Model-Driven Development with openArchitectureWare

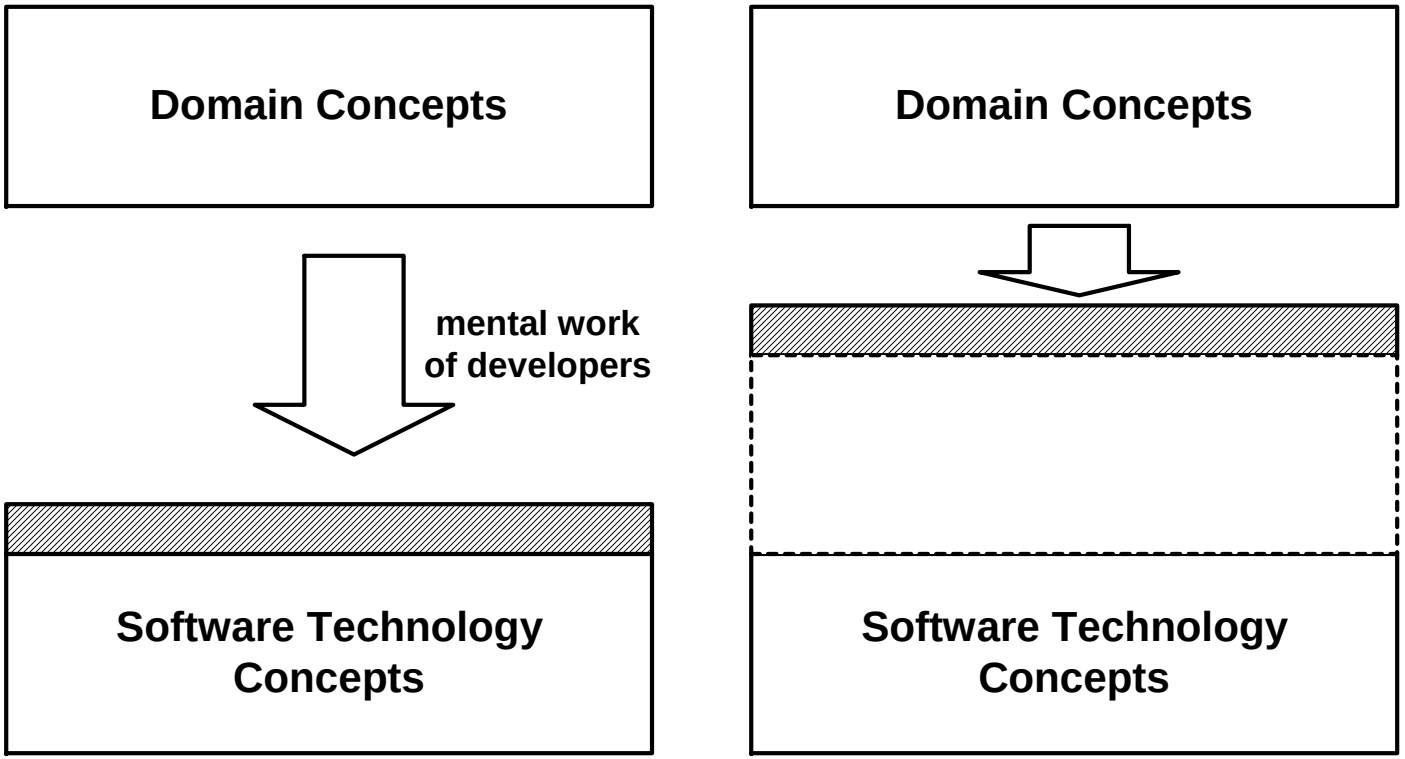
Markus Völter
voelter@acm.org
www.voelter.de

Sven Efftinge
sven@efftinge.de
www.efftinge.de

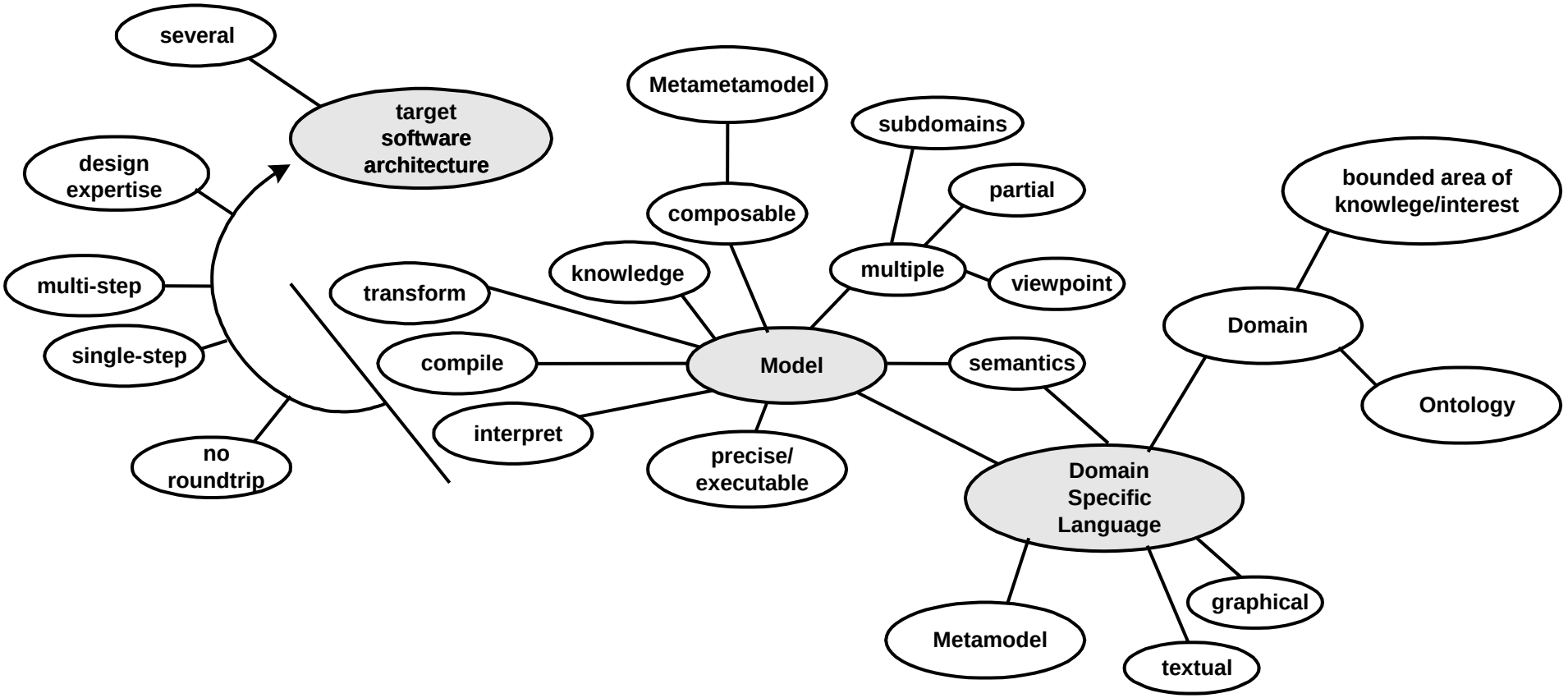
Bernd Kolb
bernd@kolbware.de
www.kolbware.de

Model Driven Development

- Model Driven Development is about making software development more **domain-related** as opposed to **computing related**. It is also about making software development in a certain domain **more efficient**.

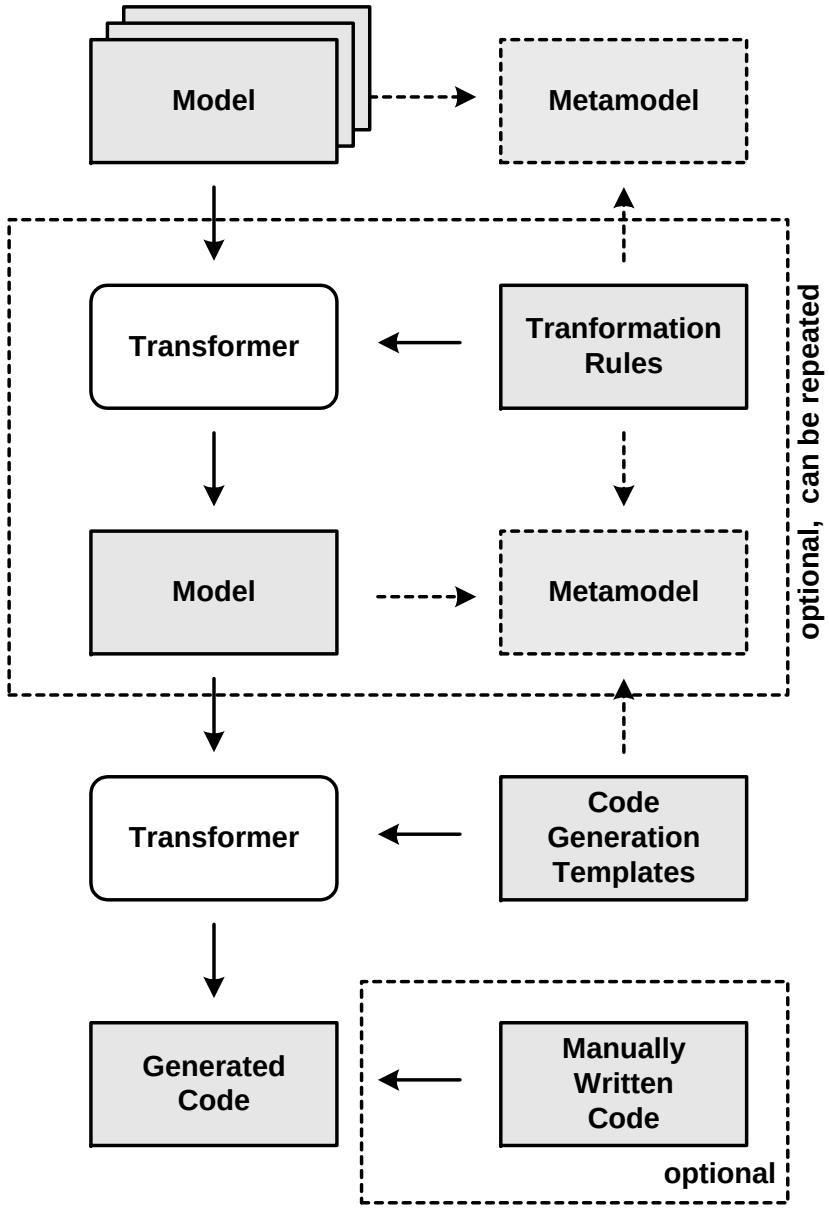


MDSD Core Concepts



How does MDSD work?

- Developer develops **model(s)** based on certain metamodel(s).
- Using **code generation templates**, the model is transformed to executable code.
- Optionally, the **generated code is merged** with manually written code.
- One or more **model-to-model transformation steps** may precede code generation.



Goals & Challenges

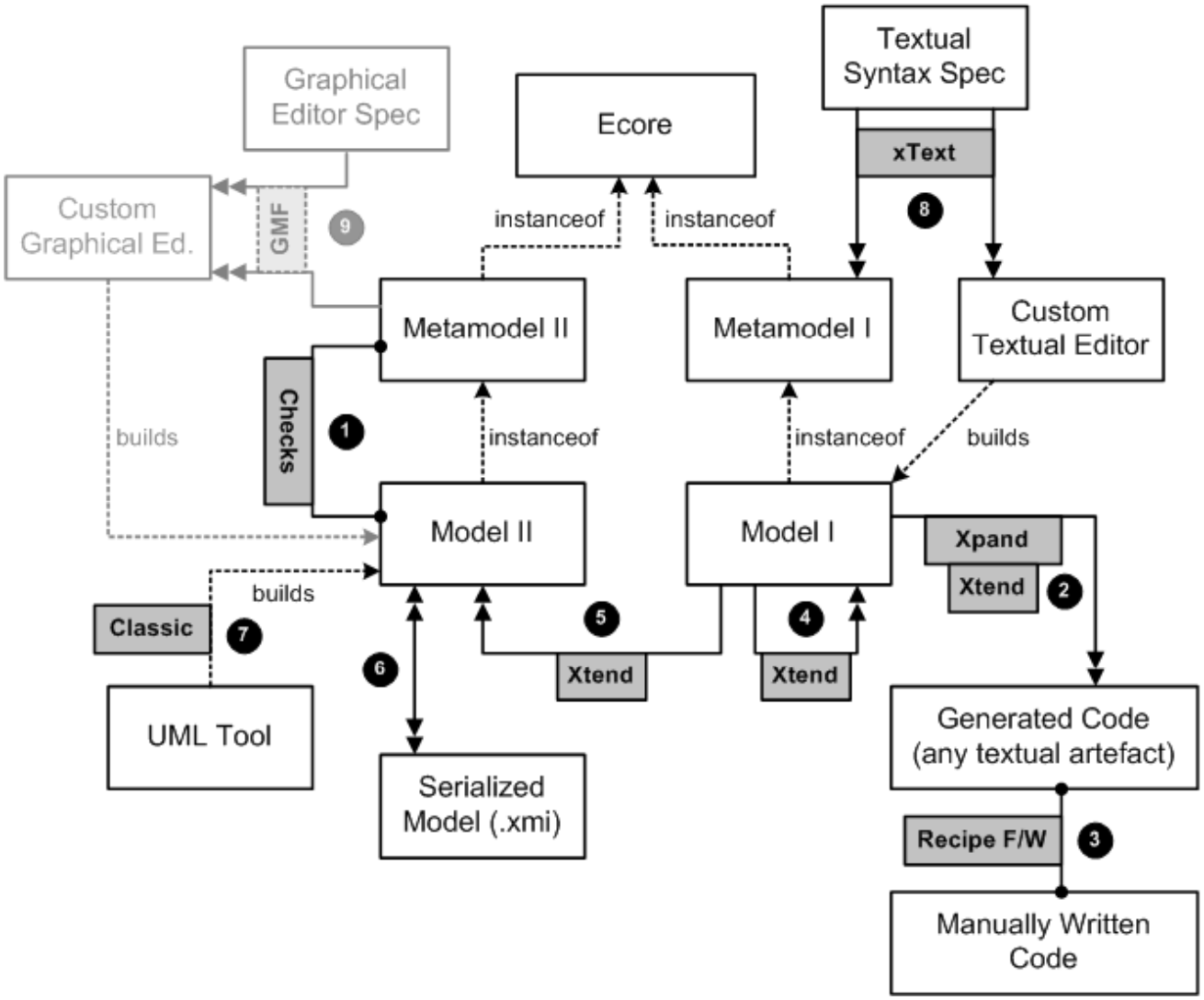
- **Goals:**
  - We need an **end-to-end tool chain** that allows us to build models, verify them and generate various artifacts from them.
  - All of this should happen in a homogeneous environment, namely Eclipse.

- **Challenges:**
  - **Good Editors** for your models
  - **Verifying** the models as you build them
  - **Transforming / Modifying** models
  - **Generating** Code
  - **Integrating** generated and non-generated code

openArchitectureWare

- Open Source
- Version 4.1.2 is current (and 4.2 in the making)
- Proven track record in various domains & project contexts
  - e.g., telcos, internet, enterprise, embedded realtime, finance, ...
- www.openarchitectureware.org
- IDE-portions based on Eclipse
- (Optional) Integration with Eclipse Modelling facilities (such as EMF)

openArchitectureWare / Eclipse EMF, GMF, EMP

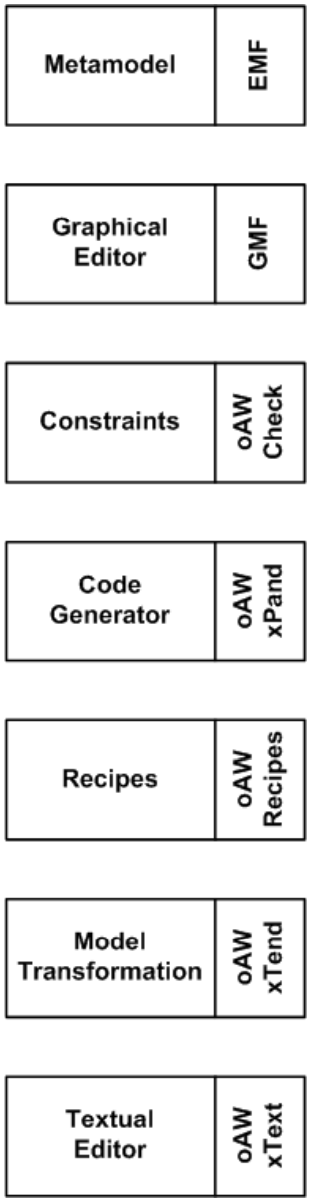


1. Model Verification using Constraints
2. Integrating Generating Code with Manually Written Code
3. Code Generation
4. Model Modification/Completion
5. Model-to-Model Transformation
6. Loading/Storing Models
7. Model Editing based on Custom-built Textual Editors
8. Model Editing using UML Tools
9. Model Editing based on Custom-built Graphical Editors (based on Eclipse GMF, not on openArchitectureWare)

© 2006-7 Völter, Efftinge, Kolb

Roadmap

- We will start by defining a **metamodel** for state machines, based on the UML metamodel
- We will then build a **graphical editor** for state machines using the well-known UML-based notation
- We will then add additional **constraints** (e.g. That states must have different names)
- Next up will be a **code generator** that creates a switch-based implementation of state machines in Java.
- **Recipes** help developers with the implementation of the actions associated with states.
- We will then cover **model-to-model transformations** and **model modifications**.
- Finally, we will build a **textual editor** for rendering the state machines textually.

