink to the first interface created by the rule.

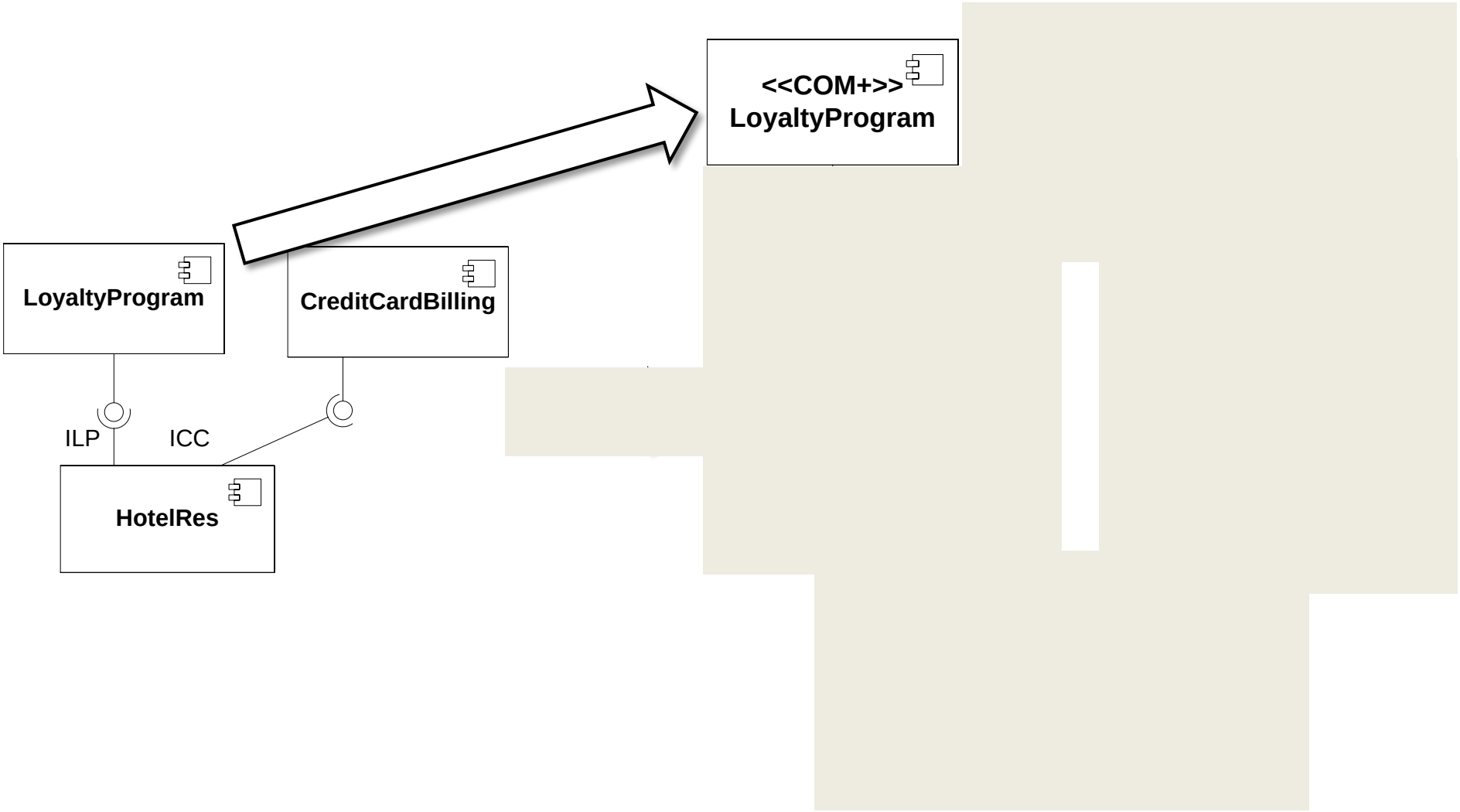
Translating Interfaces (5)



