DTU Informatik
Institut for Informatik og Matematisk Modellering

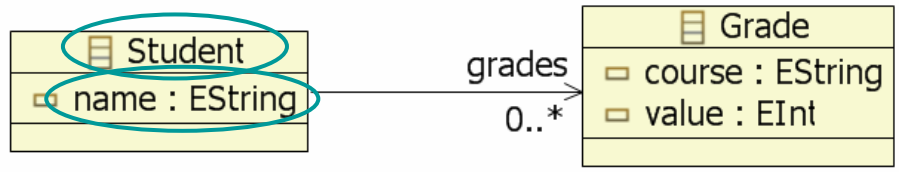
Advanced Topics in Software Engineering
(02265)

Java Emitter Templates: transform models to code!

Patrick Könemann
(pk@imm.dtu.dk)

Feb 25th 2009

Example: Model 2 Code



h o w ?

Idea: use templates!

```
public class Student {

    private String name;

    private List<Grade> grades;

}
```

```
public class Grade {

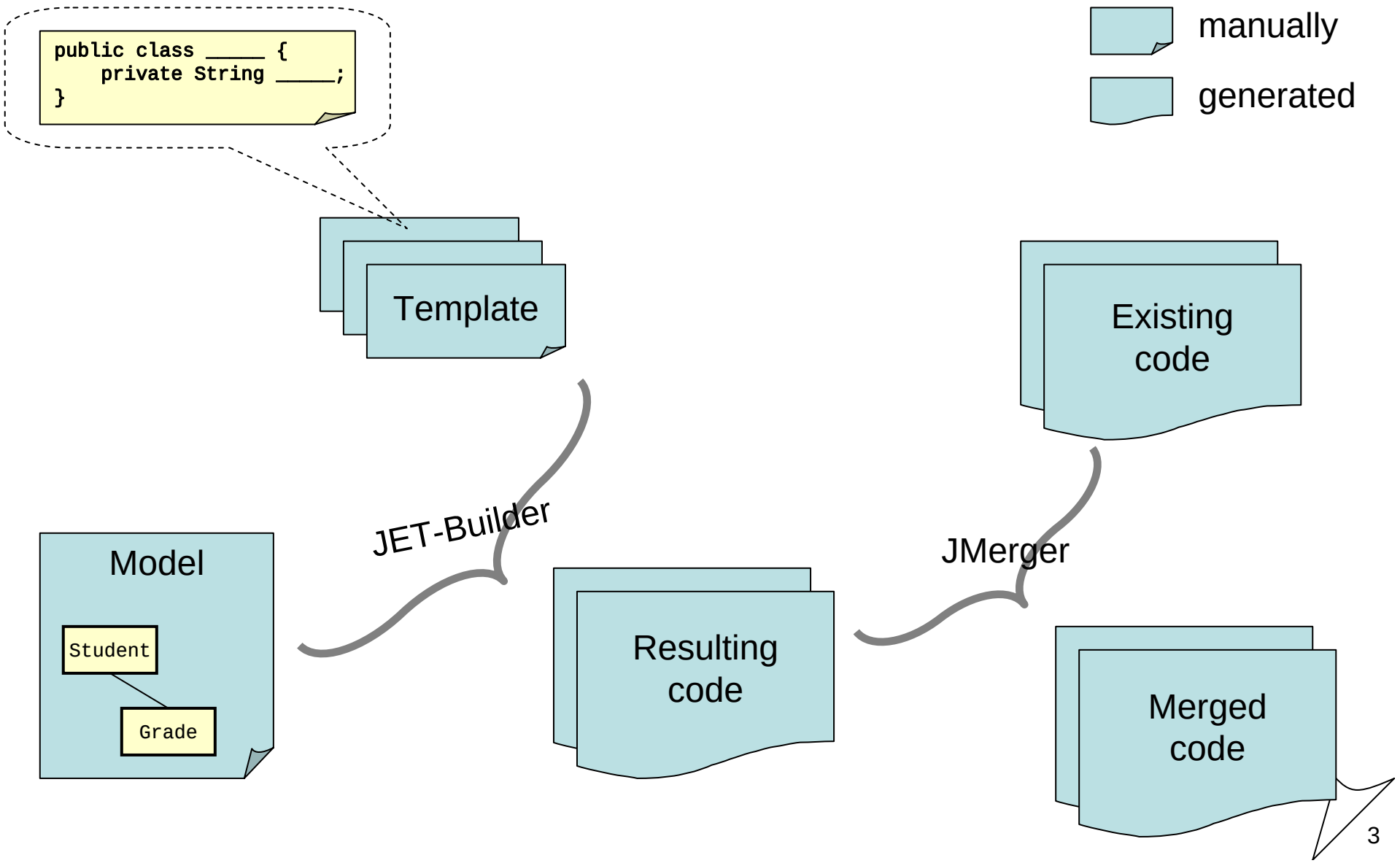
    private String course;

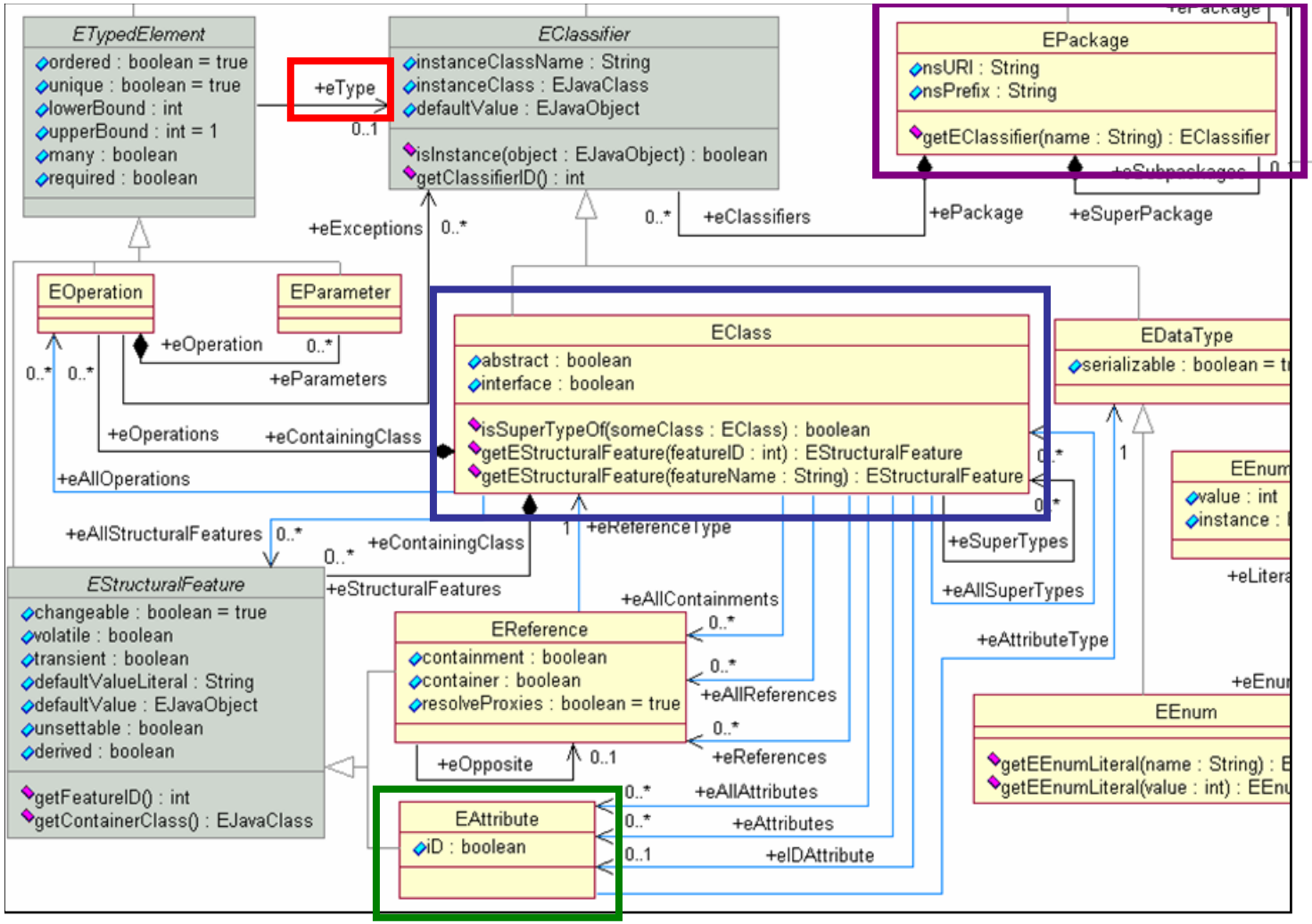
    private int value;

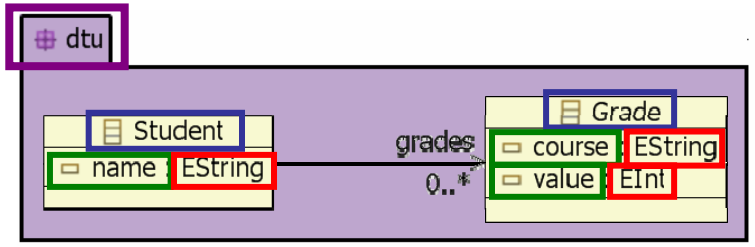
}
```

