



How MDSD improves software quality

... as well as a little bit of advertising for
openArchitectureWare ☺

Markus Völter

voelter@acm.org
www.voelter.de



About me



Markus Völter

voelter@acm.org

www.voelter.de

- Independent Consultant
- Based out of Heidenheim, Germany
- Focus on
 - Model-Driven Software Development
 - Software Architecture
 - Middleware



Advantages of MDSD

- Using MDSD can result in a **variety of advantages**, such as
 - Better integration of domain experts
 - Better testability
 - More efficient development
 - Architecture management and enforcement
 - Improved code quality & consistency
- In this talk I want to **focus on the latter two**, subsumed by the term „improved software quality“



Refining the Architecture

Building a Domain Architecture improves the architectural quality of the target system



Three kinds of Architecture

- **Conceptual Architecture**

Definition of the artifacts available for building systems as well as their properties, characteristics and interactions

-> **Metamodels**

- **Technical Architecture**

Mapping of the Conceptual Architecture to one or more technology platforms, taking into account the non-functional requirements

-> **Automatic Mapping Code Generation**

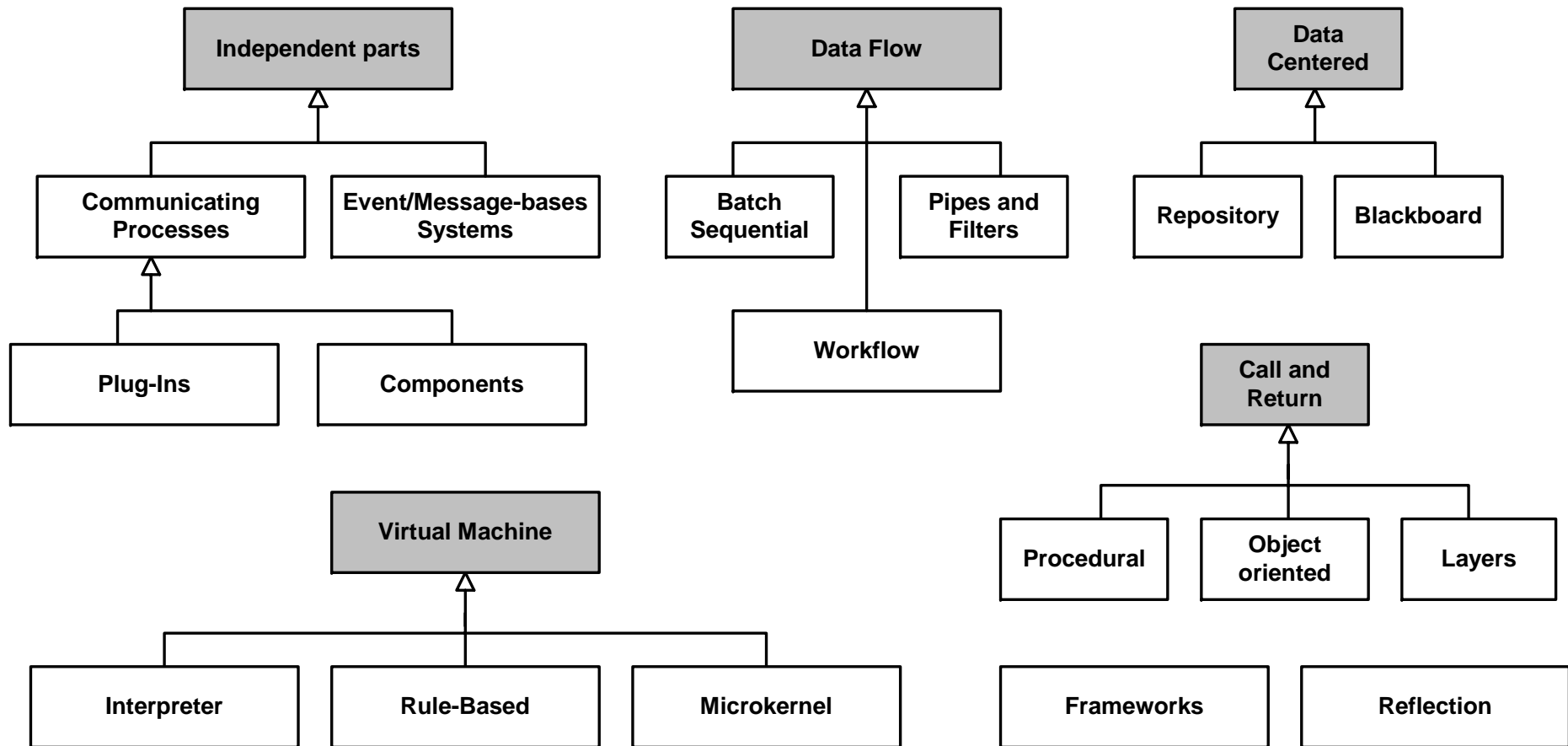
- **Functional Architecture**

Definitions of a system (as instances of the conceptual artifacts) that implements the functional requirements