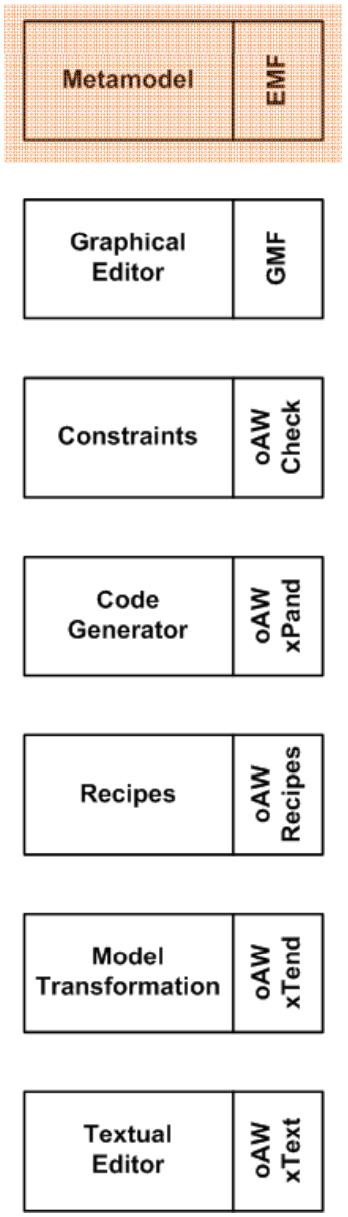


© 2006-7 Völter, Efftinge, Kolb

Defining the Metamodel

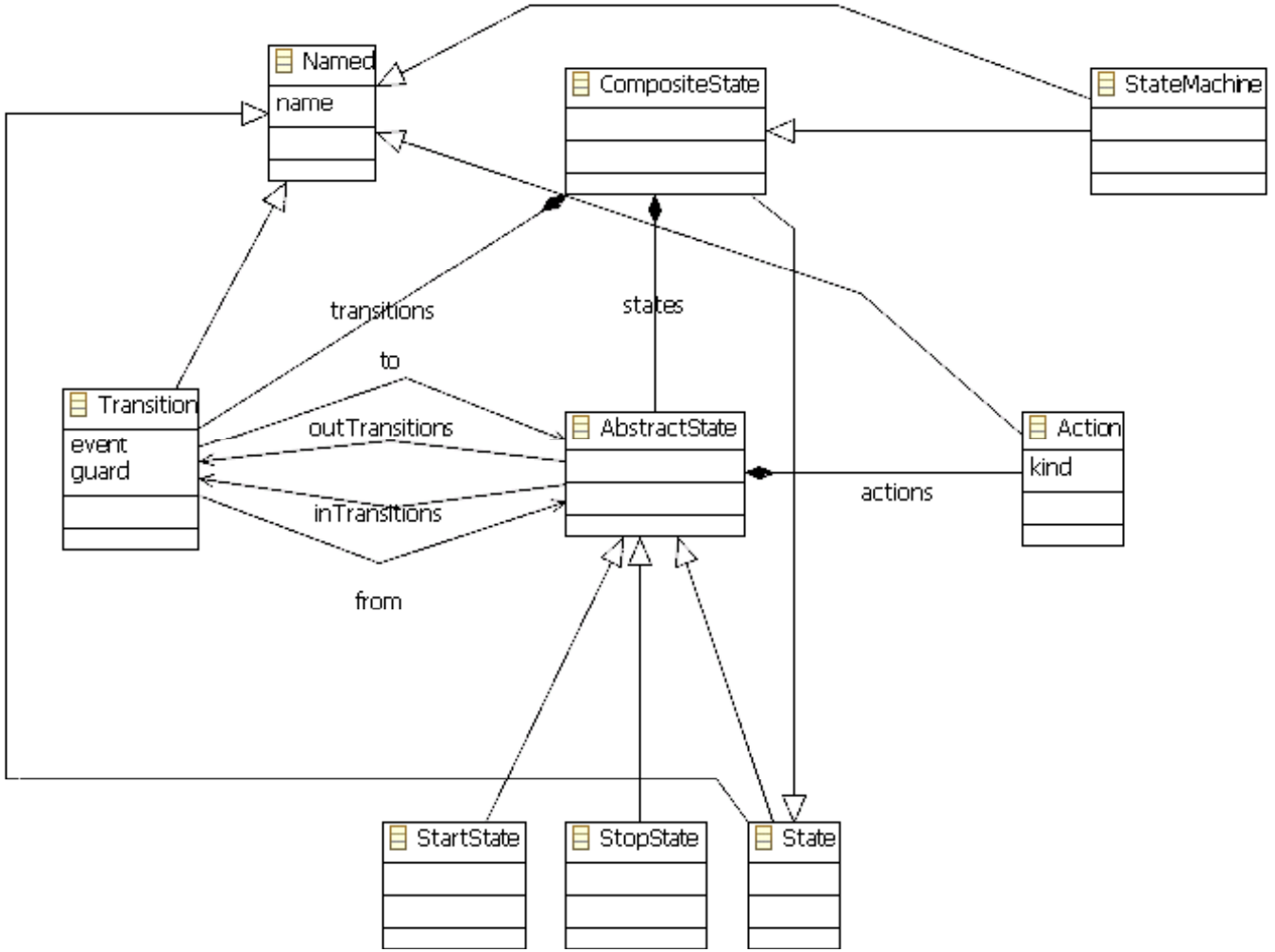
- A **state machine** consists of a number of **states**.
- States can be start states, stop states and “normal” states.
- A **transition** connects two states. States know their outgoing and incoming transitions.
- We also support **composite states** that themselves contain sub state machines.
- A state machine is itself a composite state.
- A state has **actions**. Actions can either be entry or exit actions.

- The metamodel is defined using EMF, the **Eclipse Modeling Framework**.



© 2006-7 Völter, Efftinge, Kolb

Defining the Metamodel II



Metamodel | EMF

Graphical Editor | GMF

Constraints | oAW Check

Code Generator | oAW xPand

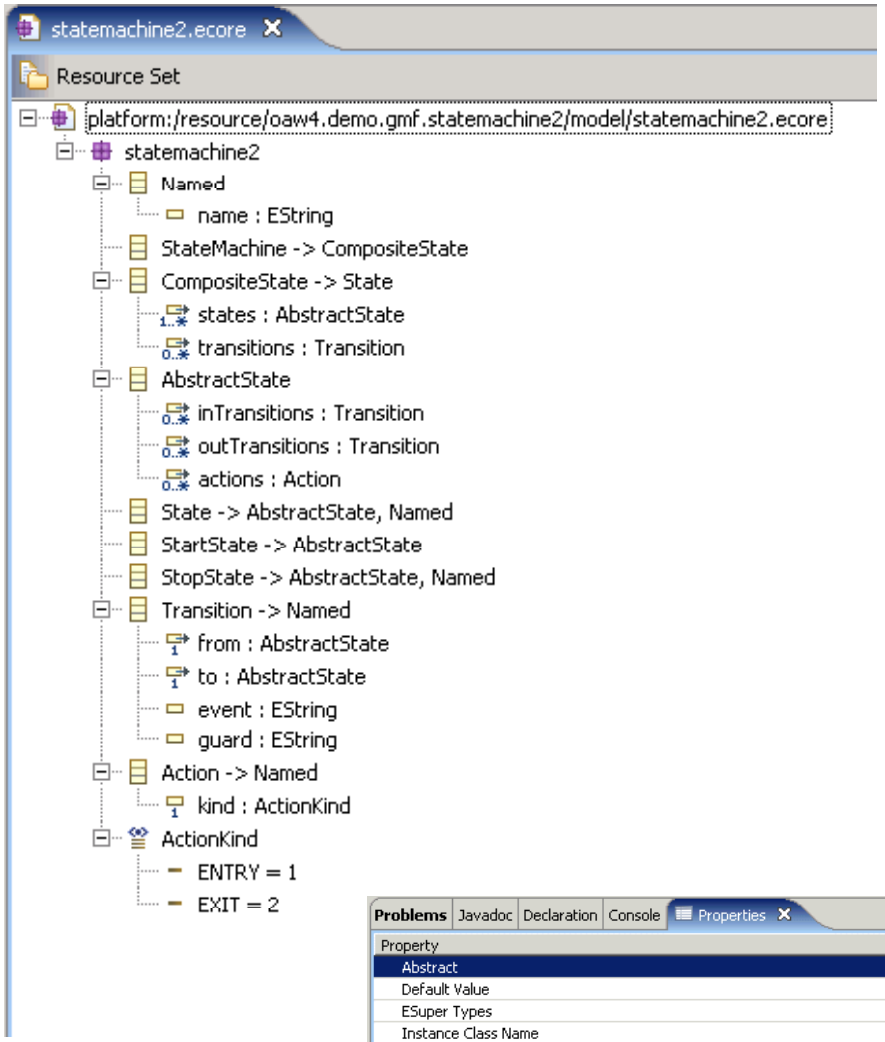
Recipes | oAW Recipes

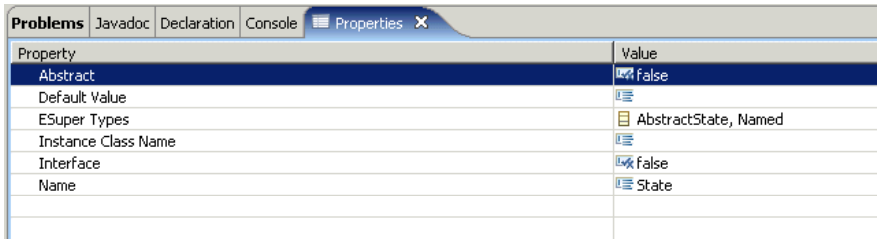
Model Transformation | oAW xTend

Textual Editor | oAW xText

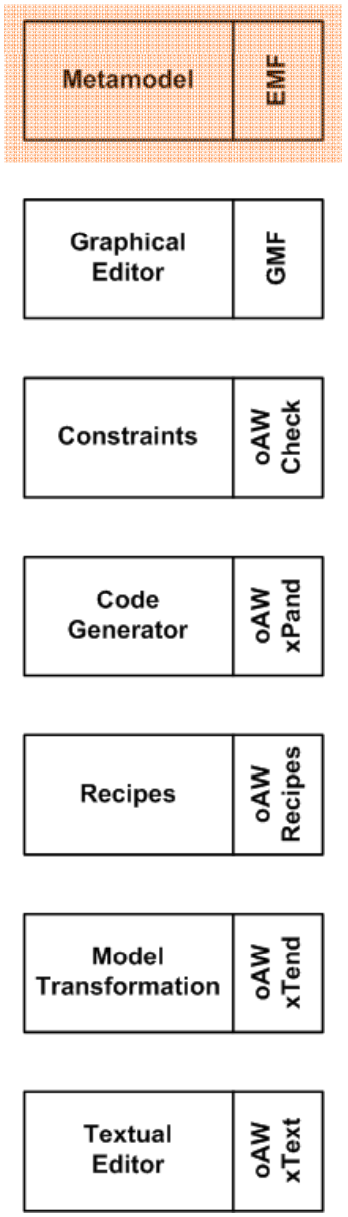
© 2006-7 Völter, Efftinge, Kolb

Defining the Metamodel III



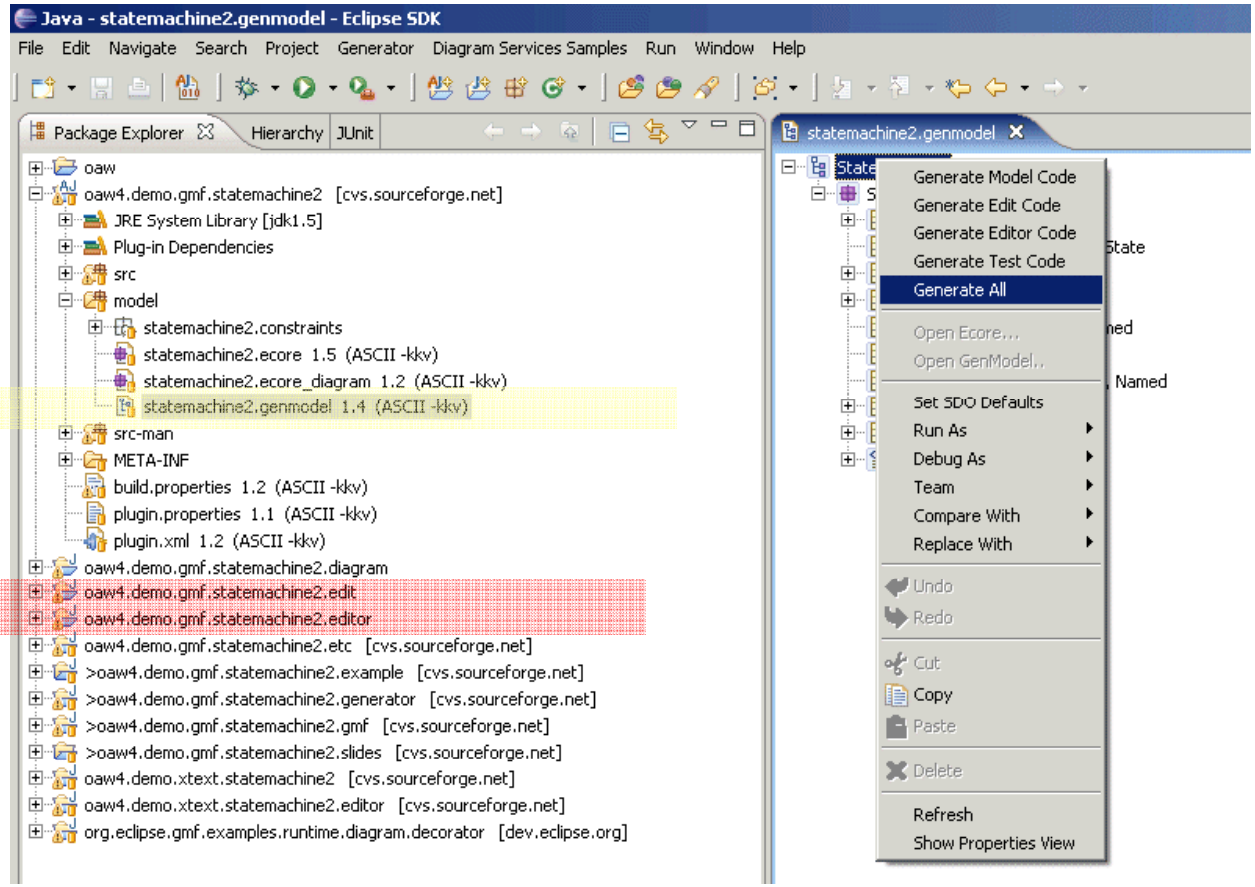


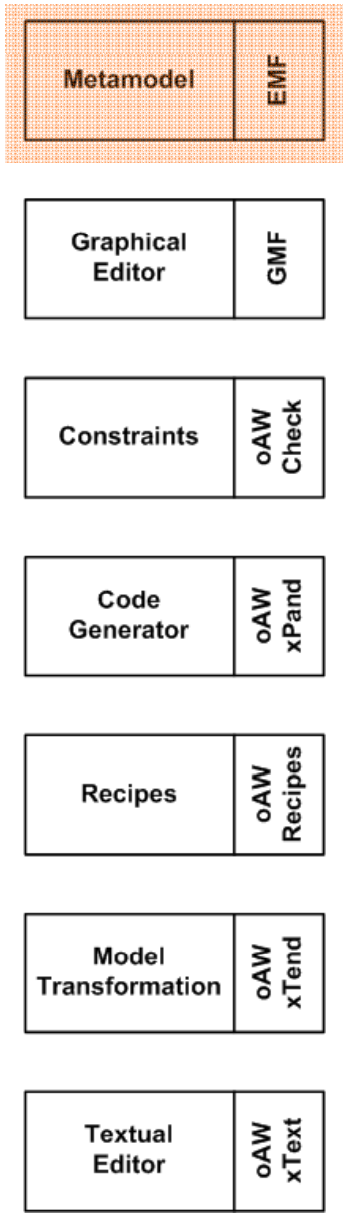
- The metamodel is defined **using EMF**.
- EMF provides **tree-based editors** to define the metamodel.
- The metamodel has its own project called *oaw4.demo.gmf.statemachine2*



© 2006-7 Völter, Efftinge, Kolb

Defining the Metamodel IV

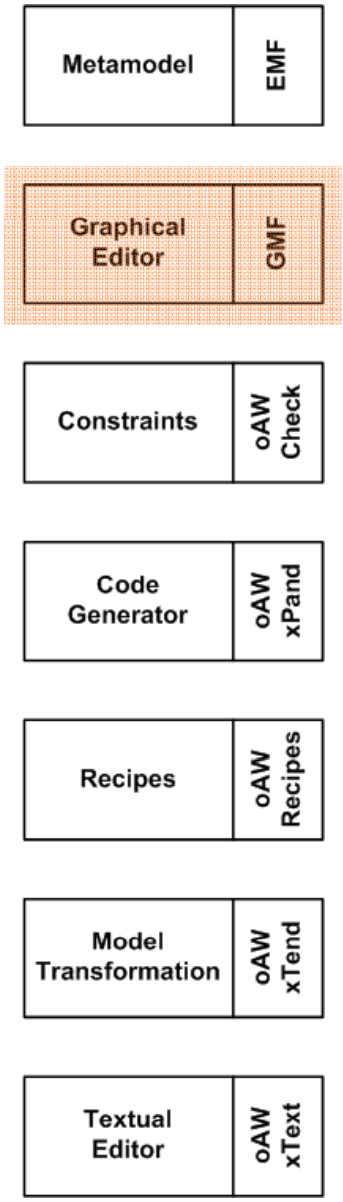




- Note that we have to create the **genmodel** as well as the **.edit** and **.editor** projects from the ecore model.
- This is necessary for the graphical editor to work.