one template?!

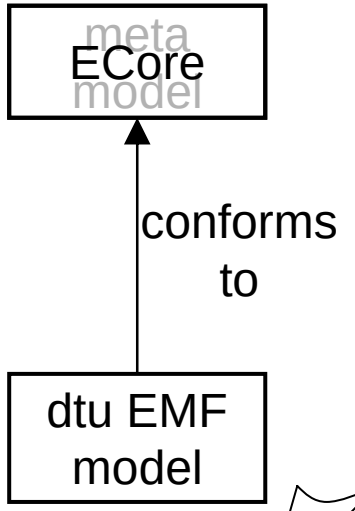
The idea of JET



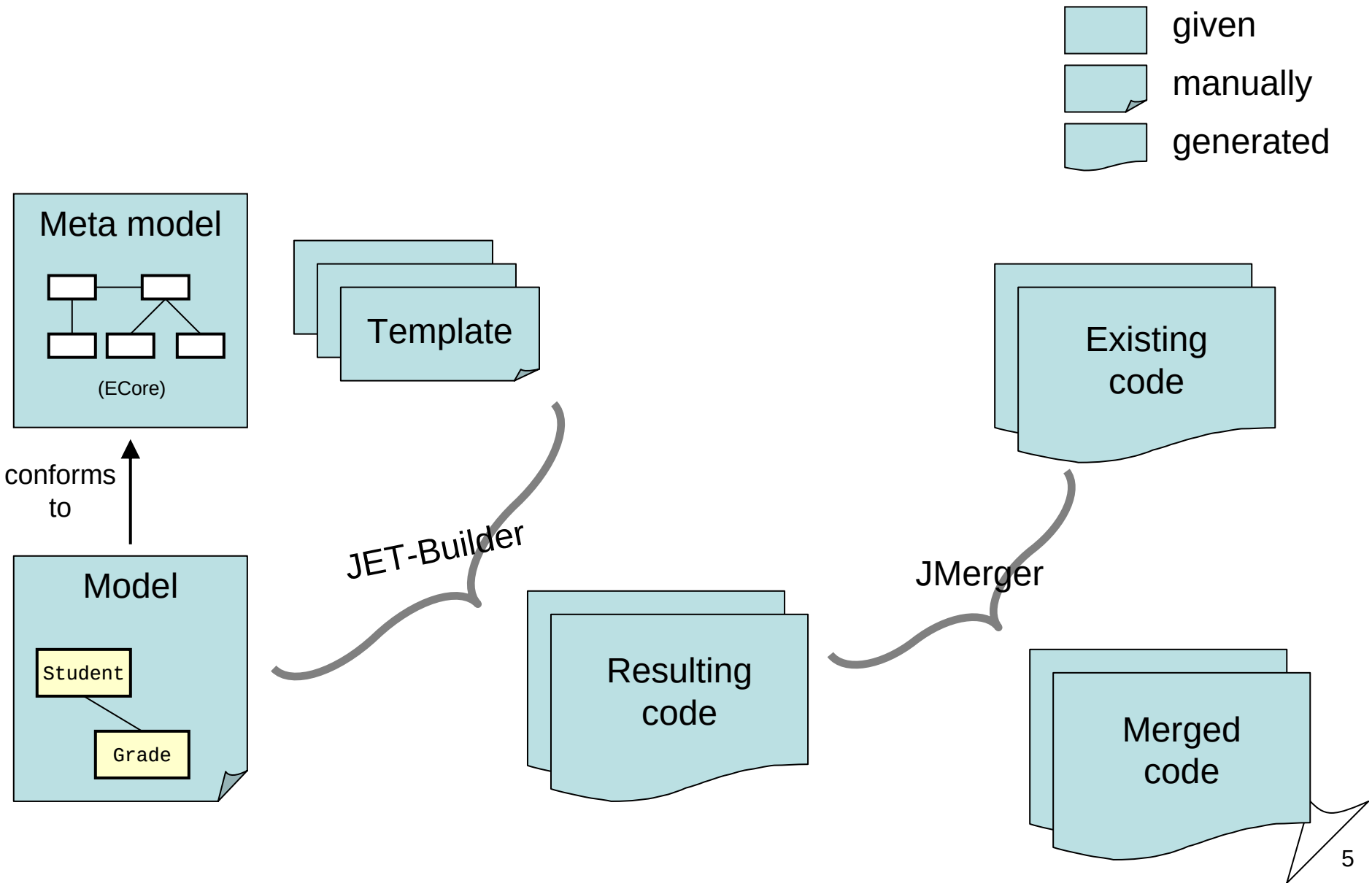




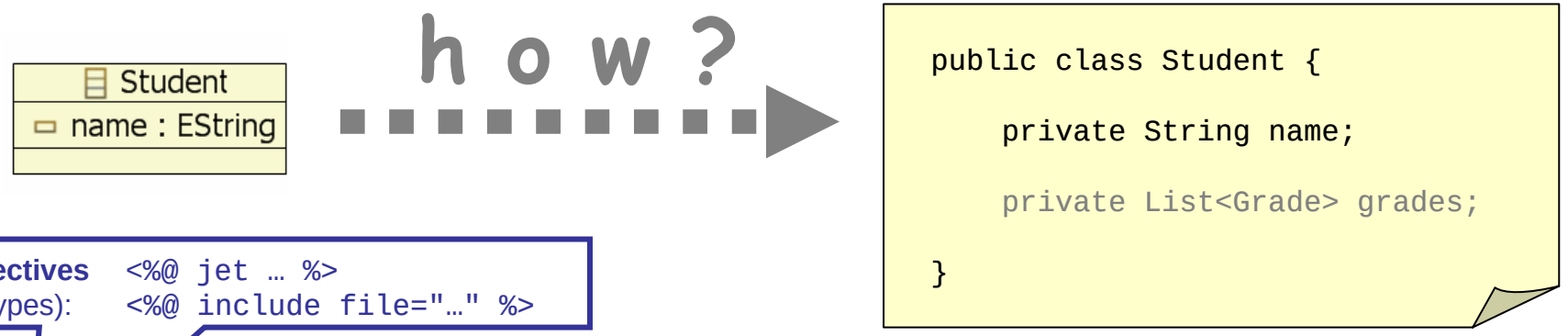
How to access the model?