-> **Easier to describe based on the metamodels**

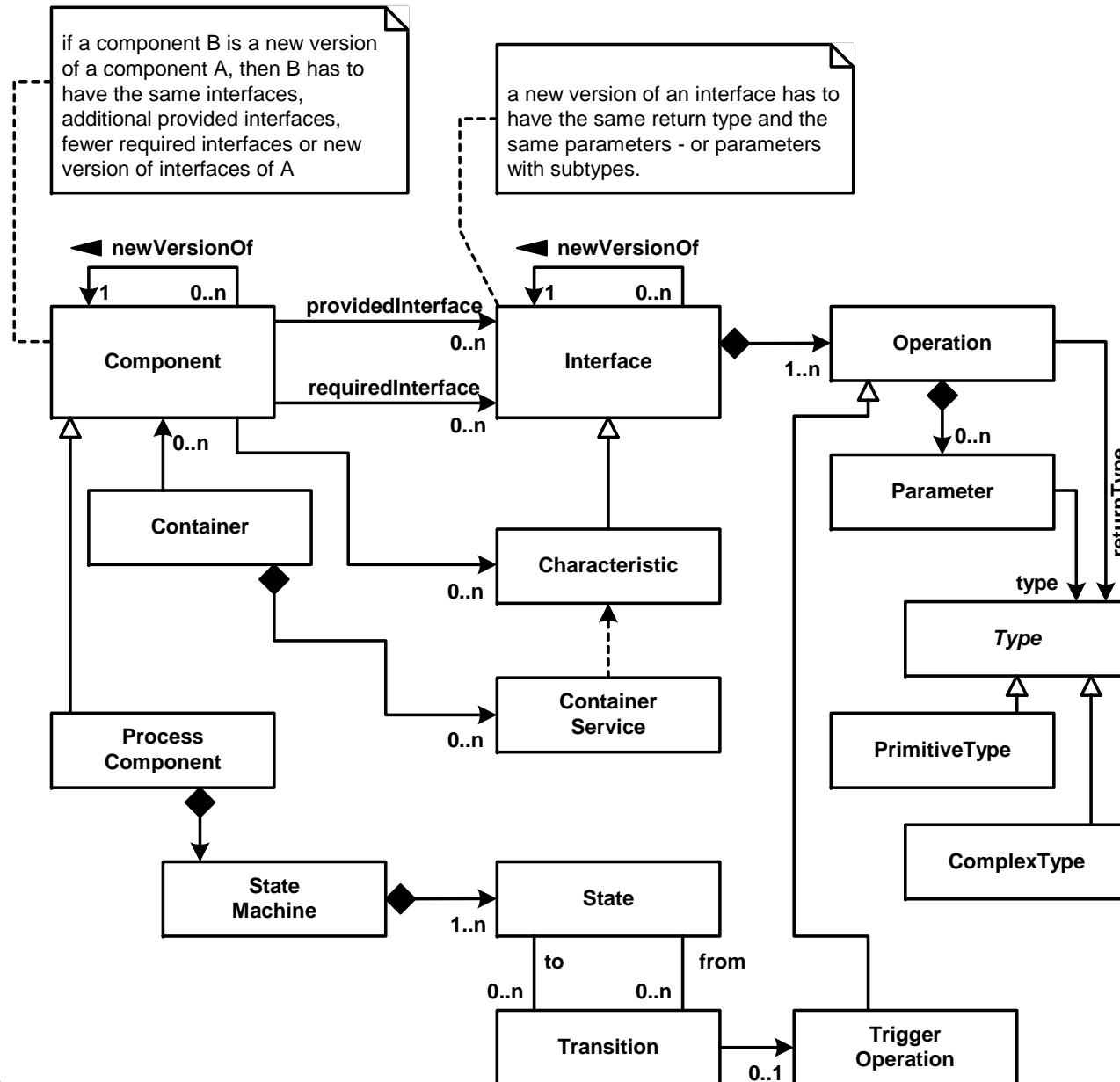


Conceptual Architecture: Starting Points





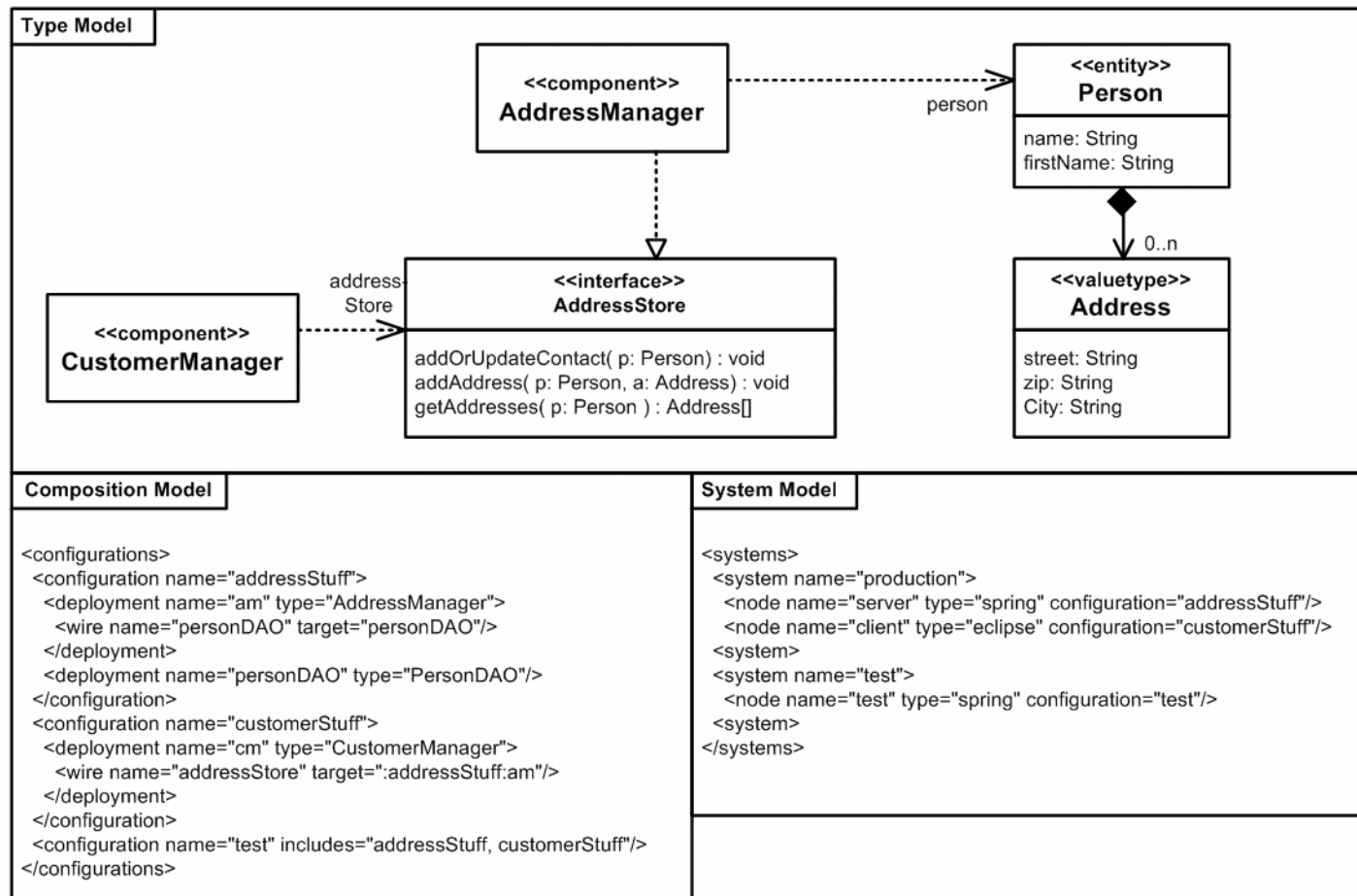
Formalization of Architecture in MDSD using Metamodels