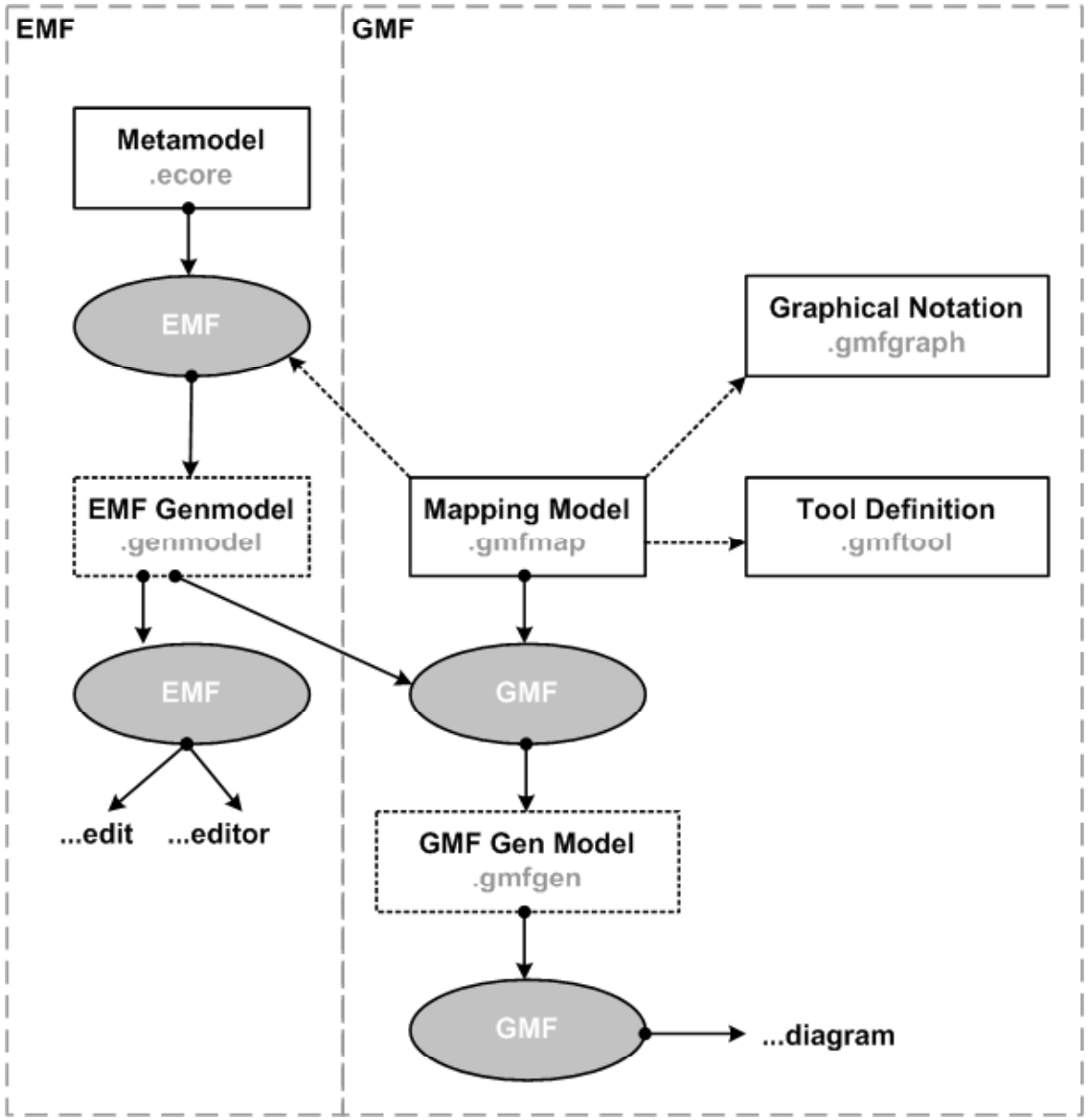
Building the graphical Editor

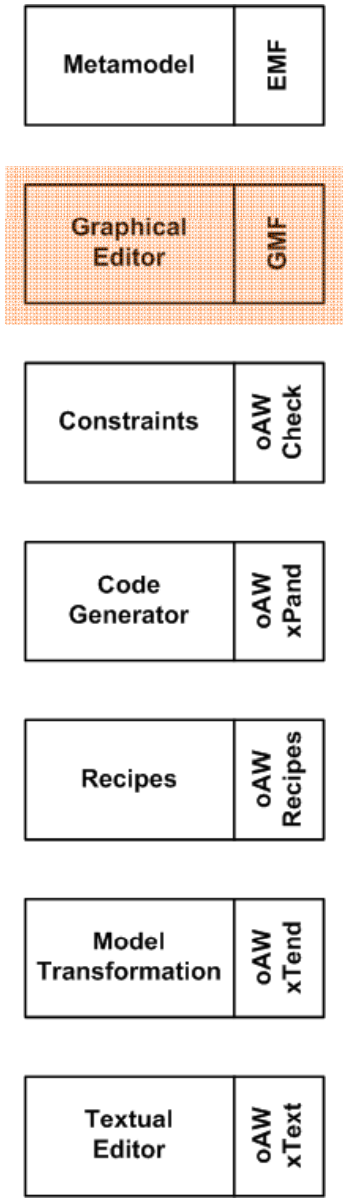
- The editor is **based on the metamodel** defined before.
- A number of additional models has to be defined:
  - A model defining the **graphical notation**
  - A model for the editor's **palette** and other tooling
  - A **mapping model** that binds these two models to the domain metamodel
- A **generator** generates the concrete editor based on these models.
- The editor is built with the Eclipse GMF, the **Graphical Modelling Framework**.



© 2006-7 Völter, Efftinge, Kolb

Building the graphical Editor II

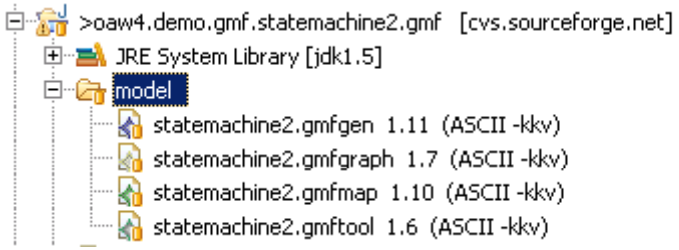


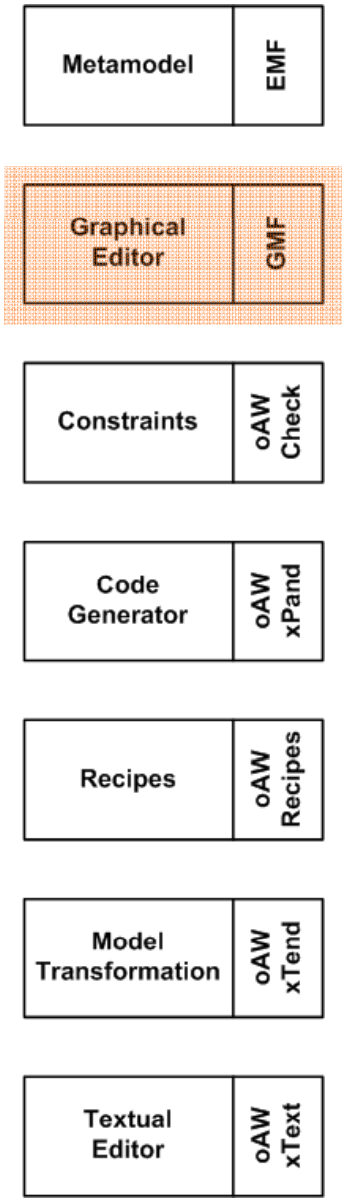


© 2006-7 Völter, Efftinge, Kolb

Building the graphical Editor III

- We use **another project** for the GMF models from which we'll create the editor:
  oaw4.demo.gmf.statemachine2.gmf
- This project contains **all the additional models** we talked about before:

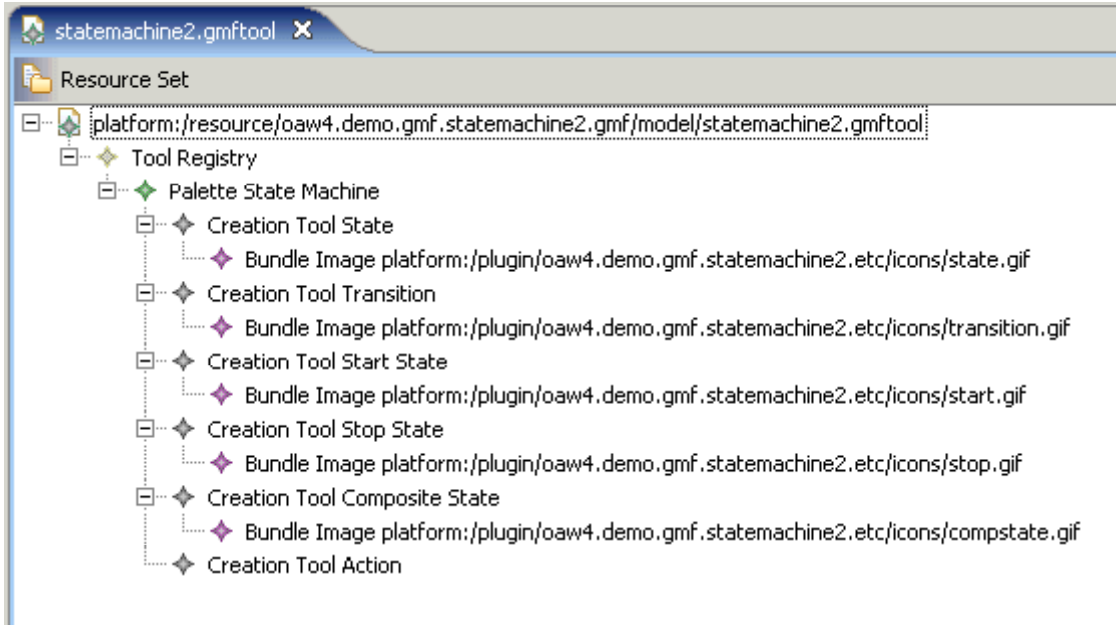




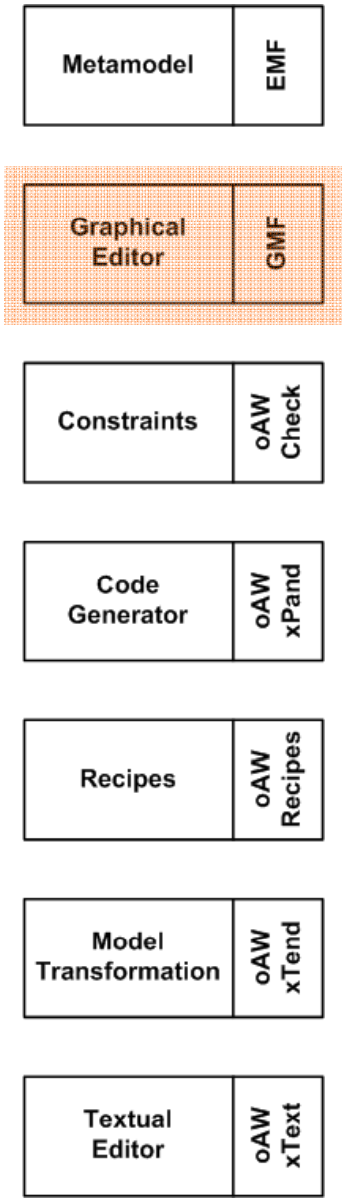
© 2006-7 Völter, Efftinge, Kolb

Building the graphical Editor IV

- The gmftool model contains the **definition of the palette** that will be used in the editor.

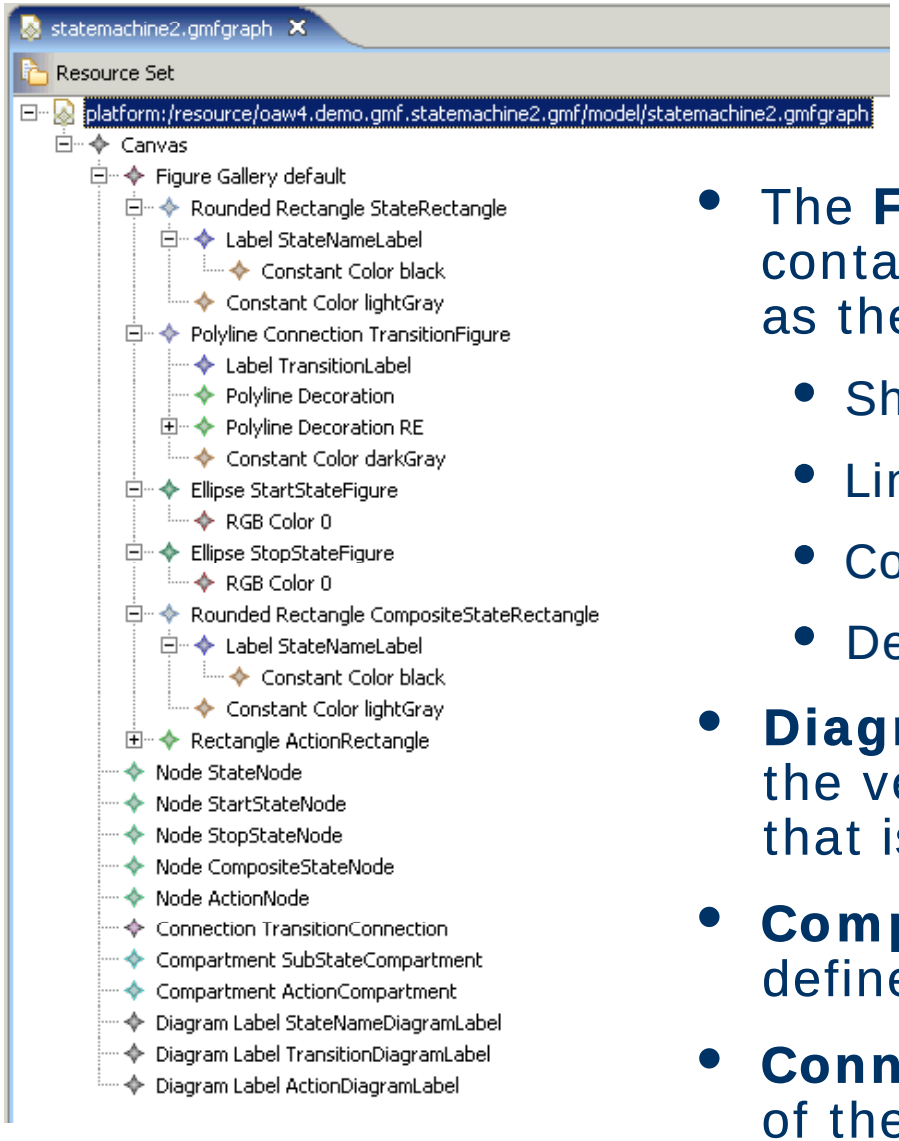


- We have **creation tools** for all the relevant metamodel elements.
- Each of these tools has a **nice icon** associated.

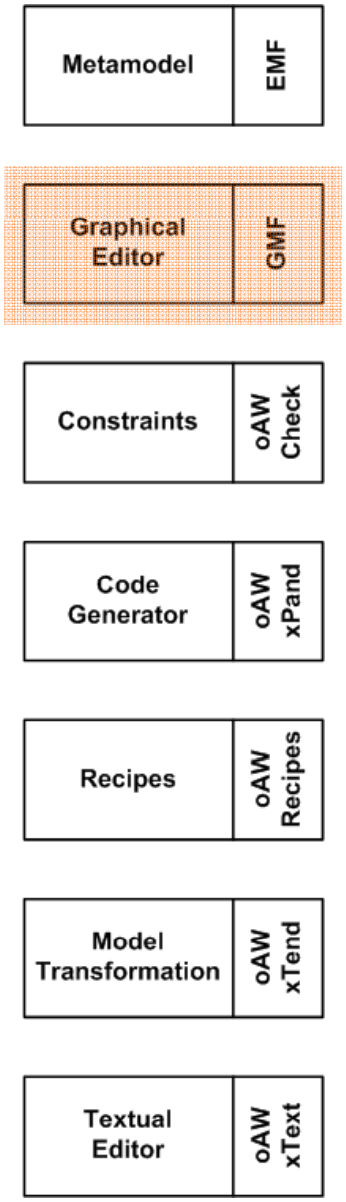


© 2006-7 Völter, Efftinge, Kolb

Building the graphical Editor V

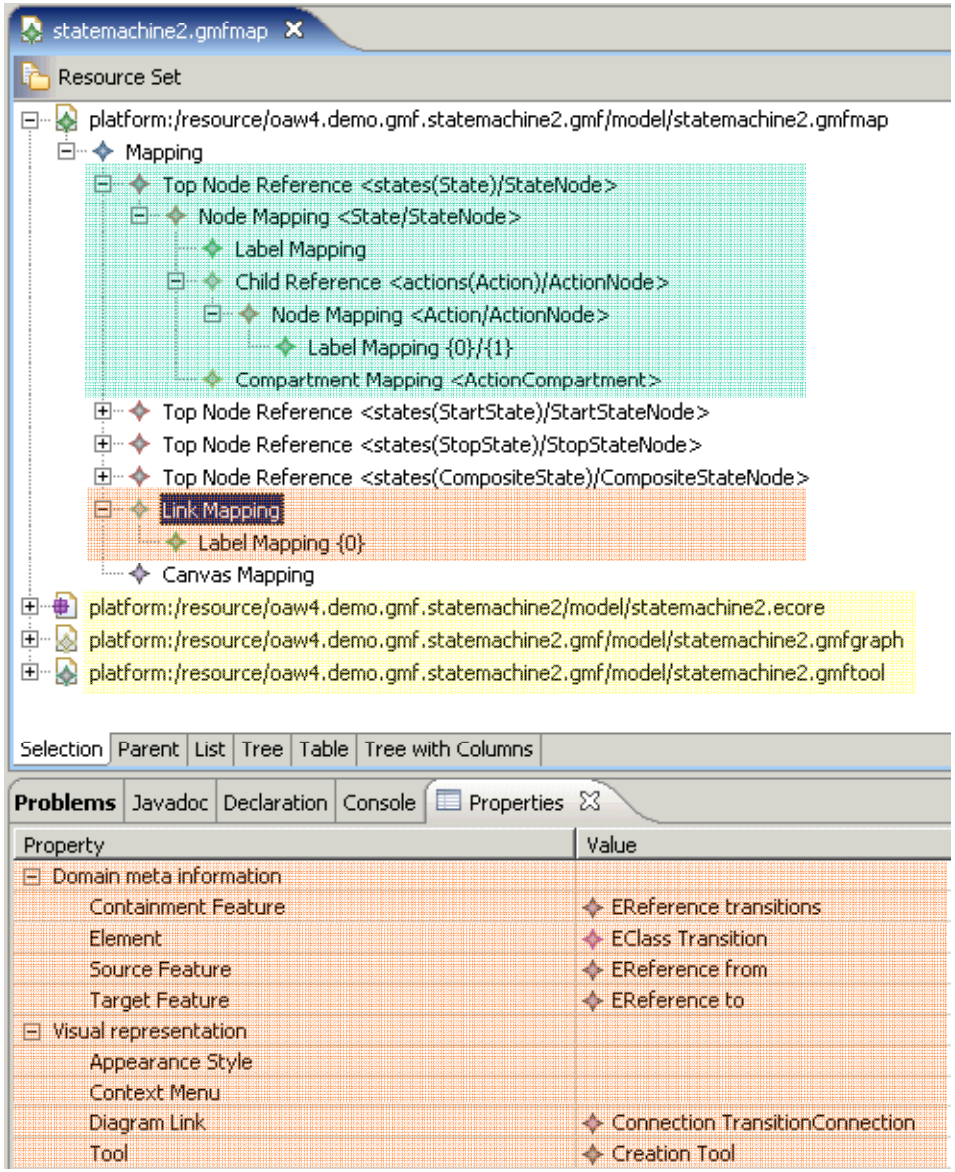


- The **Figure Gallery** contains the figures (as well as their associated labels)
  - Shapes
  - Line Style
  - Colors
  - Decorations
- **Diagram Nodes** represent the vertices in the graph that is being edited.
- **Compartments** can be defined as parts of Nodes.
- **Connections** play the role of the edges in the graph.