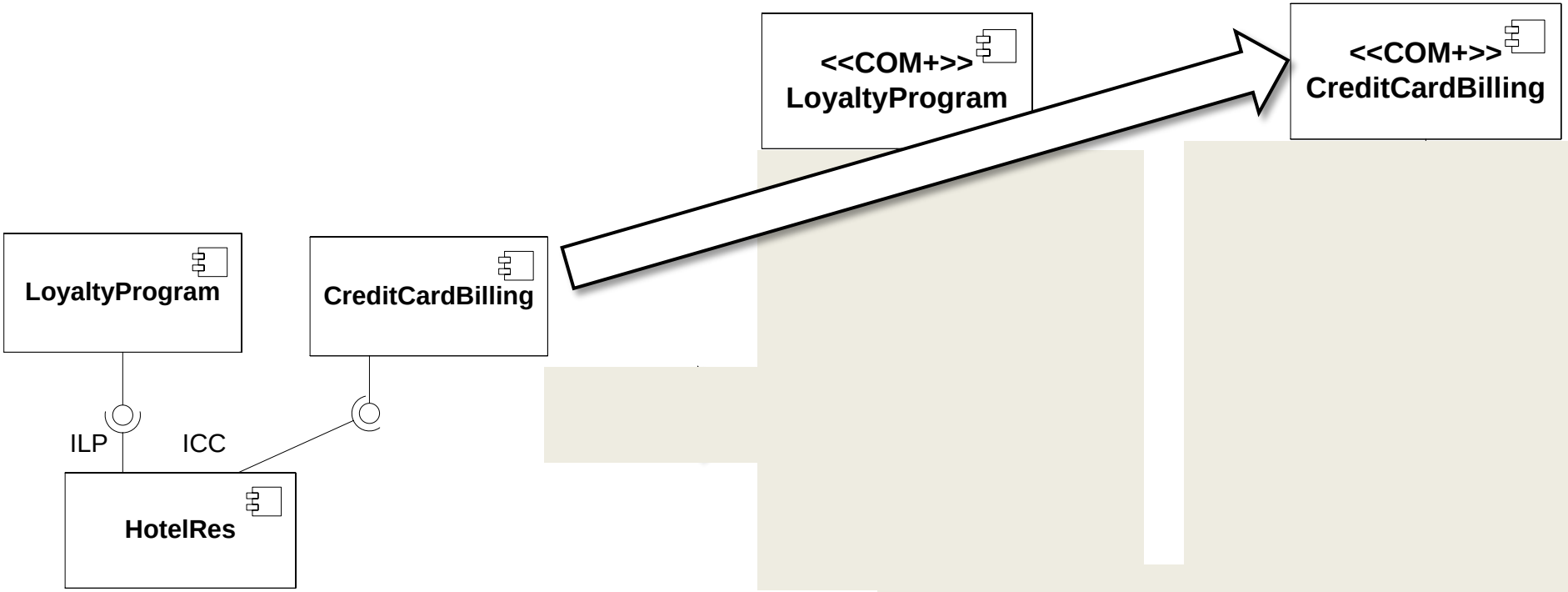
Putting it all together



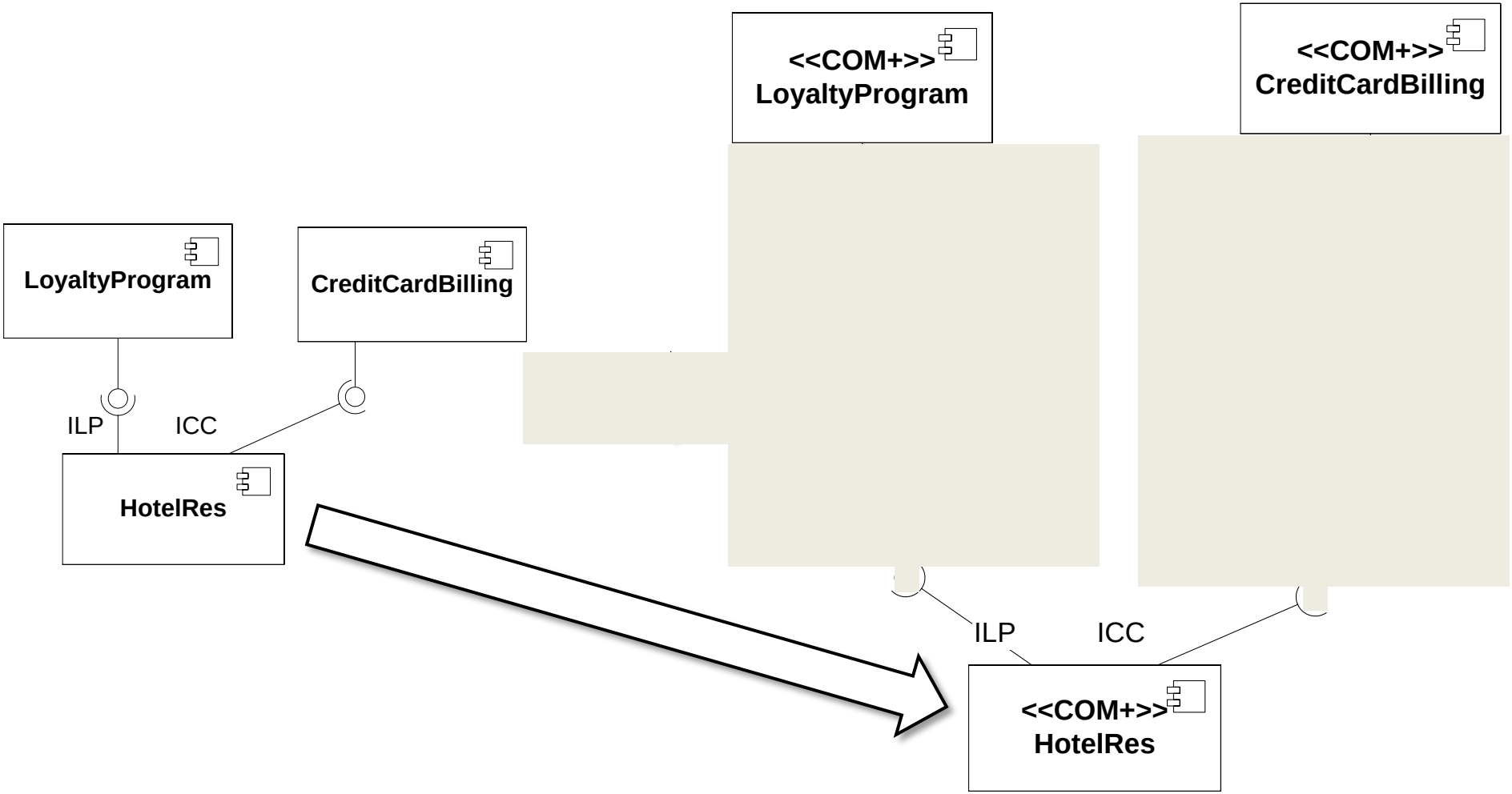