The idea of JET (cont'd)



Template for result



Directives (2 types): `<%@ jet … %>` `<%@ include file="…" %>`

Scriptlets: `<% (arbitrary java code) %>` e.g.: `<% String s = "Hello, transformation world!"; %>`

```
<%@ jet package="codegen" ... classGenerator" imports="org.eclipse.emf.ecore.*" %>
<% EPackage pack = EmfHelper.loadModel(argument); %>
<% for (EClassifier eClassifier : pack.getEClassifiers()) { %>
public <%=eClassifier.abstract ? "abstract" : "" %> class <%=eClassifier.getName()%> {
  <% for (EAttribute eAttribute : ((EClass)eClassifier).getEAttributes()) { %>
  private <%=eAttribute.getEType().getInstanceTypeName()%> <%=eAttribute.getName()%>;
  <% } %>
}
<% } %>
```

Expressions: `<%= (java expression) %>` e.g.: `<%= (s.length() > 0 ? "<null>" : s + "!") %>`

(notice the similarity to jsp, php, asp, …)

Demo

```
System.out.println(
        new ClassGenerator().
                generate("/GradedStudents/model/students.ecore"));
```

JET language

Predefined variables:

- argument – object to which the transformation is applied to
- stringBuffer – output of the transformation

Quick reference:

- **Directives** <%@ … %> There are two types of directives:
  - `jet` (must be in 1st line!)
    parameters: `package, class, imports, startTag, endTag, skeleton, nlString`
  - `include` (optional)
    parameters: `file, fail`

- **Expressions** <%= … %> Arbitrary java expressions; examples:
  - `<%=new Date().toString()%>`
  - `<%= myString.length() > 0 ? myString : "<I am a placeholder>" %>`

- **Scriptlets** <% … %> Arbitrary java code; examples:
  - ```
    <% (EClassifier eClassifier : pack.getEClassifiers()) { %>
    ...
    <% } %>
    ```
  - `<% if (argument instanceof String) stringBuffer.append("argument: " + argument); %>`
    ```

```
<%@ jet package="codegen" class="ClassGenerator" imports="org.eclipse.emf.ecore.*" %>

<% EPackage pack = EmfHelper.loadModel(argument); %>

...
```

(our template)

Use a *skeleton* to generate the generator class

```
public class CLASS
{
  public String generate(Object argument)
  {
    return "";
  }
}
```

(default skeleton)

```
package codegen;

import org.eclipse.emf.ecore.*;

public class ClassGenerator
{
  public static synchronized ClassGenerator create()
  {
    ClassGenerator result = new ClassGenerator();
    return result;
  }

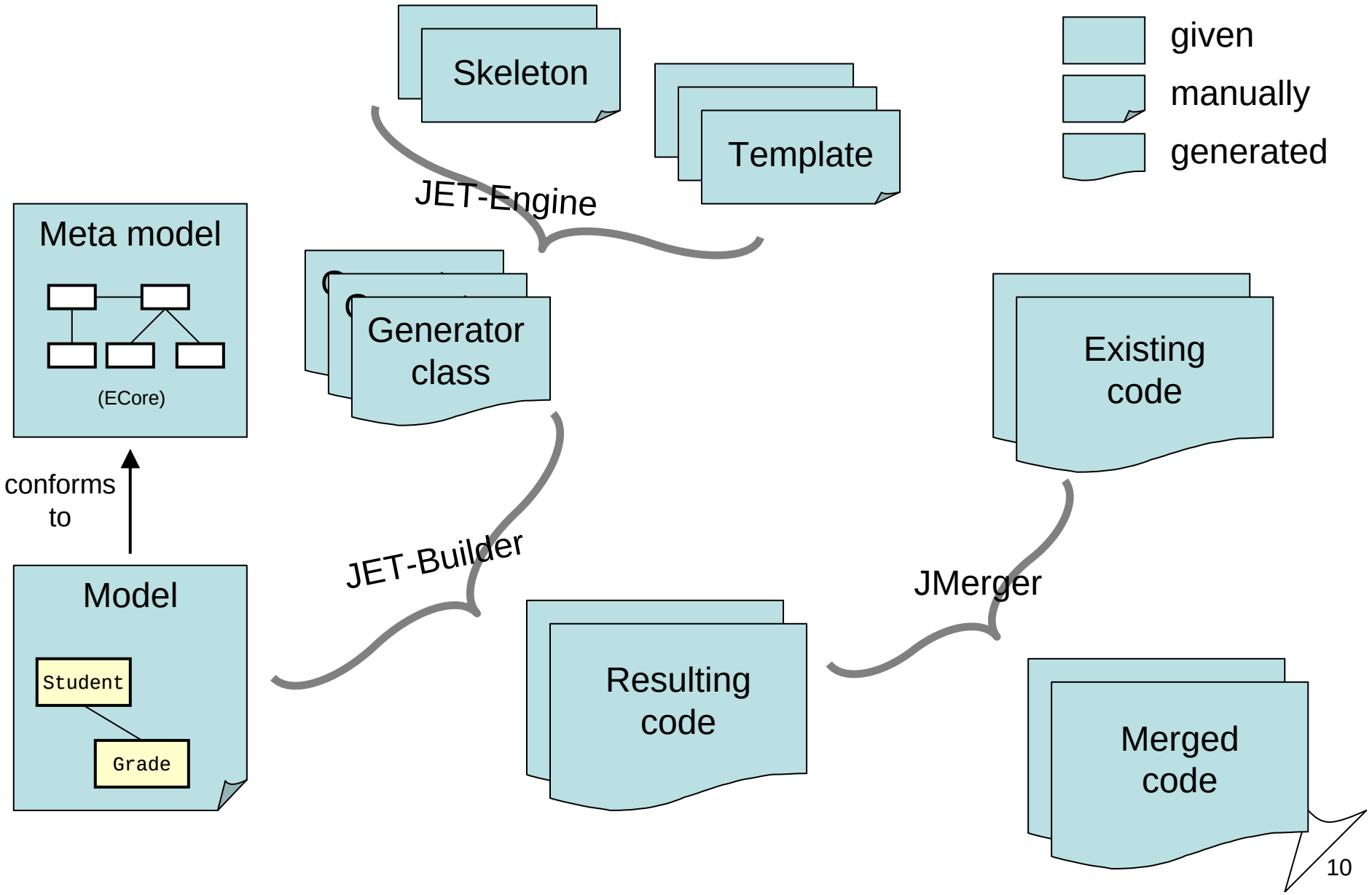
  protected final String TEXT_1 = "";
  protected final String TEXT_2 = "public class ";
  ...