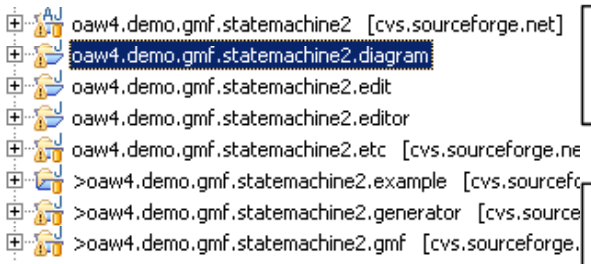


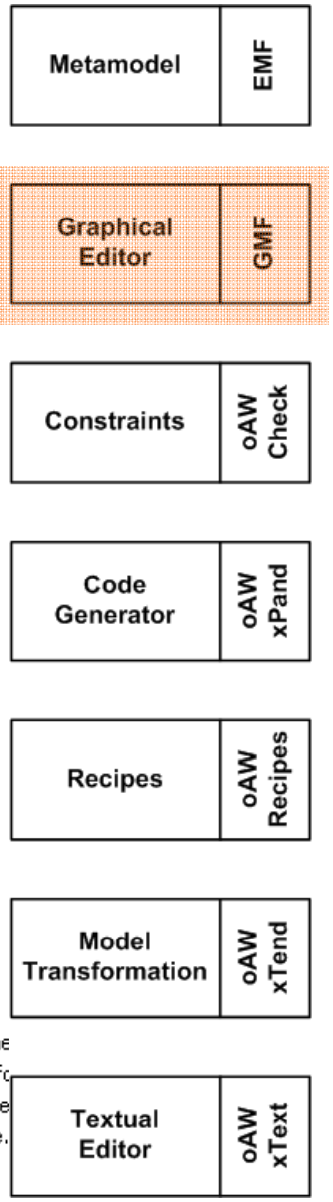
© 2006-7 Völter, Efftinge, Kolb

Building the graphical Editor VI



- We map **nodes** and **links**.
- We **include all the other models** so they can be referenced.
- **Better editors** became available with GMF final.
- From that, we generate the editor plugins:

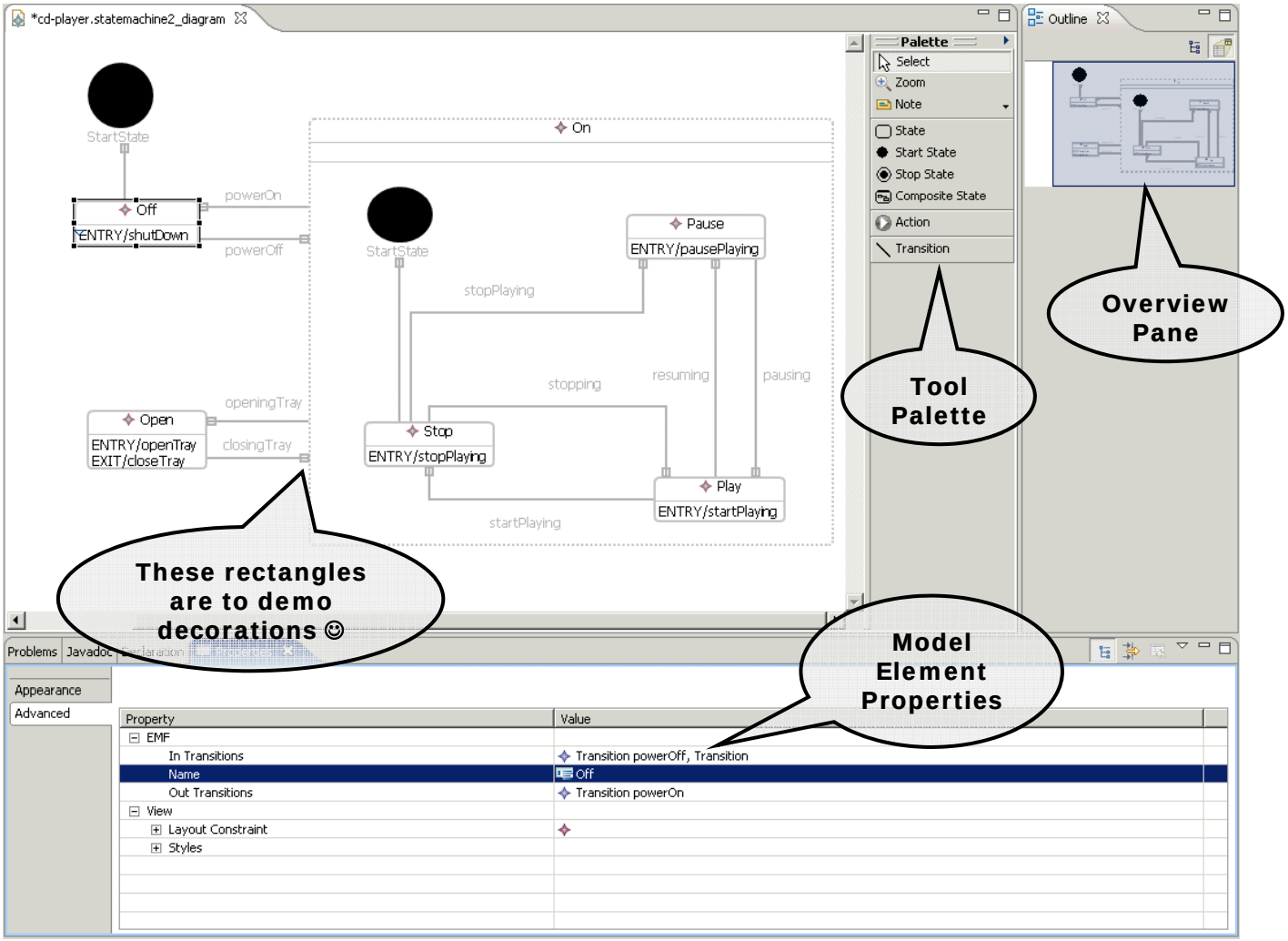




© 2006-7 Völter, Efftinge, Kolb

Building the graphical Editor VII

- Here is the **editor**, started in the runtime workbench, with our CD Player example.



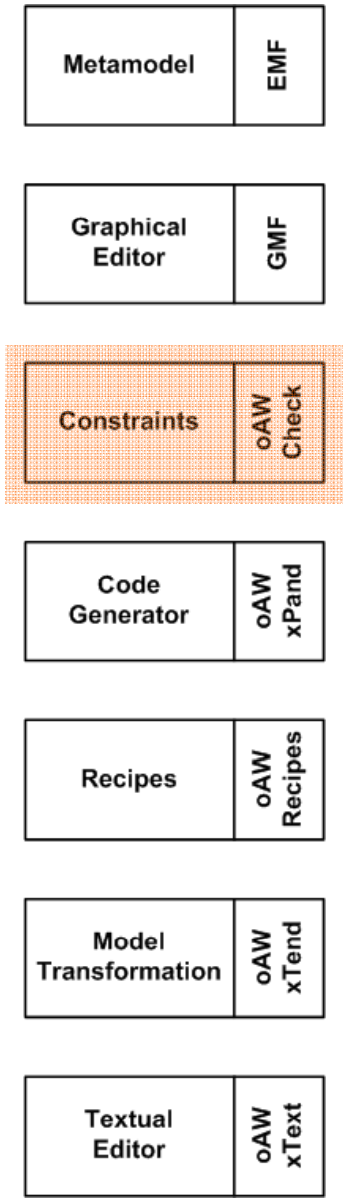
Metamodel	EMF
Graphical Editor	GMF
Constraints	oAW Check
Code Generator	oAW xPand
Recipes	oAW Recipes
Model Transformation	oAW xTend
Textual Editor	oAW xText

Constraints

- Constraints are **rules that models must conform to** in order to be valid. These are in addition to the structures that the metamodel defines.
- Formally, constraints are part of the metamodel.
- A constraint is a **boolean expression (a.k.a. predicate)** that must be true for a model to conform to a metamodel.
- Constraint Evaluation should be available
  - in **batch mode** (when processing the model)
  - as well as **interactively**, during the modelling phase in the editor

  … and **we don't want to implement constraints twice** to have them available in both places!

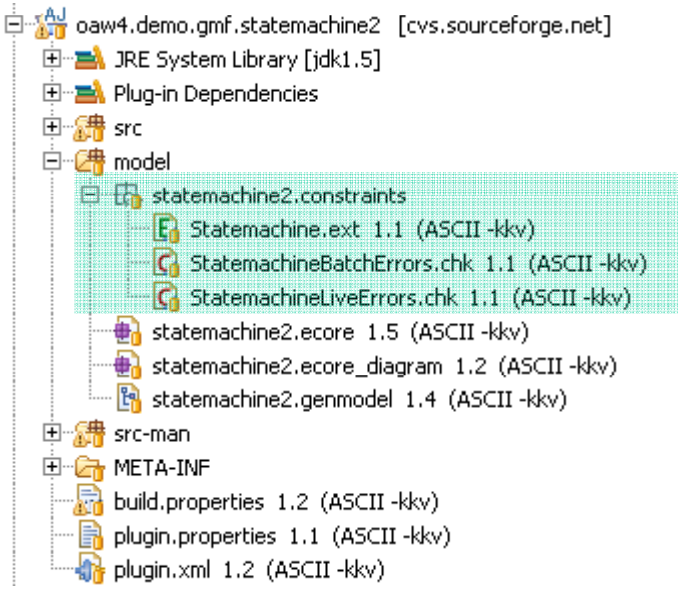
- **Functional languages** are often used here.
  - UML's OCL (Object Constraint Language) is a good example,
  - We use **oAW's check language**, which is based on OCL

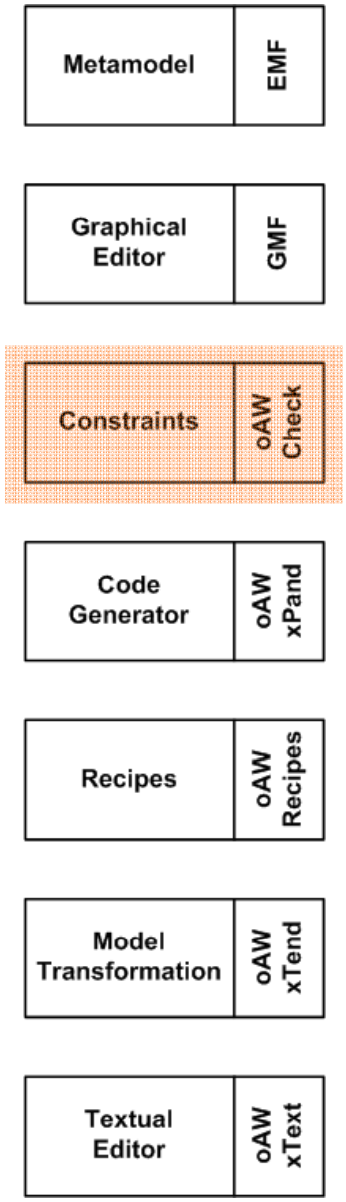


© 2006-7 Völter, Efftinge, Kolb

Constraints II

- Constraints are put into the **statemachine2** project, the same as the metamodel.
- **StatemachineBatchErrors** are used in batch validation mode (automatically evaluated every 2 seconds in the editor)
- **StatemachineLiveErrors** prevent erratic modellings in the first place.

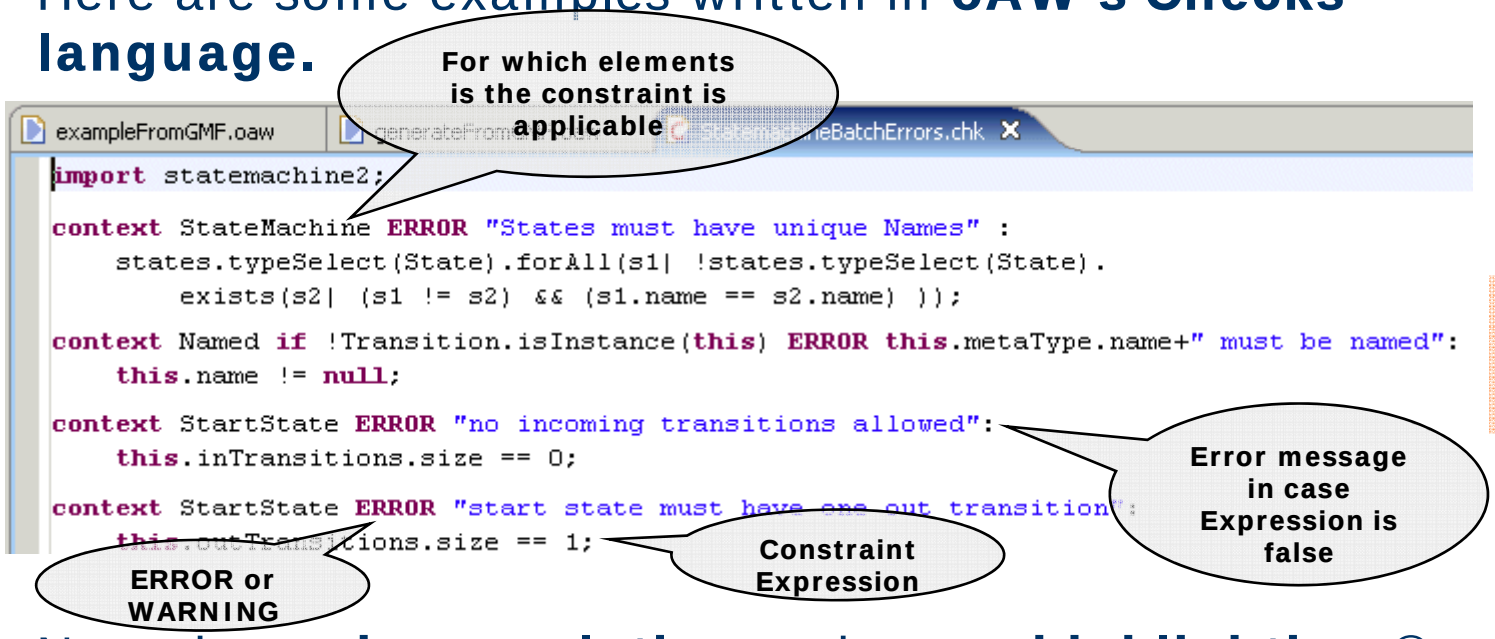




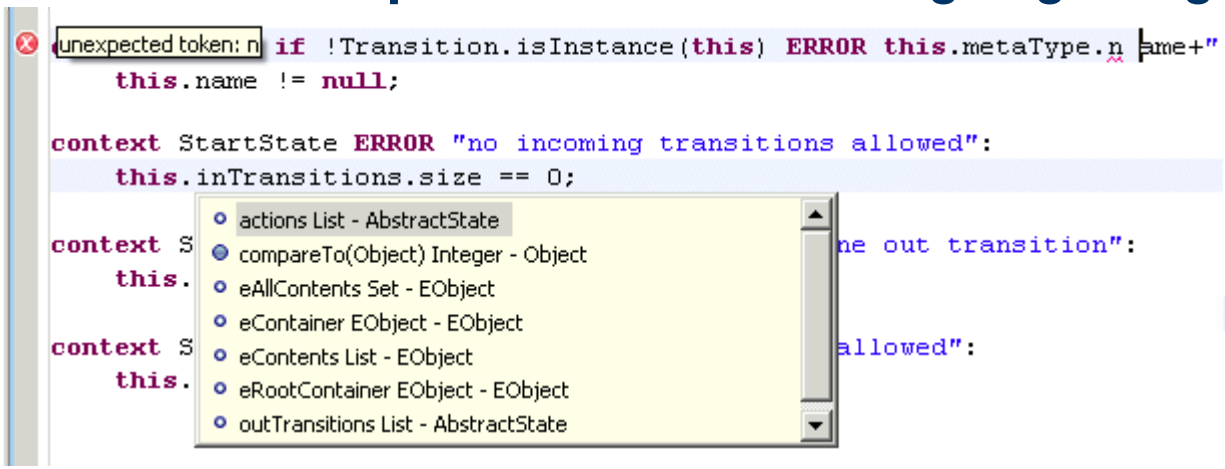
© 2006-7 Völter, Efftinge, Kolb

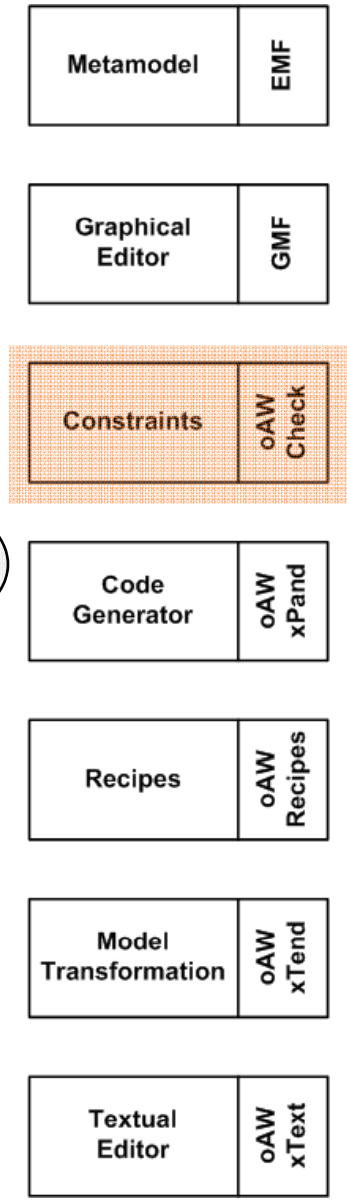
Constraints III

- Here are some examples written in **oAW's Checks language.**



- Note the **code completion** and **error highlighting** ☺

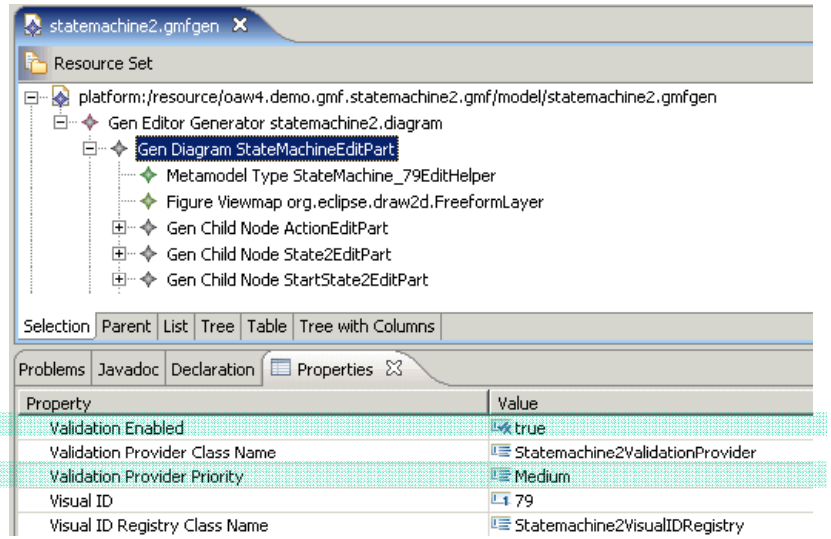


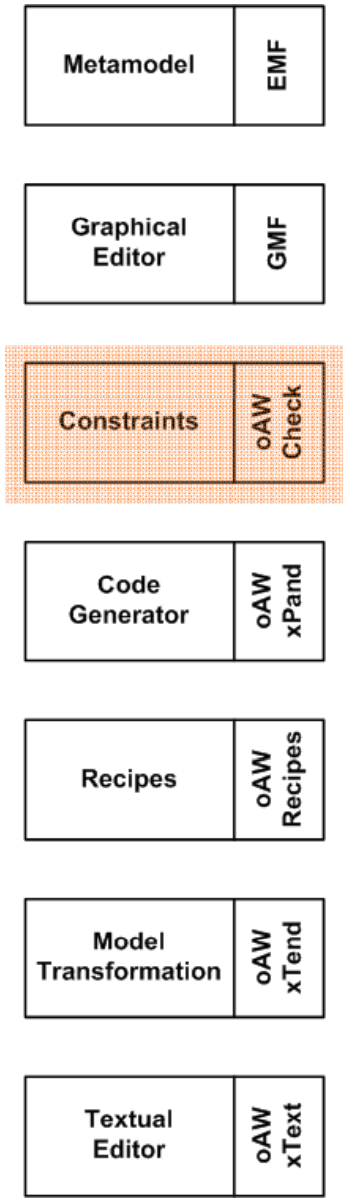


© 2006-7 Völter, Efftinge, Kolb

Constraints IV

- To make the GMF generated editors evaluate our constraints, we needed to **tweak things a little bit;** most of this is in *oaw4.demo.gmf.statemachine2.etc*
  - We wrote our own **ConstraintEvaluators** and plugged in the oAW CheckFacade.
  - We used **AspectJ** to weave in Adapters into the EMF Factory
  - We wrote a **watchdog** that does the batch evaluations whenever the model does not change for two seconds.
- Also, you have to make two important adjustments in the **gmfgen** model