Putting it all together



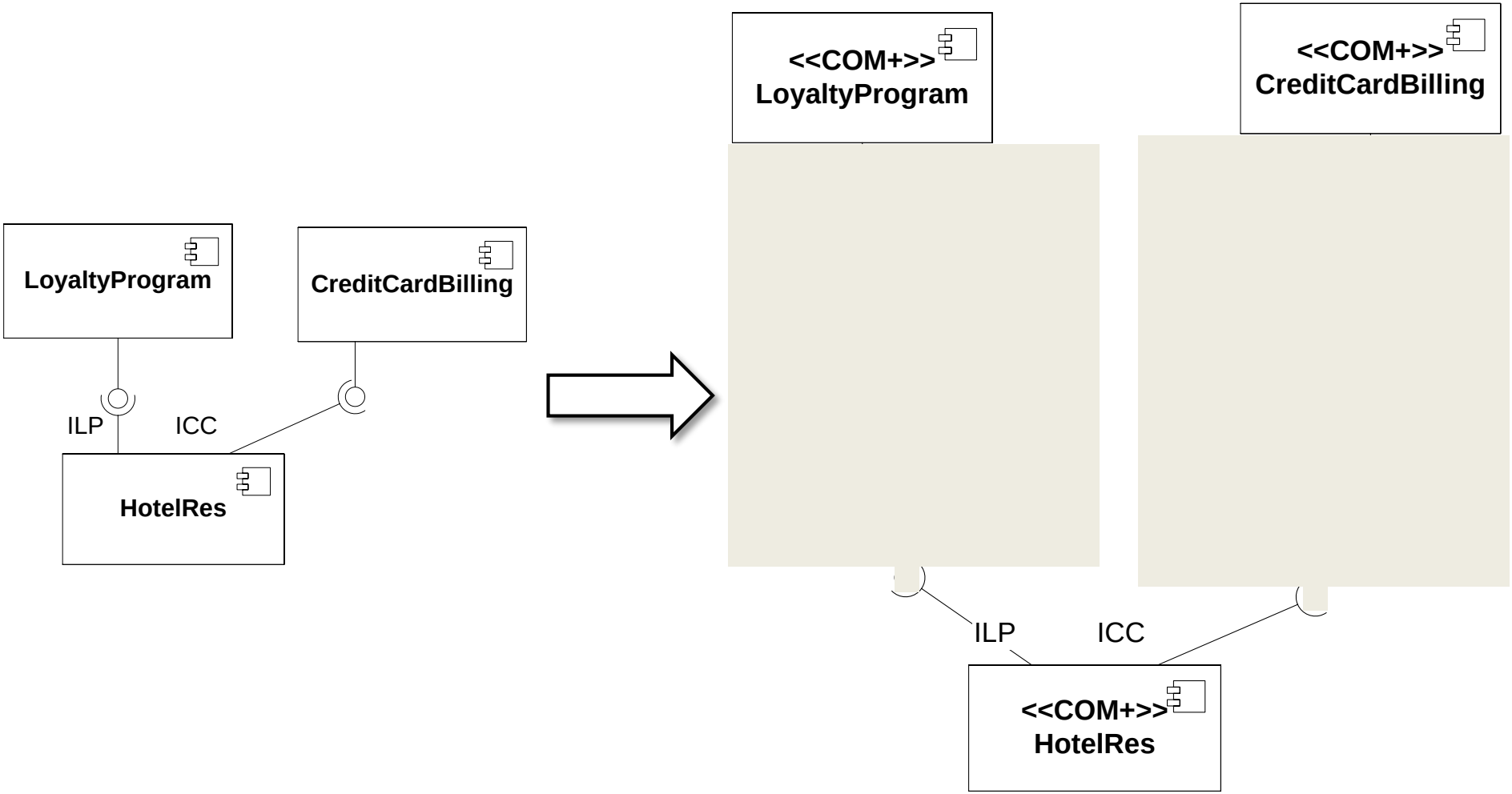
Putting it all together



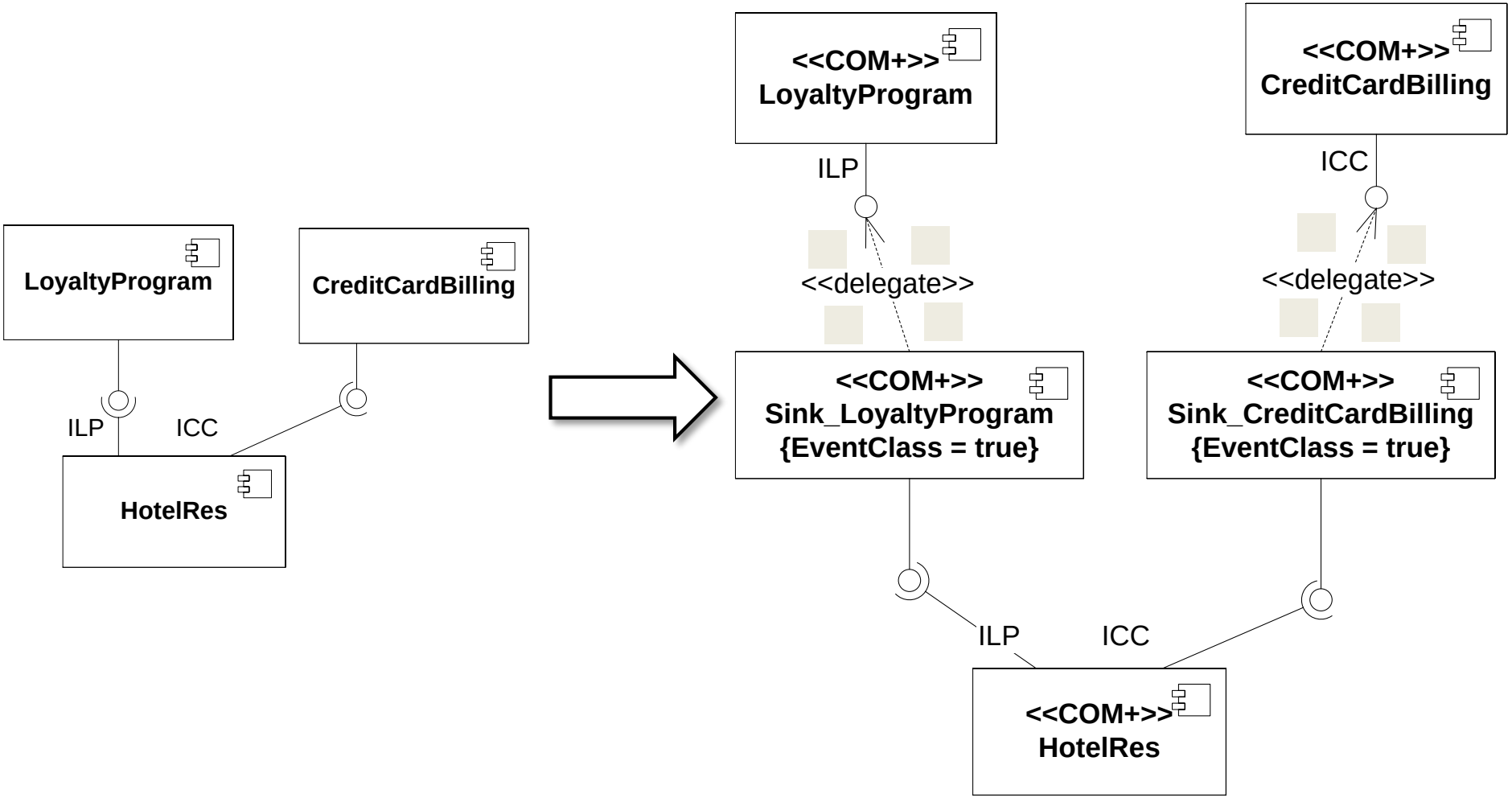
Putting it all together



Putting it all together



Putting it all together



Coming up

- Revision