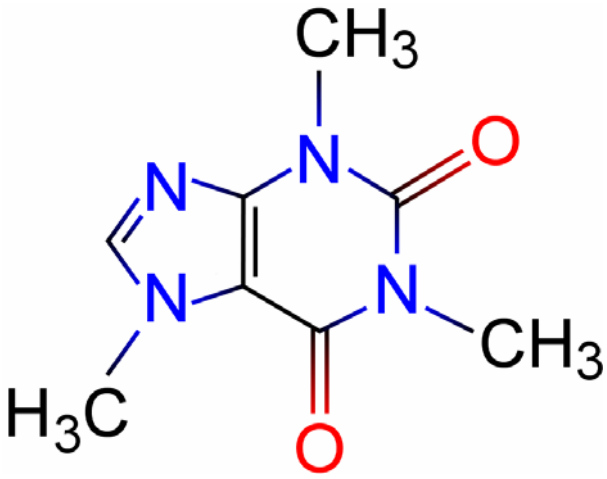
  public String generate(Object argument)
  {
    final StringBuffer stringBuffer = new StringBuffer();
    stringBuffer.append(TEXT_1);
    EPackage pack = EmfHelper.loadModel(argument);
    ...
    return stringBuffer.toString();
  }
}
```

(this was executed in the demo)

Behind the scenes: Generating generators

The idea of JET (cont'd)[2]





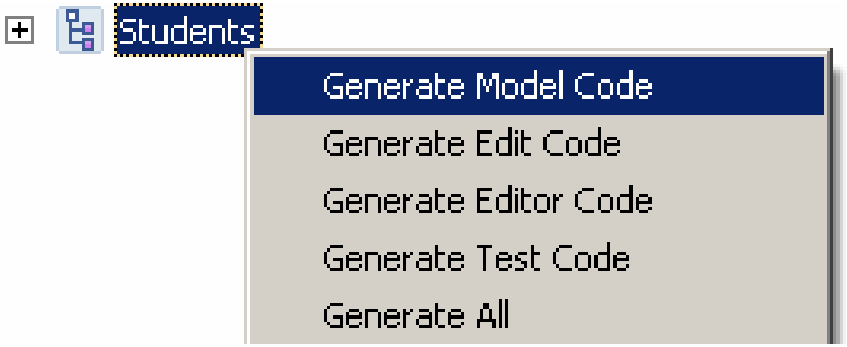
- BREAK

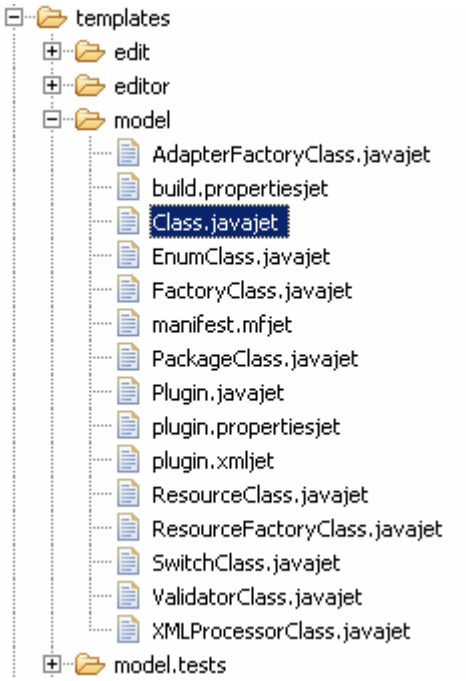