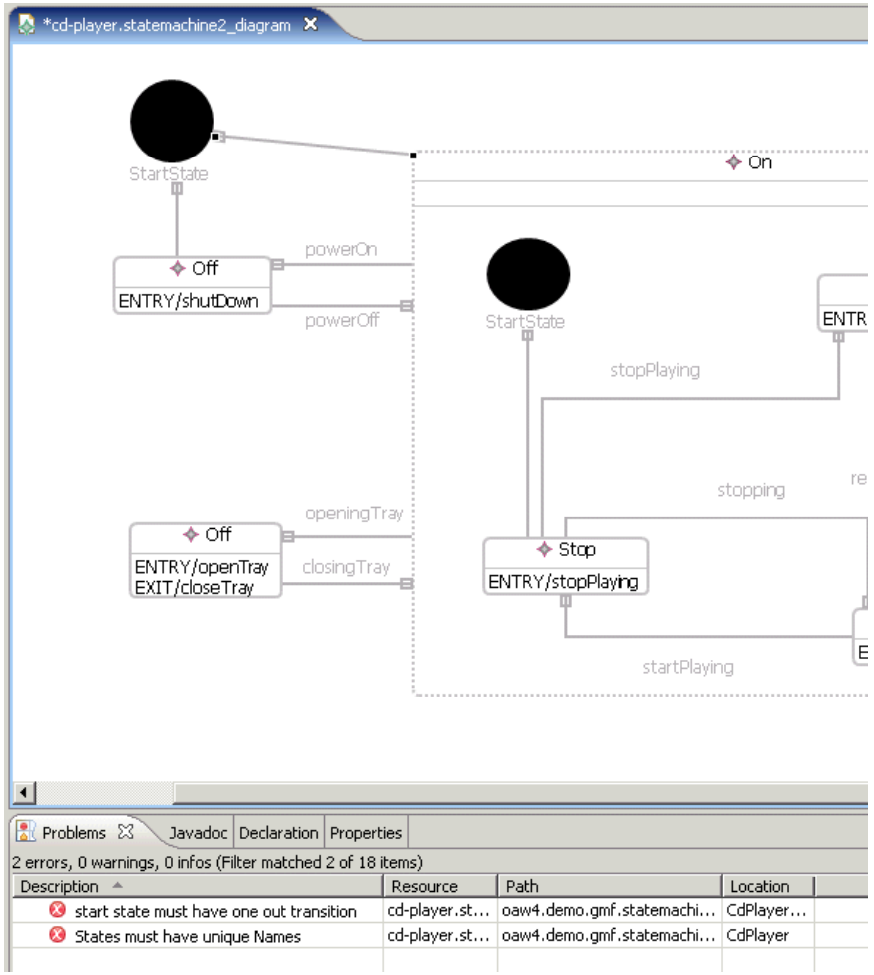


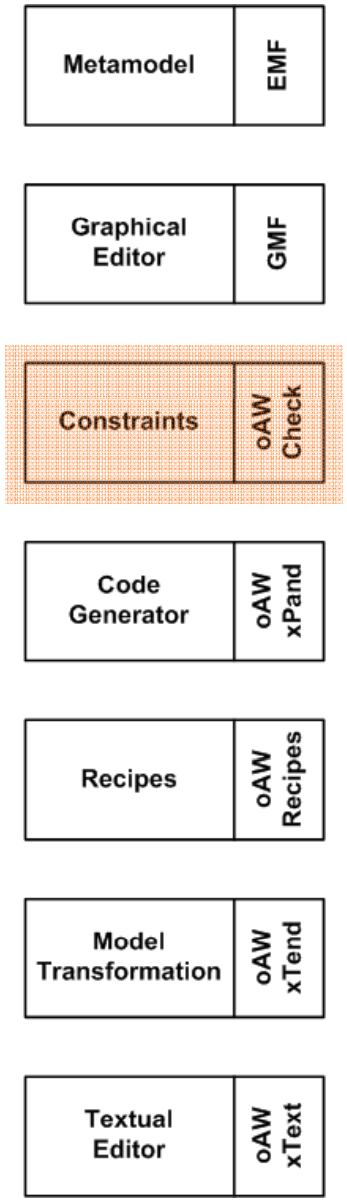


© 2006-7 Völter, Efftinge, Kolb

Constraints V

- In this model there are **two errors**
  - There are two states with the same name (Off)
  - The start state has more than one out-Transition
- The validation is executed automatically
- Clicking the error message **selects** the respective "broken" **model element** in the diagram.

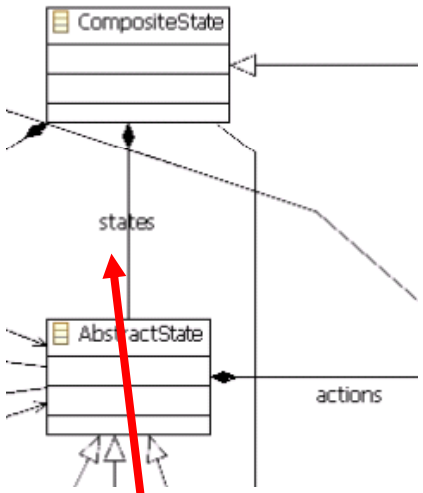




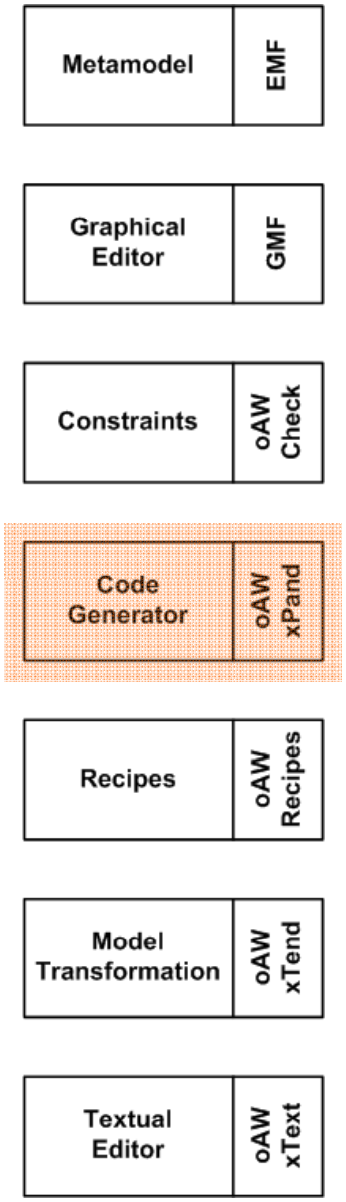
© 2006-7 Völter, Efftinge, Kolb

Code Generation

- Code Generation is used to **generate executable code** from models.
- Code Generation is **based on the metamodel** and uses **templates** to attach to-be-generated source code.
- In openArchitectureWare, we use a **template language** called **xPand**.
- It provides a number of **advanced features** such as polymorphism, AO support and a powerful integrated expression language.
- Templates can access **metamodel properties** seamlessly



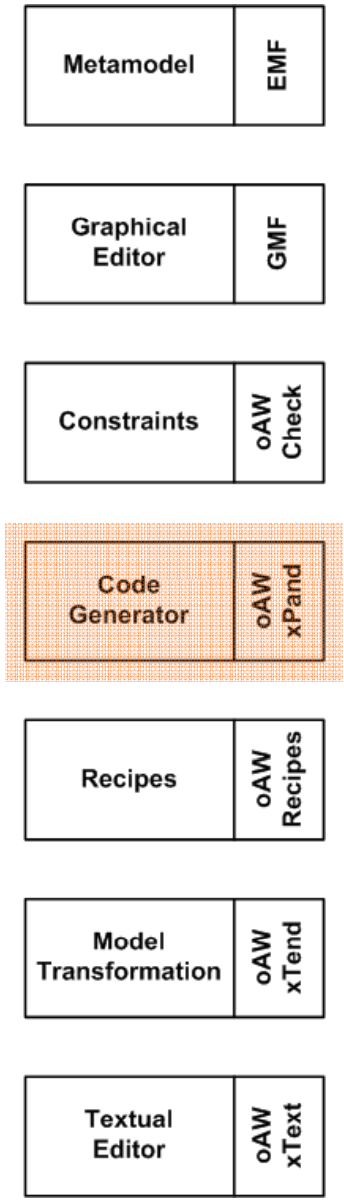

```
«DEFINE SwitchBasedImpl FOR StateMachine»

    «FOREACH states.typeSelect(State) AS s
        public static final int «s.constant
    «ENDFOREACH»
```



© 2006-7 Völter, Efftinge, Kolb

Code Generation II

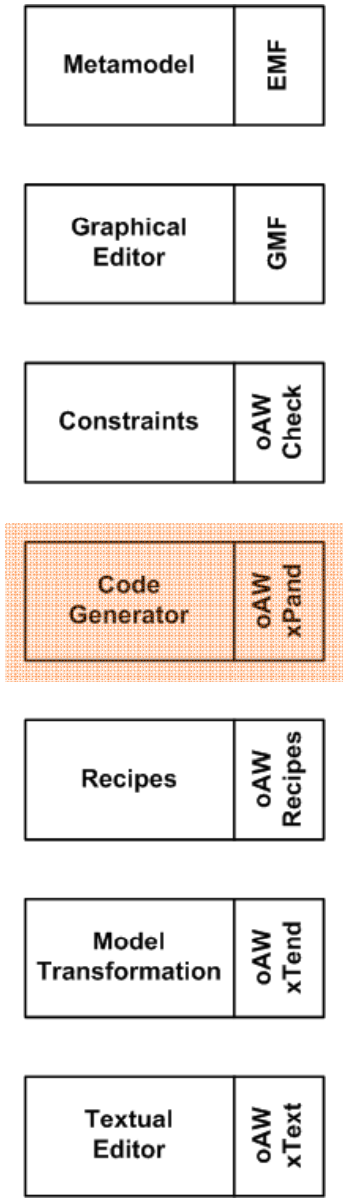
- What **kind of code** will be generated? How do you implement a state machine?
- There are **many ways** of implementing a state machine:
  - GoF's State pattern
  - If/Switch-based
  - Decision Tables
  - Pointers/Indexed Arrays
- We will use the switch-based alternative. It is neither the most efficient nor the most elegante alternative, **but it's simple**.
- For more discussion of this topic, see
  Practical State Charts in C/C++ by *Miro Samek*



© 2006-7 Völter, Efftinge, Kolb

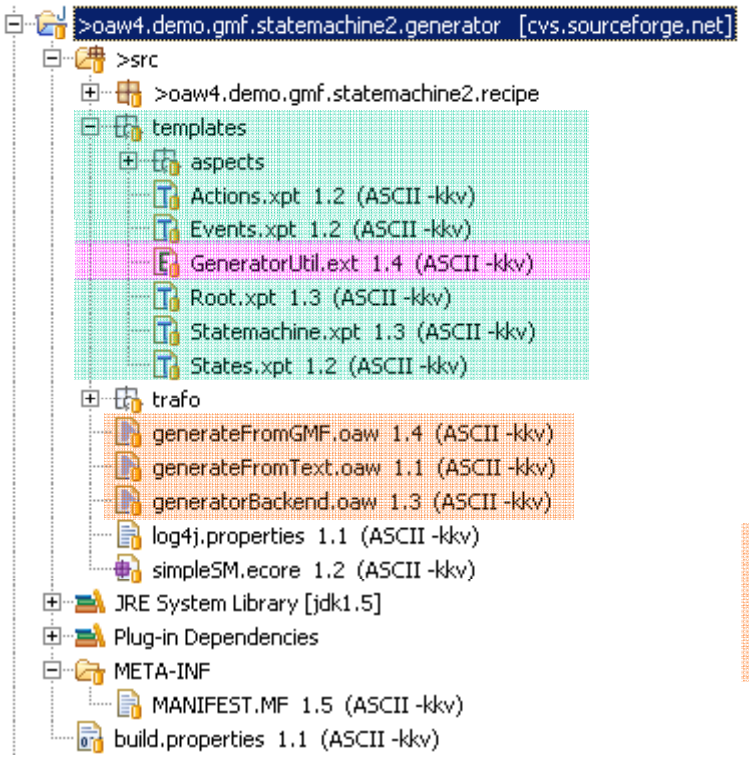
Code Generation III: Pseudocode

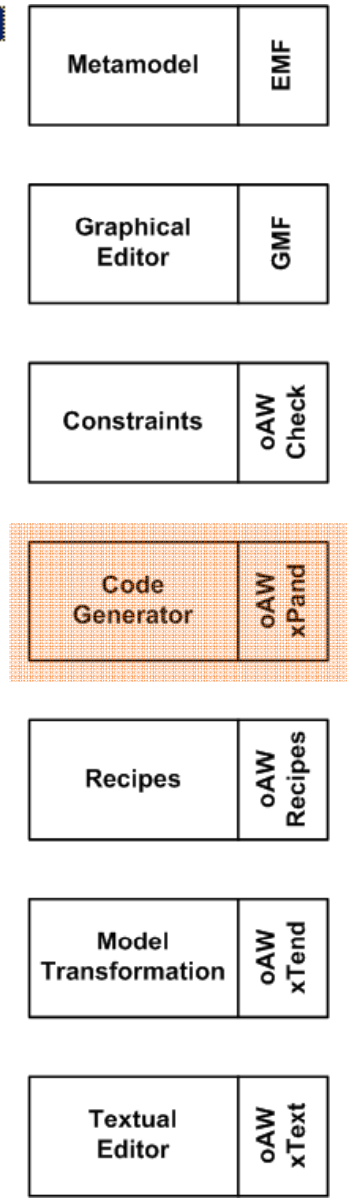
- Generate an **enumeration** for the states
- Generate an **enumeration** for the events
- Have a **variable** that remembers the state in which the state machine is currently in.
- Implement a function **trigger(event)** which
  - First **switches over all states** to find out the current state
  - Check whether there's a **transition for the event** passed into the function
  - If so,
    - execute **exit action** of current state,
    - Set **current state** to target of transition
    - Execute **entry action** of this new current state
    - Return
- And also handle **nested states** ☺



Code Generation IV

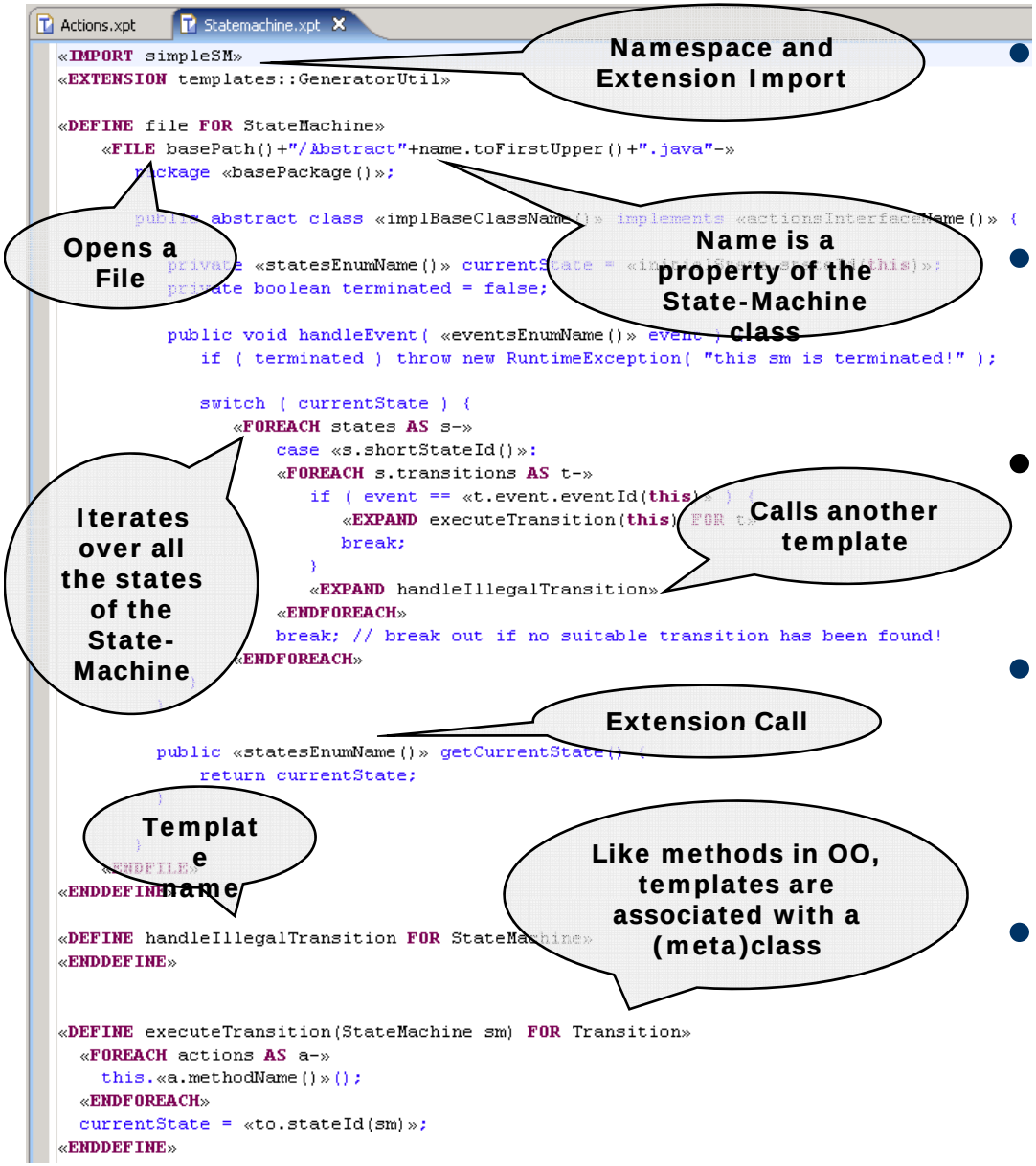
- The **generator** is located in the *oaw4.demo.gmf.statemachine2.generator* project.
- There are a number of **code generation templates.**
  - Extensions are also defined.
- There are also workflow files (.oaw) that control the **workflow** of a generator run.
- Different workflow files contain different “parts” of the overall generator run and **call each other**.
- Workflow files are in some small way like ant files.



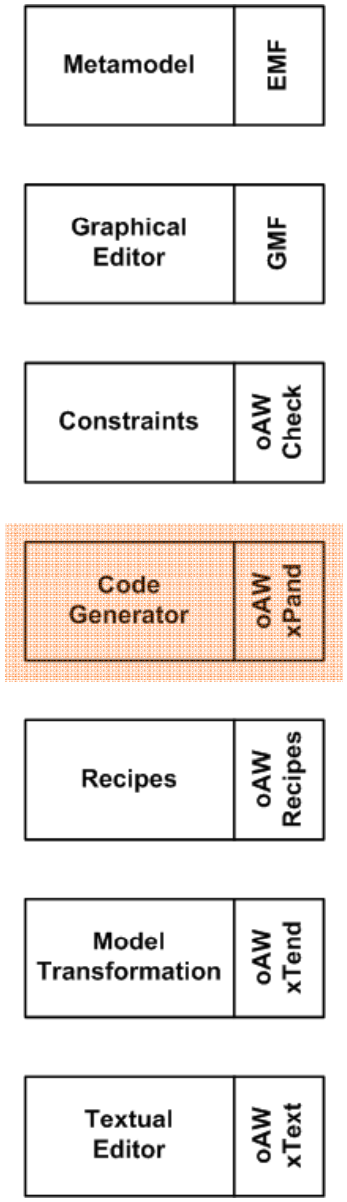


© 2006-7 Völter, Efftinge, Kolb

Code Generation V



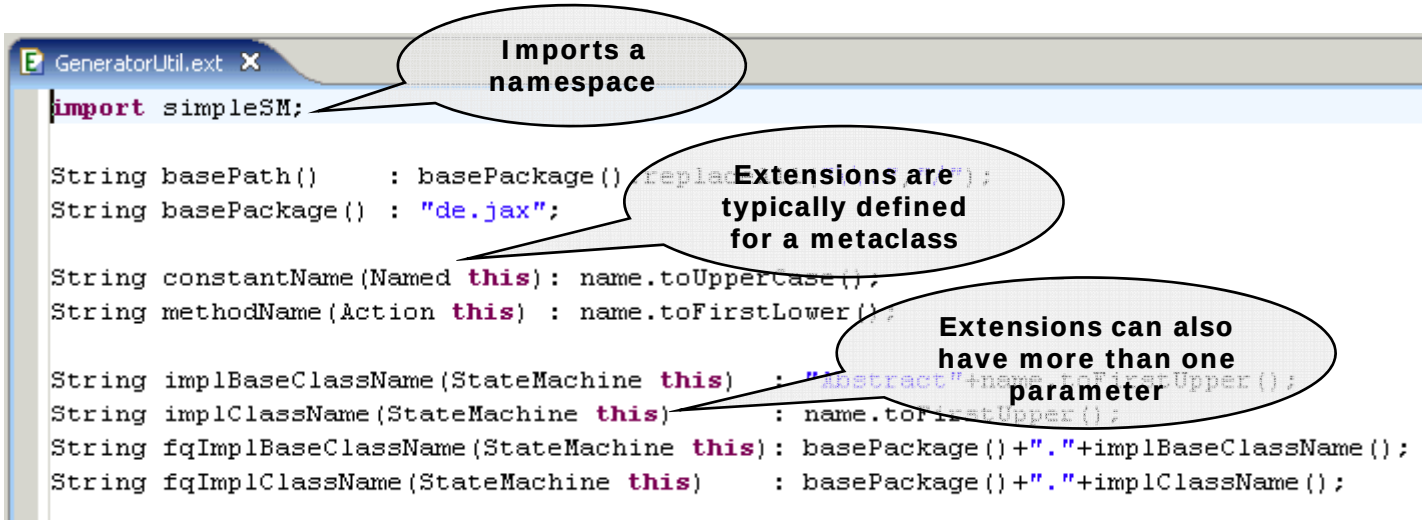
- The **blue text** is generated into the target file.
- The **capitalized words** are xPand keywords
- **Black text** are metamodel properties
- DEFINE...END-DEFINE blocks are called **templates**.
- The whole thing is called a **template file**.



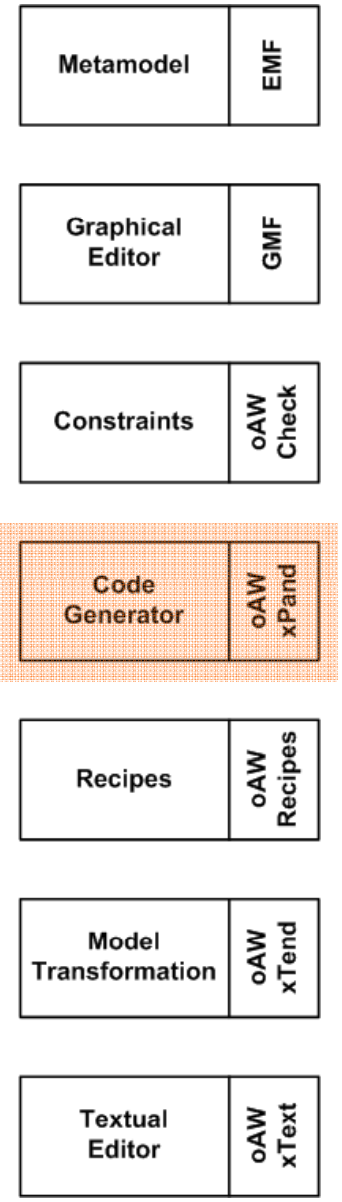
© 2006-7 Völter, Efftinge, Kolb

Code Generation VI

- One can **add behaviour to existing metaclasses** using oAW's **Xtend** language.



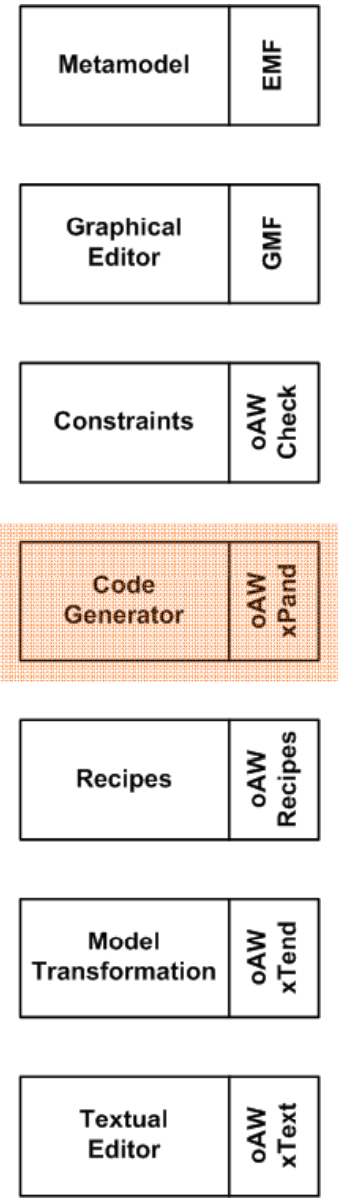
- Extensions can be called using **member-style syntax**: *myAction.methodName()*
- Extensions can be used in **Xpand templates, Check files** as well as in other **Extension files**.
- They are imported into template files using the **EXTENSION** keyword



© 2006-7 Völter, Efftinge, Kolb

Code Generation VII

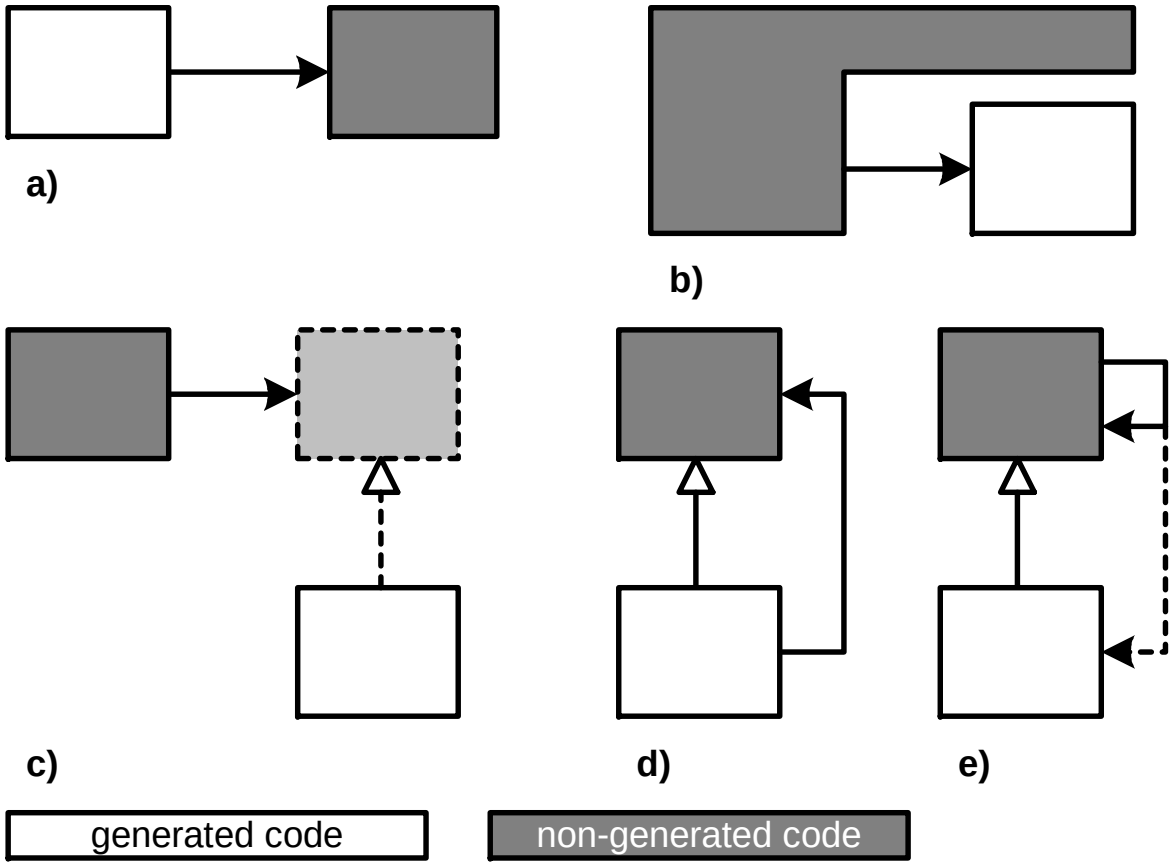
- Workflow **loads** the model, **checks** it (same constraints as in Editor!) and then **generates** code.

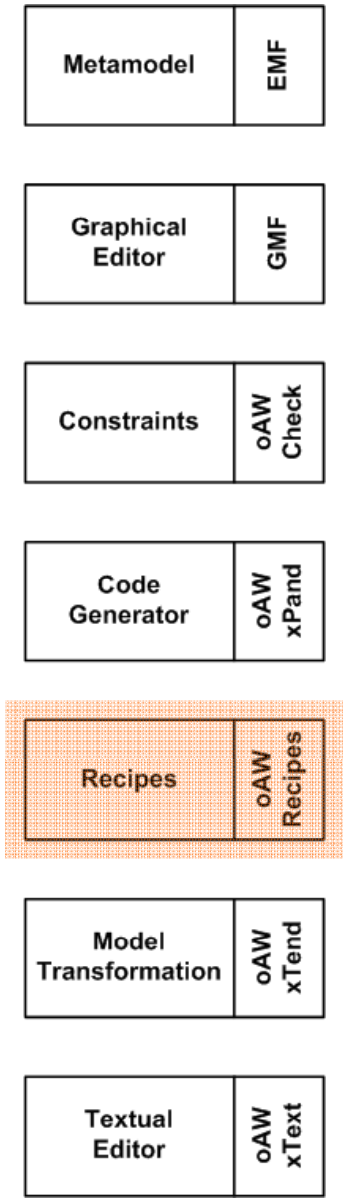


© 2006-7 Völter, Efftinge, Kolb

Recipes I

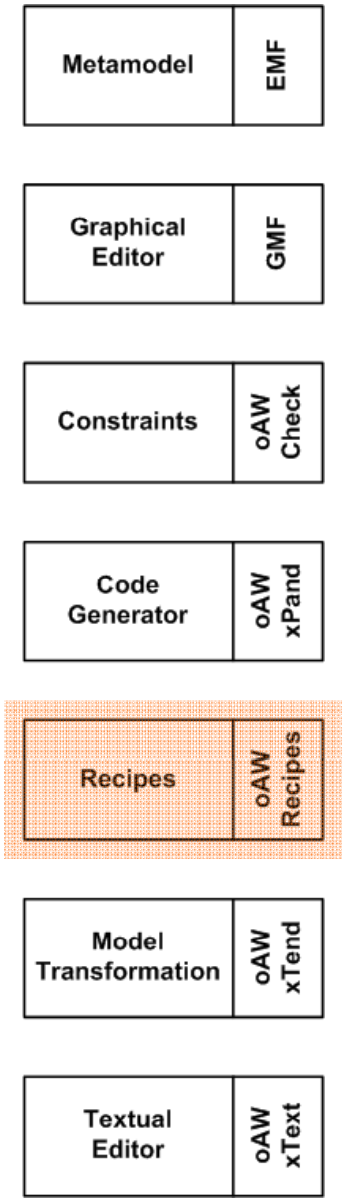
- There are various ways of integrating generated code with non-generated code:





Recipes II

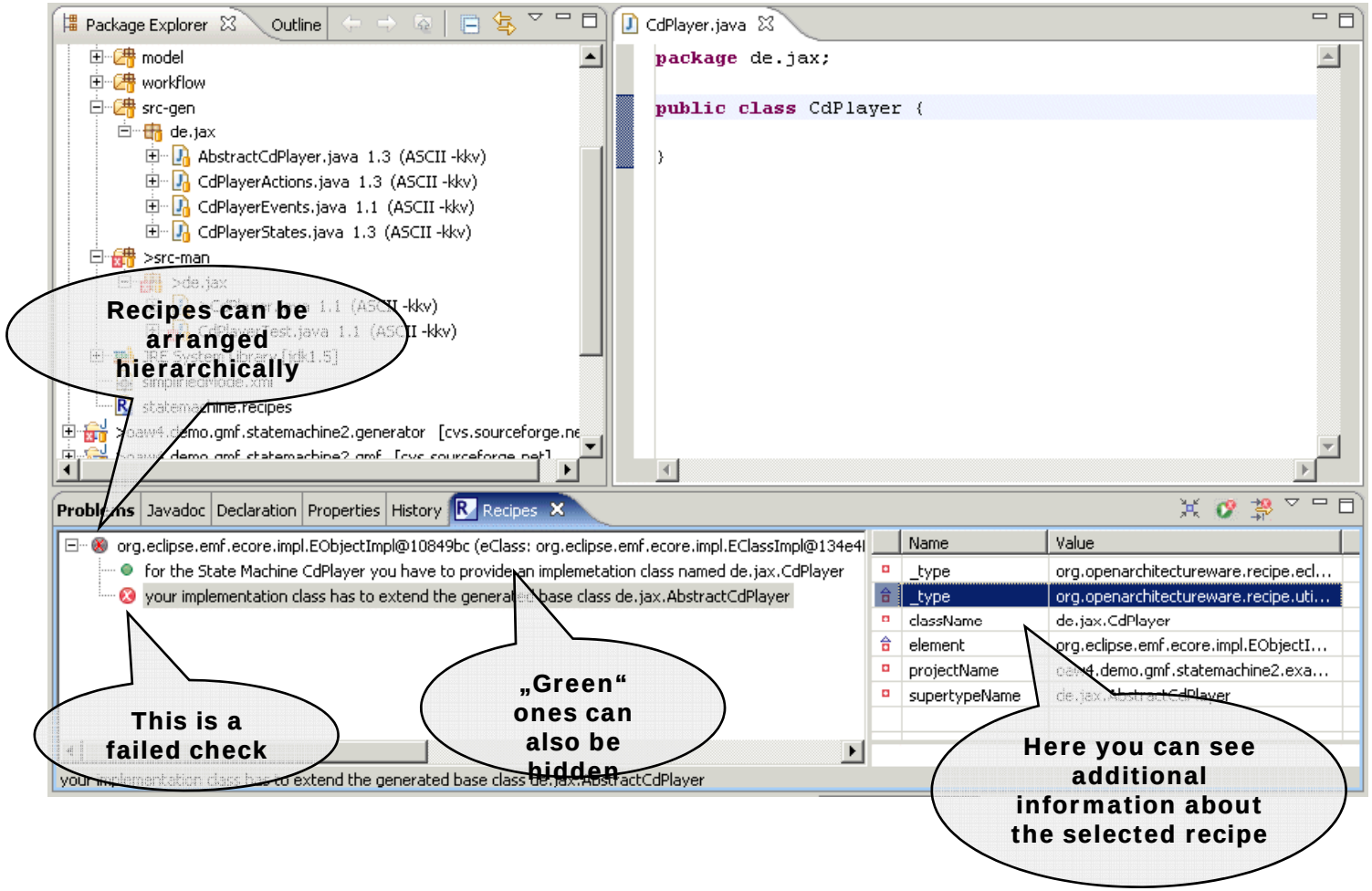
- To help developers to “do the right thing” **after the generator** has created base classes and the like, you can use a recipe framework.
- It provides a **task-based approach** to “completing” the generated code with manual parts.
- This works the following way:
  - As part of the generator run, **you instantiate checks** that you write to a file
  - After the generator finishes, the **IDE** (here: Eclipse) loads these checks and verifies them against the complete code base (i.e. Generated + manual)
  - If things don’t conform to the rules, **messages are output** helping the developer to fix things.
- For example, in the state machine case, actions must be implemented in subclasses.