Known uses

- Code generation in Wizards for Plug-In development in Eclipse
- Code generation in EMF / GMF

  E.g. `Class.javajet` has >2000 lines!

Demo

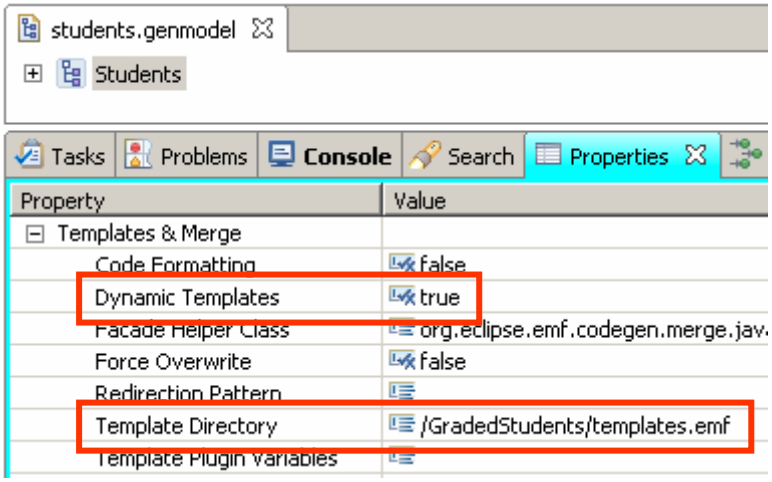




Example: extend EMF codegen

Dynamic templates:

- ‘override’ default templates (`org.eclipse.emf.codegen.ecore`)
- Store your template correctly:



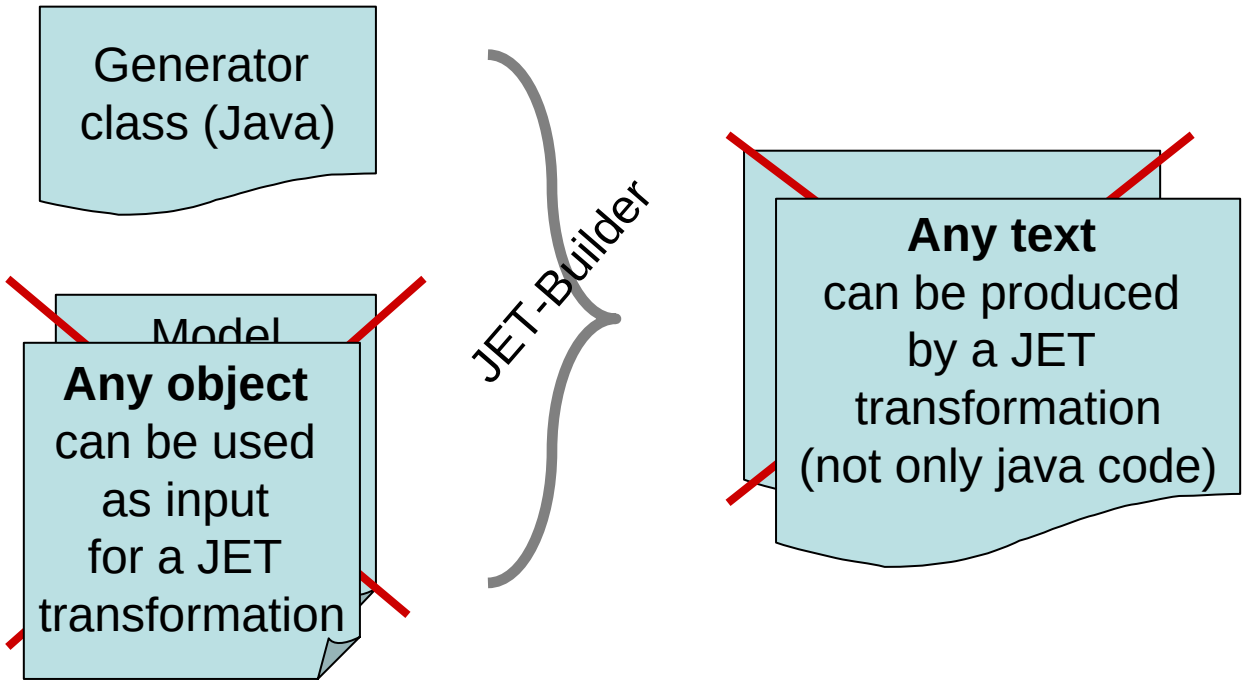
- Excerpt from `Class.javajet` (`includes` are very handy here):