© 2006-7 Völter, Efftinge, Kolb

Recipes III

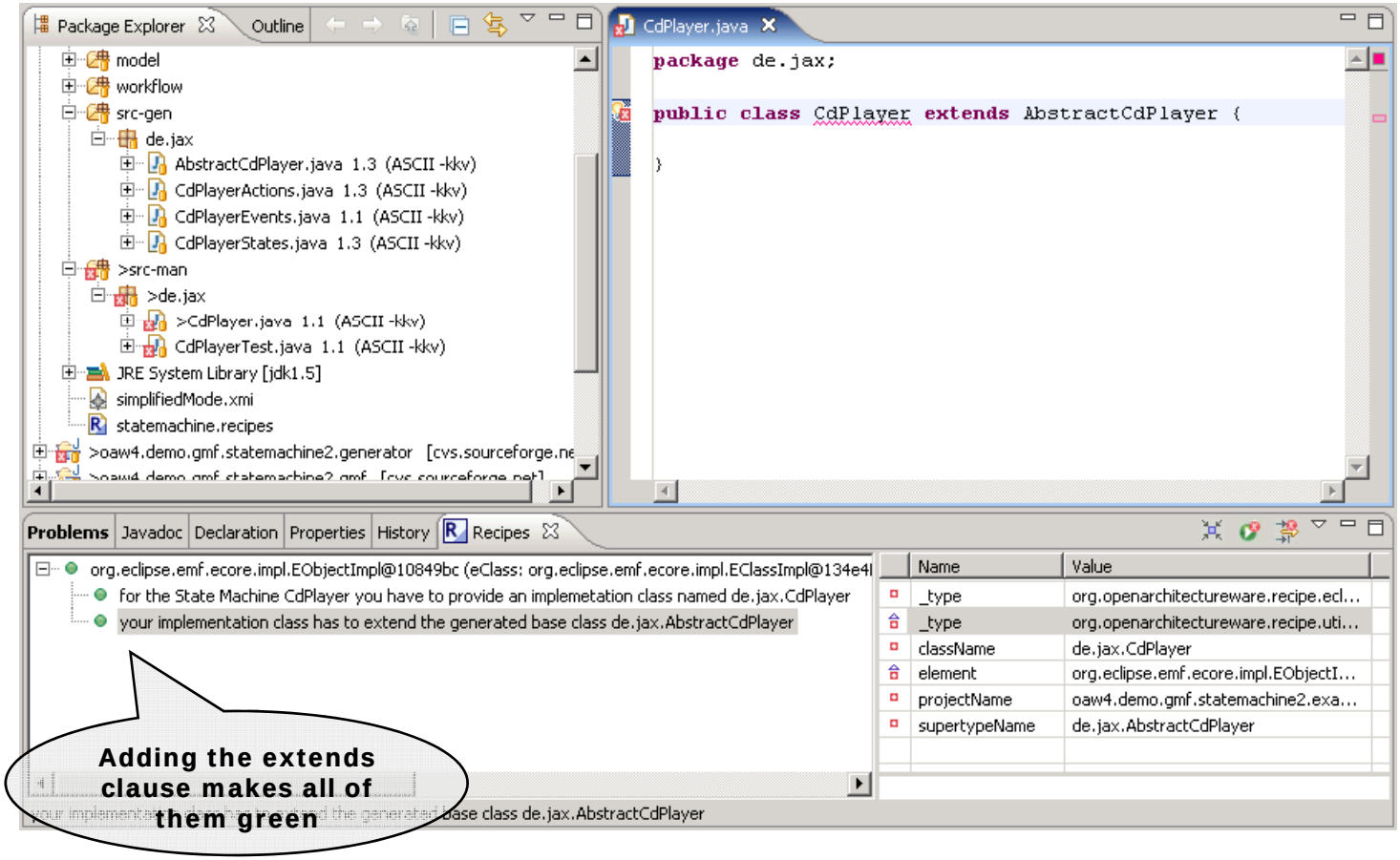
- Here's an error that suggests that I **extend** my manually written class **from the generated base class:**



Metamodel	EMF
Graphical Editor	GMF
Constraints	oAW Check
Code Generator	oAW xPand
Recipes	**oAW Recipes**
Model Transformation	oAW xTend
Textual Editor	oAW xText

Recipes IV

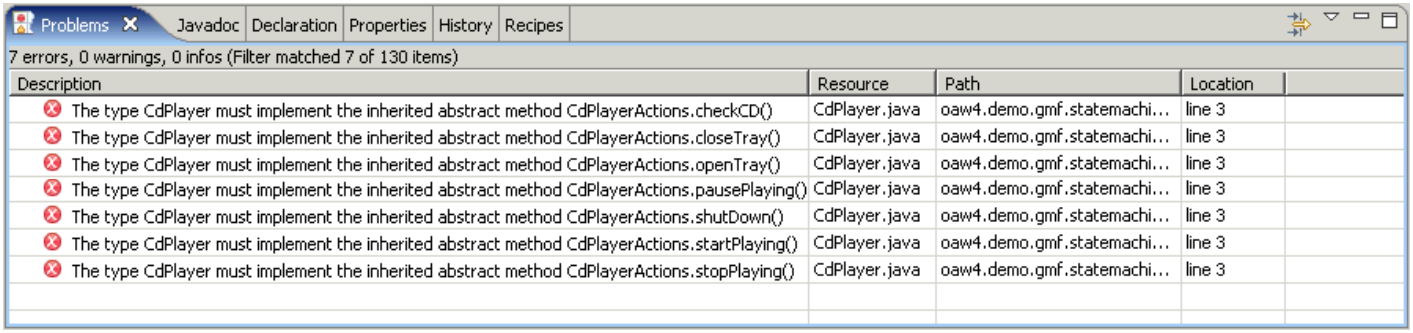
- I now **add the respective *extends* clause**, and the message goes away – automatically.



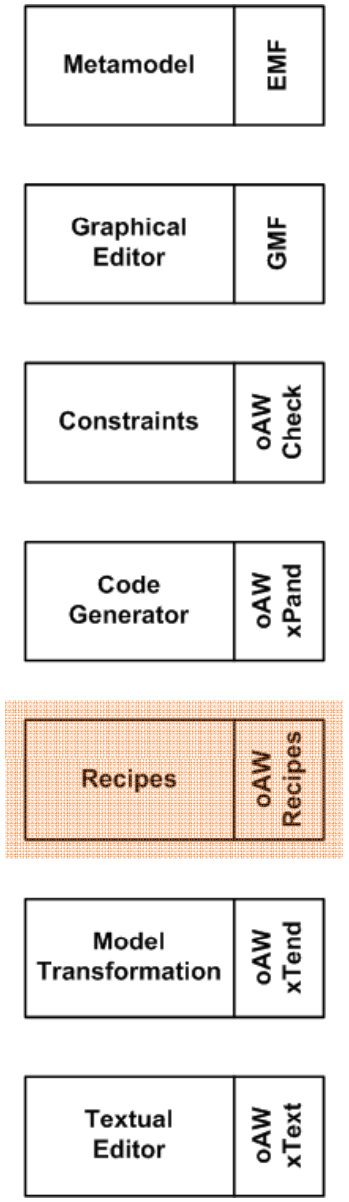
© 2006-7 Völter, Efftinge, Kolb

Recipes V

- Now I get a number of compile errors because I have to **implement the abstract methods** defined in the super class:



- I finally implement them sensibly, and everything is ok.
- The Recipe Framework and the Compiler have **guided me through the manual implementation steps.**
  - If I didn't like the compiler errors, we could also add recipe tasks for the individual operations.
  - oAW comes with a number of **predefined recipe checks for Java**. But you can also define your own checks, e.g. to verify C++ code.



© 2006-7 Völter, Efftinge, Kolb

Recipes VI

- Here's the **implementation of the Recipes**. This workflow component must be added to the workflow.

RecipeCreator.java

```
package oaw4.demo.gmf.statemachine2.recipe;

import java.util.ArrayList;

public class RecipeCreator extends AbstractExpressionRecipeCreator {

    @Override
    protected Collection internalCreateRecipes(ExpressionFacade facade,
            String project) {
        List<Check> checks = new ArrayList<Check>();
        Object sm = facade.evaluate("this");
        String name = (String) facade.evaluate("name");

        ElementCompositeCheck ecc = new ElementCompositeCheck(sm,
                "statemachine implementation must be completed.");
        checks.add( ecc );

        String implClassName = (String)facade.evaluate("fqImplClassName()");
        String implBaseClassName = (String)facade.evaluate("fqImplBaseClassName()");
        JavaClassExistenceCheck extCheck = new JavaClassExistenceCheck(
                "for the State Machine "+name+" you have to provide an implemetation class named "
                +implClassName,
                project, implClassName );
        ecc.addChild( extCheck );
        JavaSupertypeCheck superCheck = new JavaSupertypeCheck(
                "your implementation class has to extend the generated base class "
                +implBaseClassName,
                project, implClassName, implBaseClassName );
        ecc.addChild( superCheck );
        return checks;
    }

}
```

You extend one of a number of suitable base classes...

...and override a suitable template method

You can then create any number of checks.

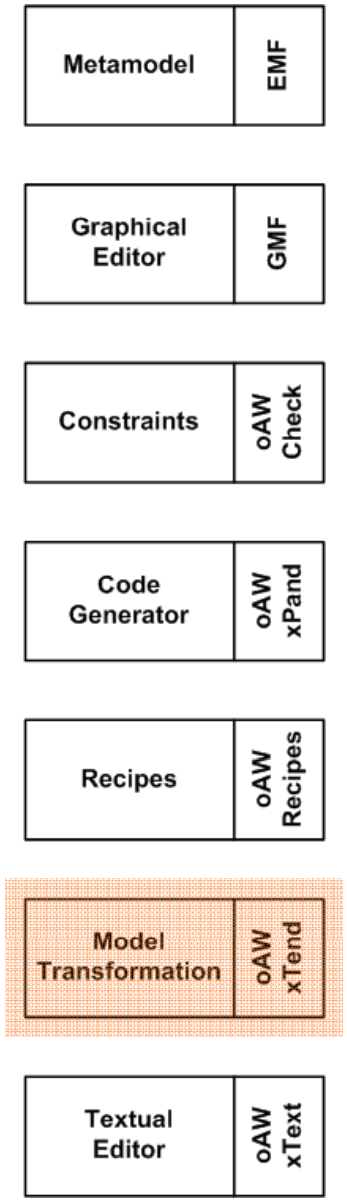
This one checks that a class extends another one

And return the checks to the framework

Metamodel	EMF
Graphical Editor	GMF
Constraints	oAW Check
Code Generator	oAW xPand
Recipes	**oAW Recipes**
Model Transformation	oAW xTend
Textual Editor	oAW xText

Model Transformations I

- **Model Transformations** create one or more new models from one or more input models. The input models are left unchanged.
  - Often used for stepwise refinement of models and modularizing generators
  - Input/Output Metamodels are different
- **Model Modifications** are used to alter or complete an existing model
- For both kinds, we use the **xTend language**, an extension of the openArchitectureWare expression language.
- **Alternative languages** are available such as ATL, MTF or Tefkat (soon: various QVT implementations)



Model Transformation II

- The **model modification** shows how to add an additional state and some transitions to an existing state machine (emergency shutdown)

AddEmergencyShutdown.ext

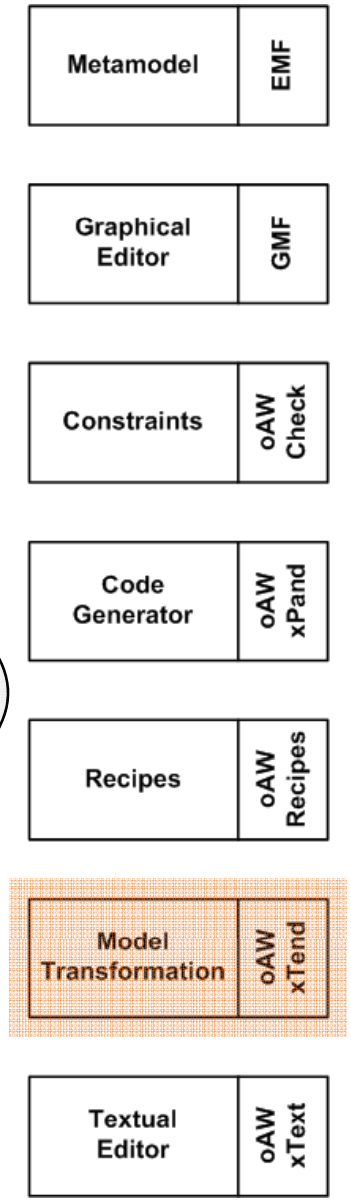
```
import statemachine2;

extension statemachine2::constraints::Statemachine;

StateMachine modify(StateMachine sm) :
   sm.transitions.addAll(sm.allConcreteStates().createTransition()) ->
   sm.states.add(createShutDown()) ->
   sm;

private create State this createShutDown() :
   setName("EmergencyShutDown");

private create Transition this createTransition(State s) :
   setEvent("Error")->
   setName("Aborting") ->
   setFrom(s) ->
   setTo(createShutDown());
```

Extensions can import other extensions

The main function

„create extensions“ guarantee that for each set of parameters the *identical* result will be returned.

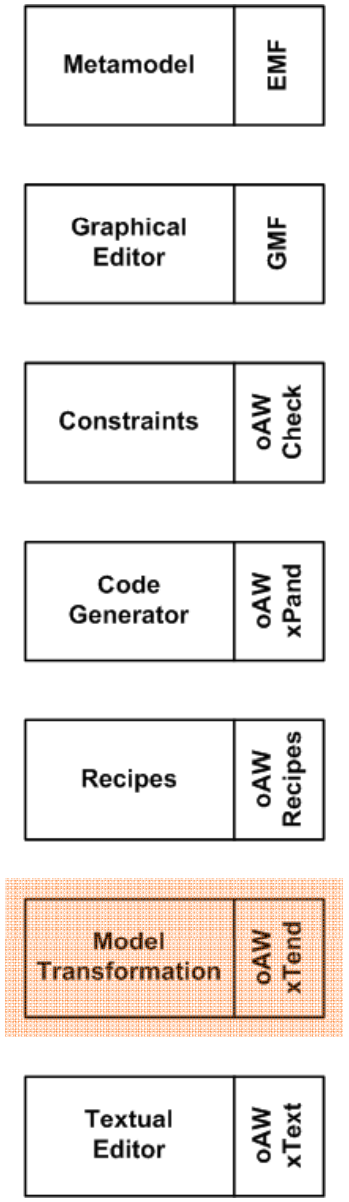
Therefore createShutDown() will always return the same element.



© 2006-7 Völter, Efftinge, Kolb

Model Transformation III

- The generator is based on an **implementation-specific metamodel** without the concept of composite states.
- This makes the **templates simple**, because we don‘t have to bridge the whole abstraction gap (from model to code) in the templates.
- Additionally, the **generator is more reusable**, because the abstractions are more general.
- We will show a transformation which transforms models described with our GMF editor into models expected by the generator.



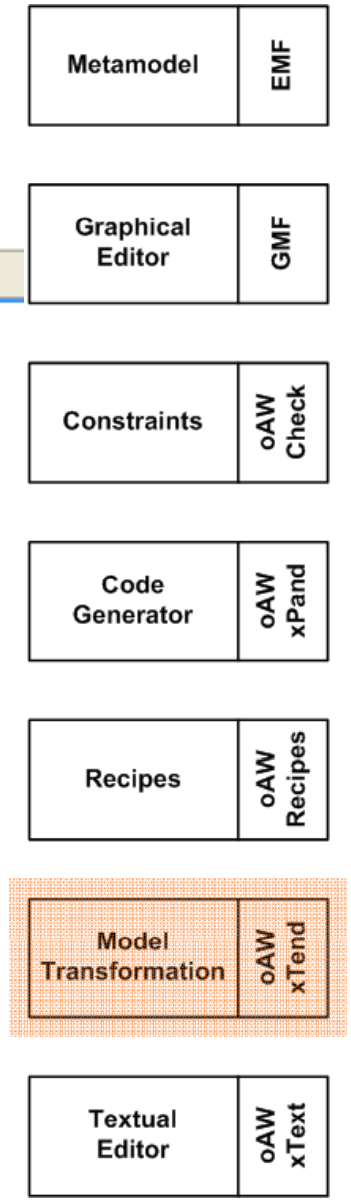
© 2006-7 Völter, Efftinge, Kolb

Model Transformation IV

- We want to transform from the editor's metamodel '**statemachine2**' to the generator's metamodel '**simpleSM**'

```
import statemachine2;

extension statemachine2::constraints::Statemachine;
extension org::openarchitectureware::util::IO;

create simpleSM::StateMachine createStatemachine(StateMachine sm) :
  setName(sm.name) ->
  setInitialState(sm.concreteState().createState() ) ->
  states.addAll(sm.allConcreteStates().createState()) ->
  actions.addAll(sm.eAllContents.typeSelect(Action).name.createAction()) ->
  events.addAll(sm.eAllContents.typeSelect(Transition).event.withoutName(null).createEvent());

private create simpleSM::State createState(State s) :
  setName(s.name) ->
  transitions.addAll(s.allOutTransitions().createTransition(s));

private create simpleSM::Action createAction(String n) :
  setName(n);

private create simpleSM::Event createEvent(String n) :
  setName(n);

private create simpleSM::Transition createTransition(Transition t,State s) :
  actions.addAll(allActions(s,t.to.concreteState()).name.createAction()) ->
  setEvent(t.event.createEvent()) ->
  setTo(t.to.concreteState().createState());
```

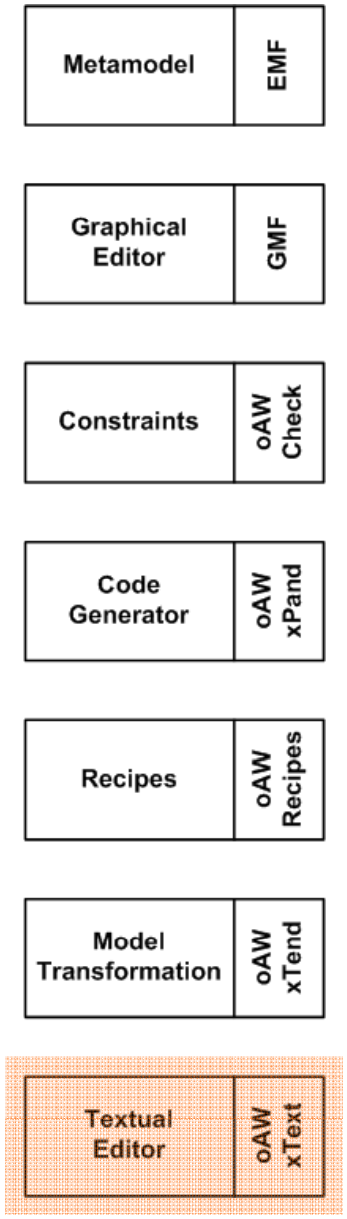
- We need to 'normalize' composite states.
- States inherit outgoing transitions from their parent states
- For those transitions the exit actions are inherited, too
- Unify action and event elements with the same name



© 2006-7 Völter, Efftinge, Kolb

Textual Editor I

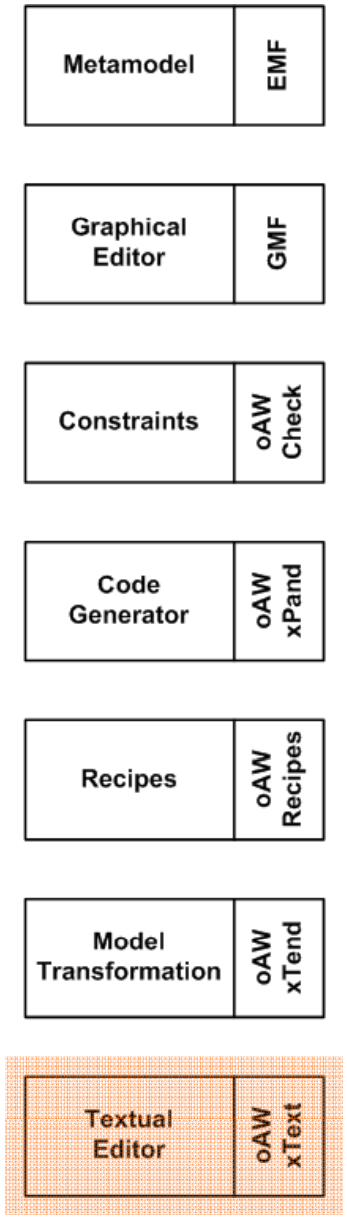
- A graphical notation is not always the best syntax for DSLs.
- So, while GMF provides a means to generate editors for graphical notations, we also need to be able to come up with **editors for textual syntaxes**.
- These **editors need to include** at least
  - Syntax hightlighting
  - Syntax error checking
  - Semantic constraint checking



© 2006-7 Völter, Efftinge, Kolb

Textual Editor II

- We use oAW's textual DSL generator framework **xText**
- Based on a BNF-like language it provides:
  - An **EMF-based metamodel** (representing the AST)
  - An **Antlr parser** instantiating **dynamic** EMF-models
  - An **Eclipse text editor plugin** providing
    - **syntax highlighting**
    - An **outline view**,
    - **syntax checking**
    - as well as **constraints checking** based on a *Check* file, as always oAW



© 2006-7 Völter, Efftinge, Kolb

Textual Editor III

- The **grammar** (shown in the boostrapped editor)
- The **generated** eCore AST model

```
CompositeState :
    "statemachine" name=ID "{"
        (entryActions+=Action)*
        (transitions+=Transition)*
        (exitActions+=Action)*
    "}";

Abstract AbstractState :
  CompositeState | State;

State :
    "state" name=ID "{"
        (entryActions+=Action)*
        (transitions+=Transition)*
        (exitActions+=Action)*
    "}";

Action :
    "@" name=ID;

Transition :
    event=ID "->" state=ID;
```

A literal

The first rule describes the root element of the AST

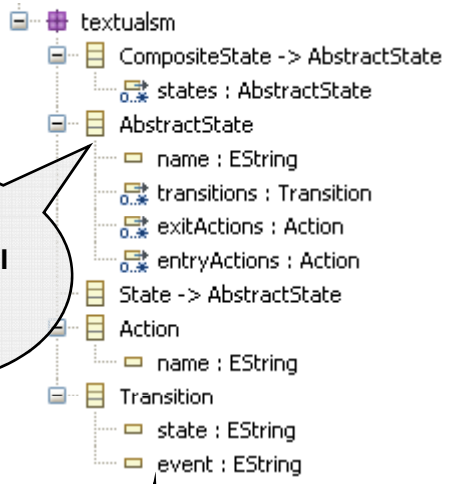
Rule name

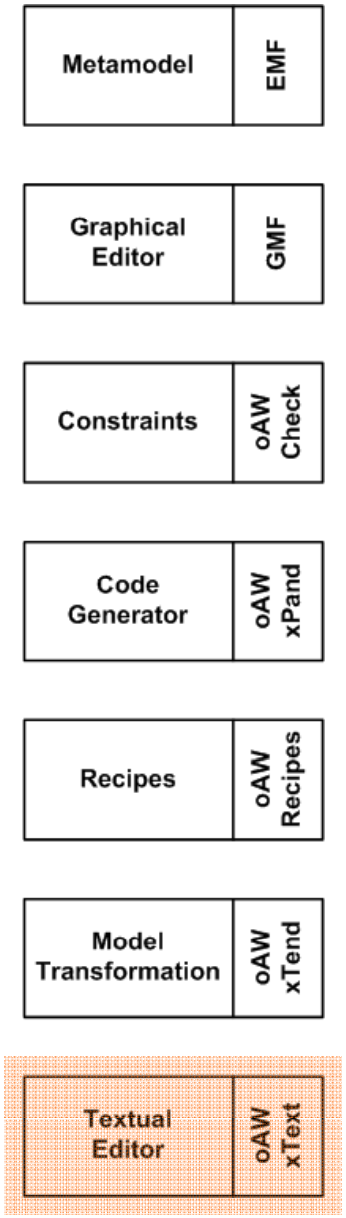
Rule names will become the AST classes

States contain a number of entry actions, transitions and exit actions

Assigns an indentifier to a variable (here: state)

These variables will become attributes of the AST class

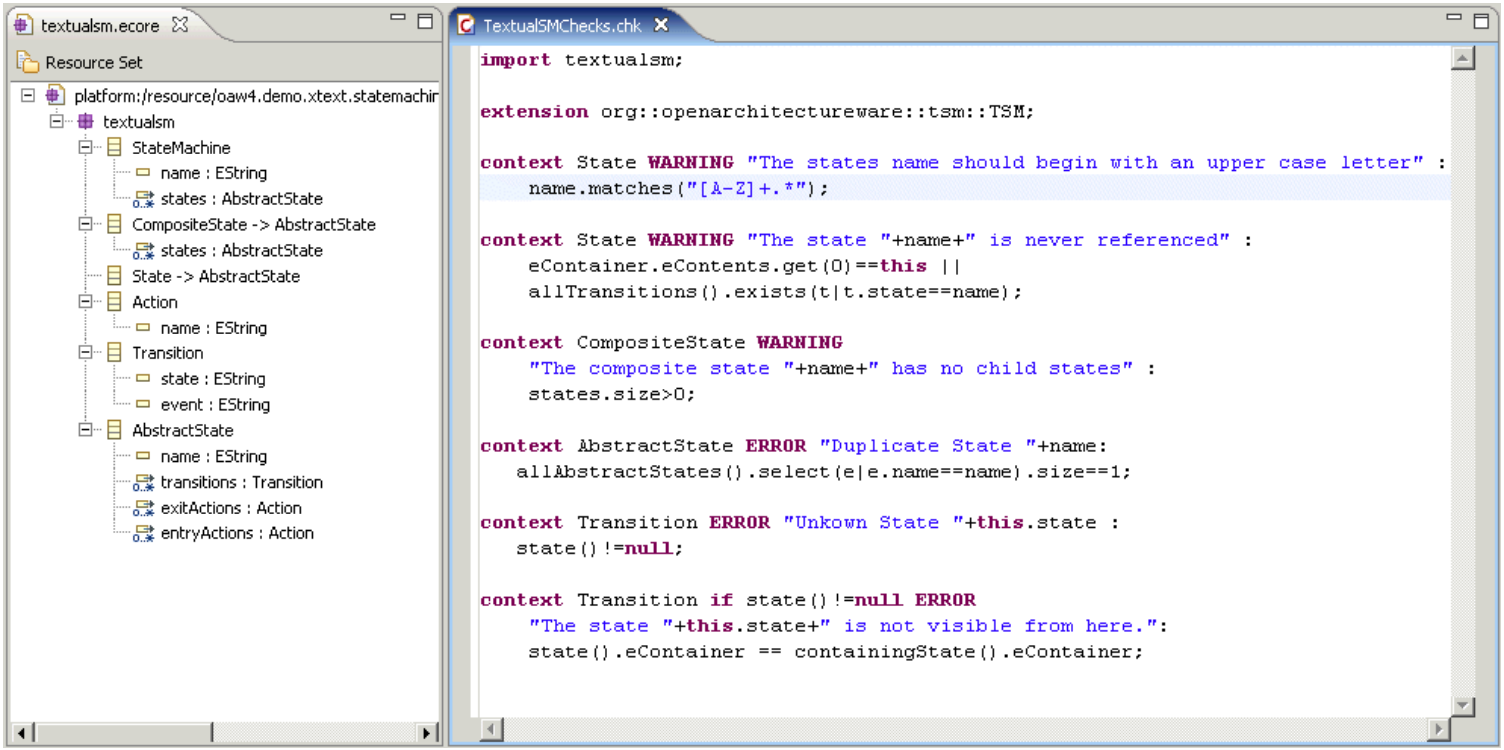


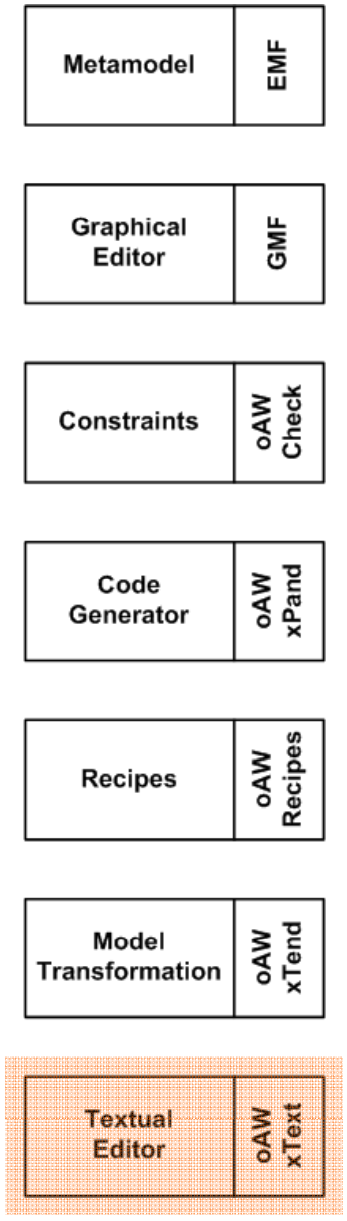


© 2006-7 Völter, Efftinge, Kolb

Textual Editor IV

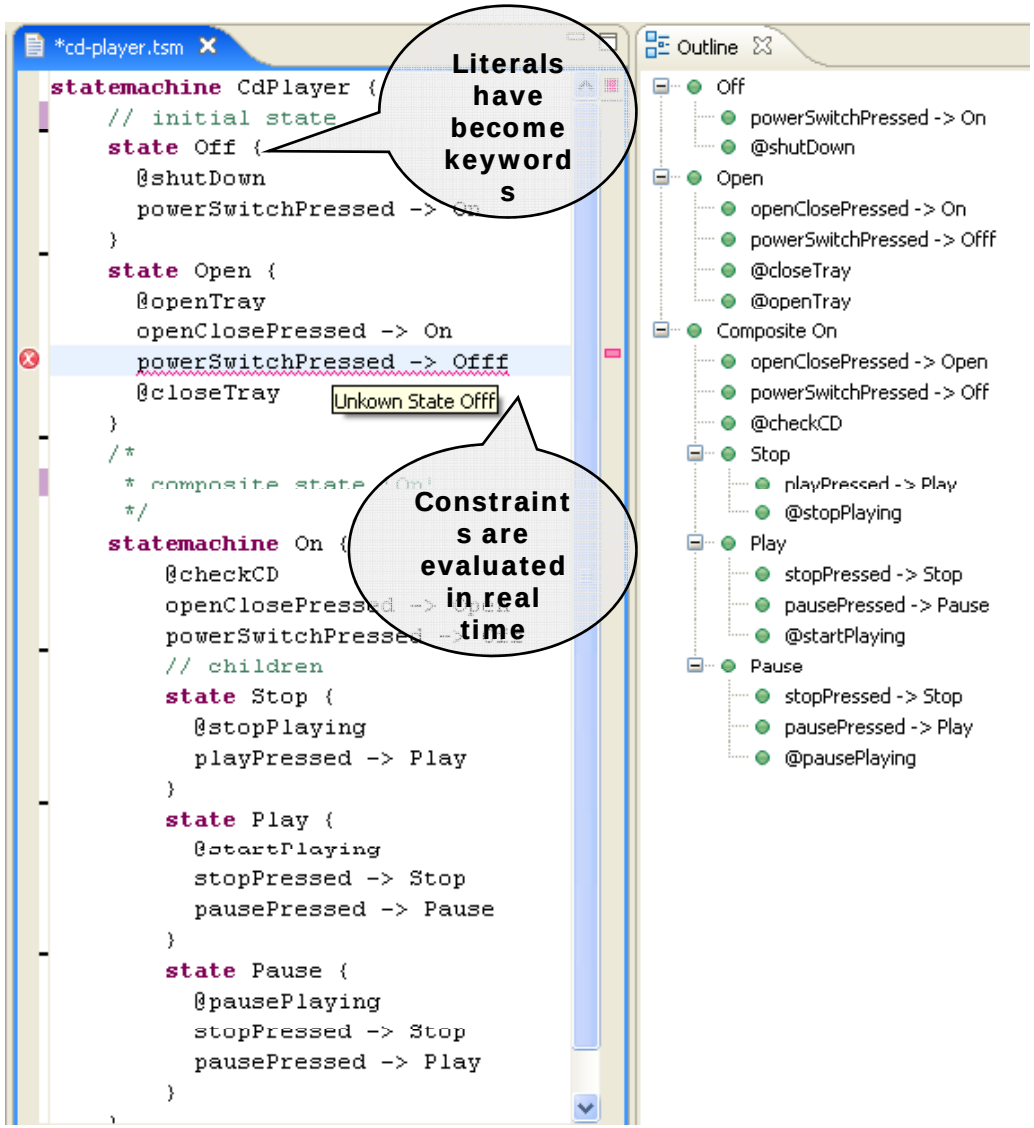
- You can define **additioal constraints** that should be validated in the generated editor.
- This is based on oAW's *Check* language
  - i.e. These are constraints like all the others you've already come across

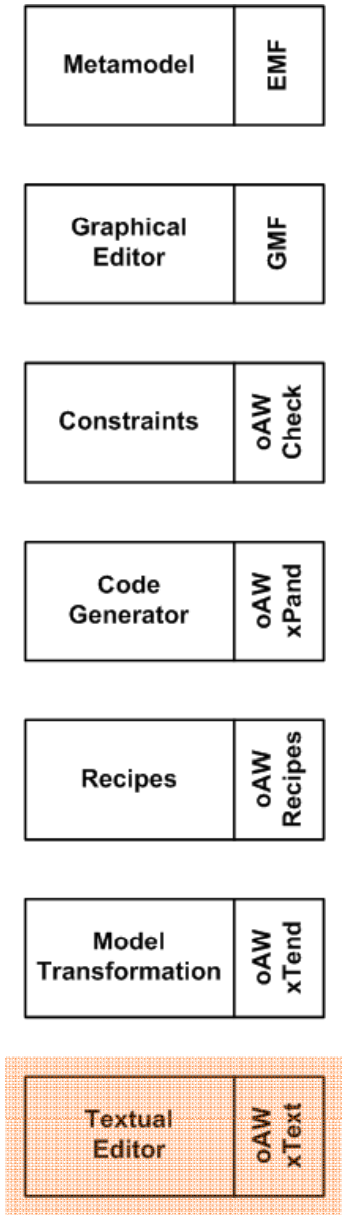




Textual Editor V

- The **generated editor** and it's **outline view**

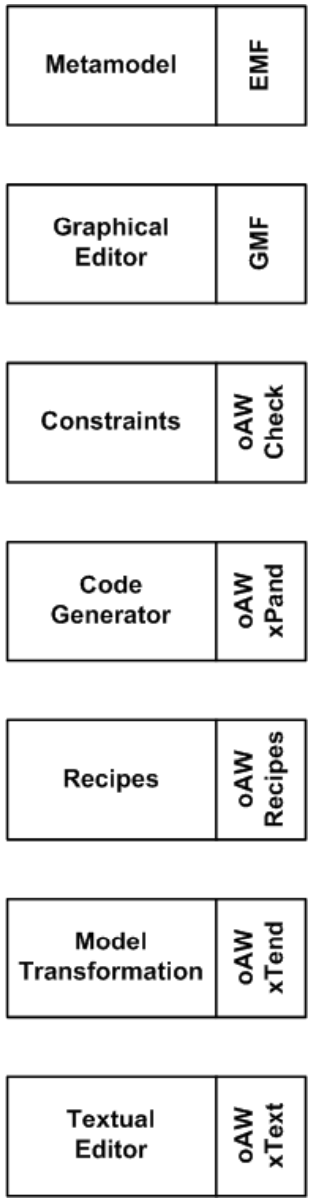




© 2006-7 Völter, Efftinge, Kolb

Summary

- The tool chain we've just shown provides an **end-to-end solution for MDSD,**
  - Completely Open Source
  - Using standards wherever worthwhile,
  - And pragmatic solutions wherever necessary.
- To get the tools, go to
  - www.eclipse.org/emf
  - www.eclipse.org/gmf
  - www.openarchitectureware.org, www.eclipse.org/gmt/oaw
- THANK YOU.



© 2006-7 Völter, Efftinge, Kolb