```
<%@ include file="Class/implementedGenOperation.TODO.override.javajetinc" fail="alternative" %>
<%@ start %>
    // TODO: implement this method
    // Ensure that you remove @generated or mark it @generated NOT
    throw new UnsupportedOperationException();
<%@ end %><%//Class/implementedGenOperation.todo.override.javajetinc%>
```

Demo

Advantages of JET

- Java as a powerful template language
- Skeletons for customized generator classes
- Arbitrary source model (e.g. models, arrays / lists, files, …)
- Arbitrary text output (e.g. text-reports / statistics, code, xml, …)



Disadvantages

- Bound to Eclipse
- Compilation of generator classes required
  - Produces a .JETEmitter project for internal purposes
  - If a template is changed, the generator class is overwritten
- Hard to debug
  - No syntax check for templates
    (JET editor with syntax highlighting etc. still buggy)
- Transformation logic and output are mixed up
- Optimized for text output – generating models is more complicated
  (Model-to-model transformation are covered later)

References

- JET project website
  http://www.eclipse.org/modeling/m2t/?project=jet
- Basic JET tutorial, including quick reference
  http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html
- Advanced JET tutorial
  http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html
- FAQs (input models; running jet; extensions; user specific code; …)
  http://wiki.eclipse.org/M2T-JET-FAQ

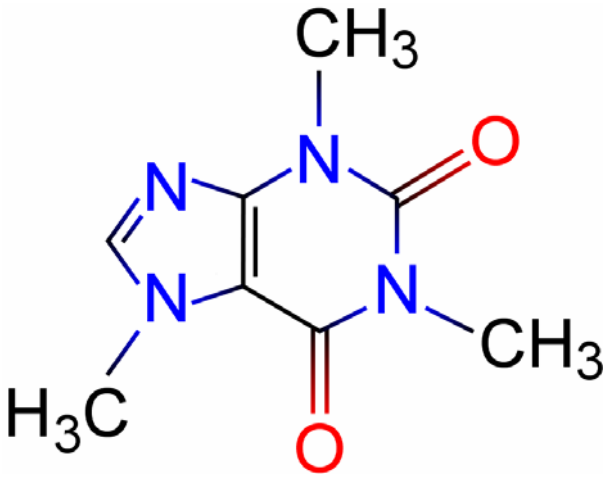
More M2T

- MOF Model To Text Transformation Language (MOFM2T)
  http://www.omg.org/docs/formal/08-01-16.pdf

- Model To Text Language (MTL); (Implementation of MOFM2T)
  http://www.eclipse.org/modeling/m2t/?project=mtl

- Most CASE tools have template-based code generation facilities:
  - openArchitectureWare
  - androMDA
  - JUDE
  - IBM Rational Software Architect*
  - Borland Together*
  - Sparx Systems Enterprise Architect*
  - …

(*comercial tool)

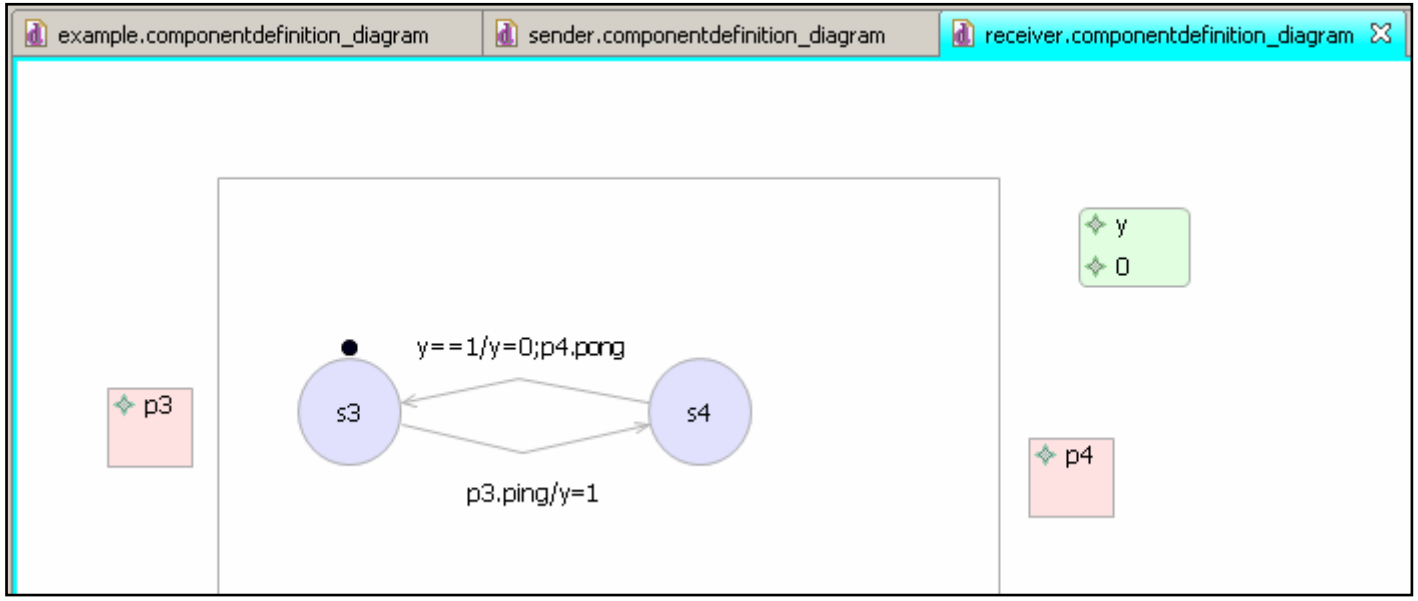


- BREAK

Tutorial / Exercise (1/4)

Lets generate a simple automaton! :o)

This is a component definition including a simple automaton:

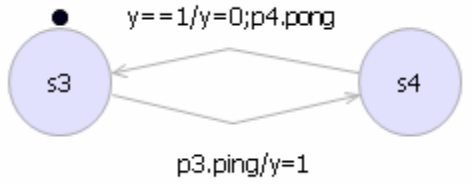


Goal:

Create a jet transformation to generate java classes for component automata!

Tutorial / Exercise (2/4)

The code could look like that:




```
public class ReceiverAutomaton {

        private enum States { s3, s4 }

        private States currentState;

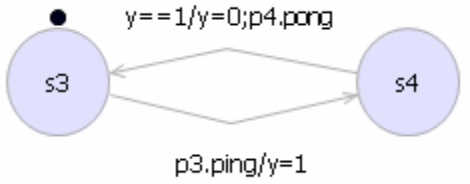
        public ReceiverAutomaton() {
                currentState = States.s3;
        }

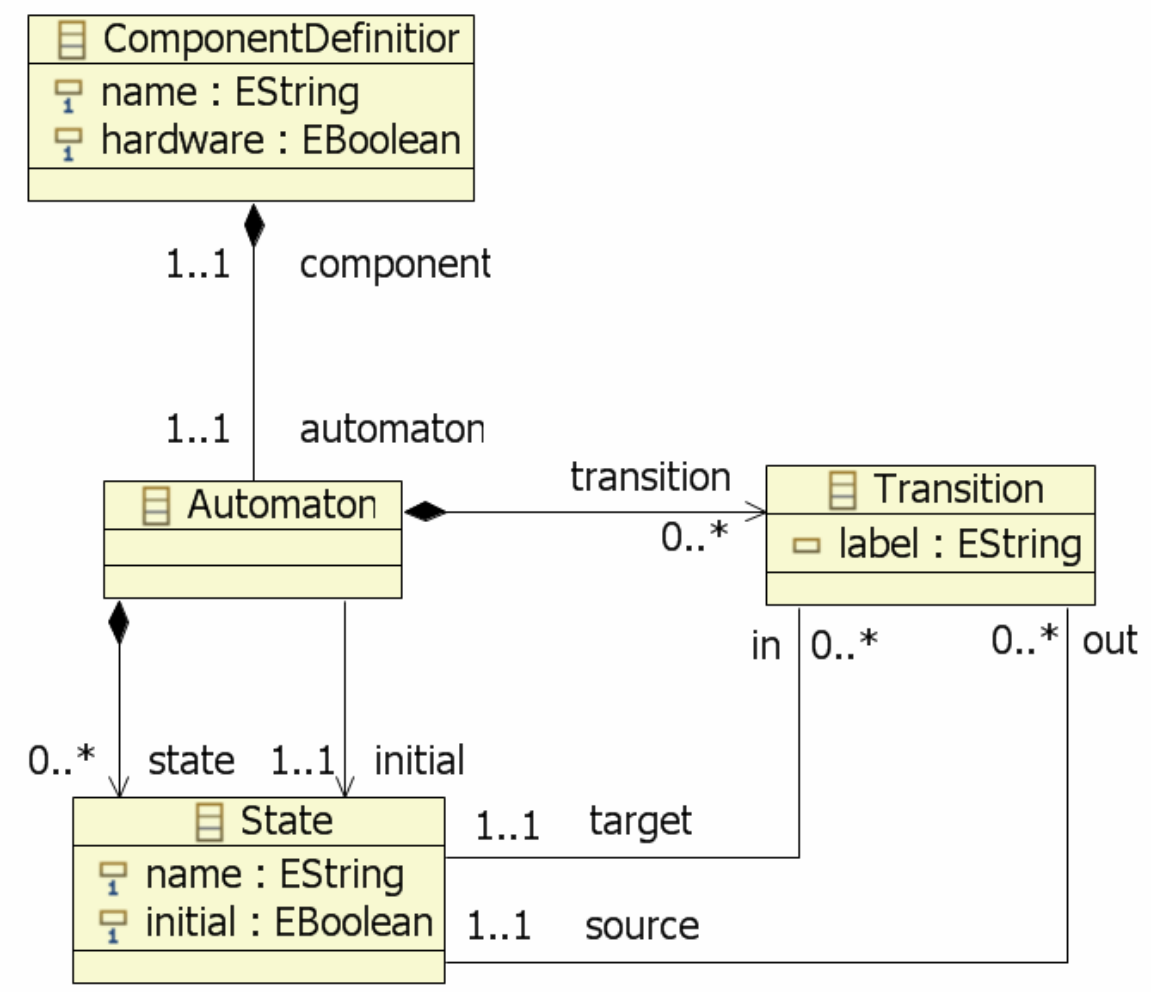
        public boolean step() {
                switch (currentState) {
                case s3:
                        currentState = States.s4;
                        return true;
                case s4:
                        currentState = States.s3;
                        return true;
                }
                return false;
        }
}
```

- Just consider:
  - *states*
  - *state names*
  - *initial state*
  - *transitions*
- Do *not* evaluate transition labels
- If there are more outgoing transitions in a state, just take *any* of them
- For testing purposes, add a `System.out.println()` to print the fired transition

Tutorial / Exercise (3/4)

How to access the automaton:





Tutorial / Exercise (4/4)

- Who does _not_ have a (fellow student's) laptop to work with?
- Follow the instructions for assignment 4
  - Questions? Ask me!
- After 45min, we will discuss your results / do it together