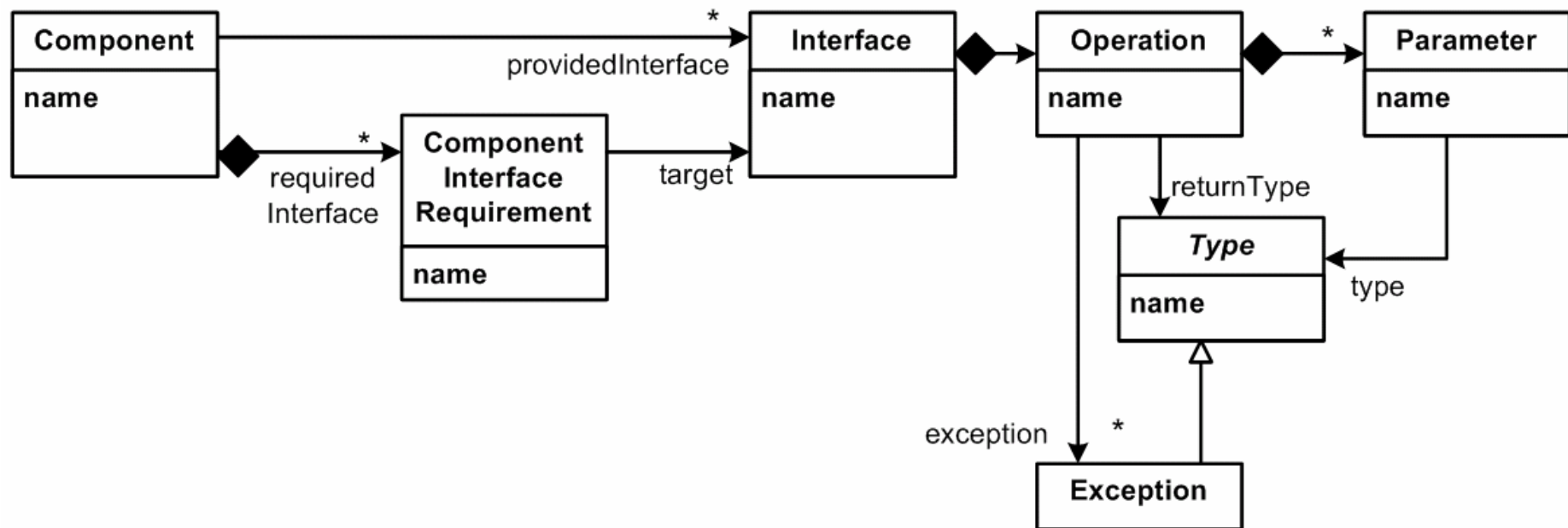


Multi-Viewpoint models

- **Type Model:** Components, Interfaces, Data Types
- **Composition Model:** Instances, "Wirings"
- **System Model:** Nodes, Channels, Deployments

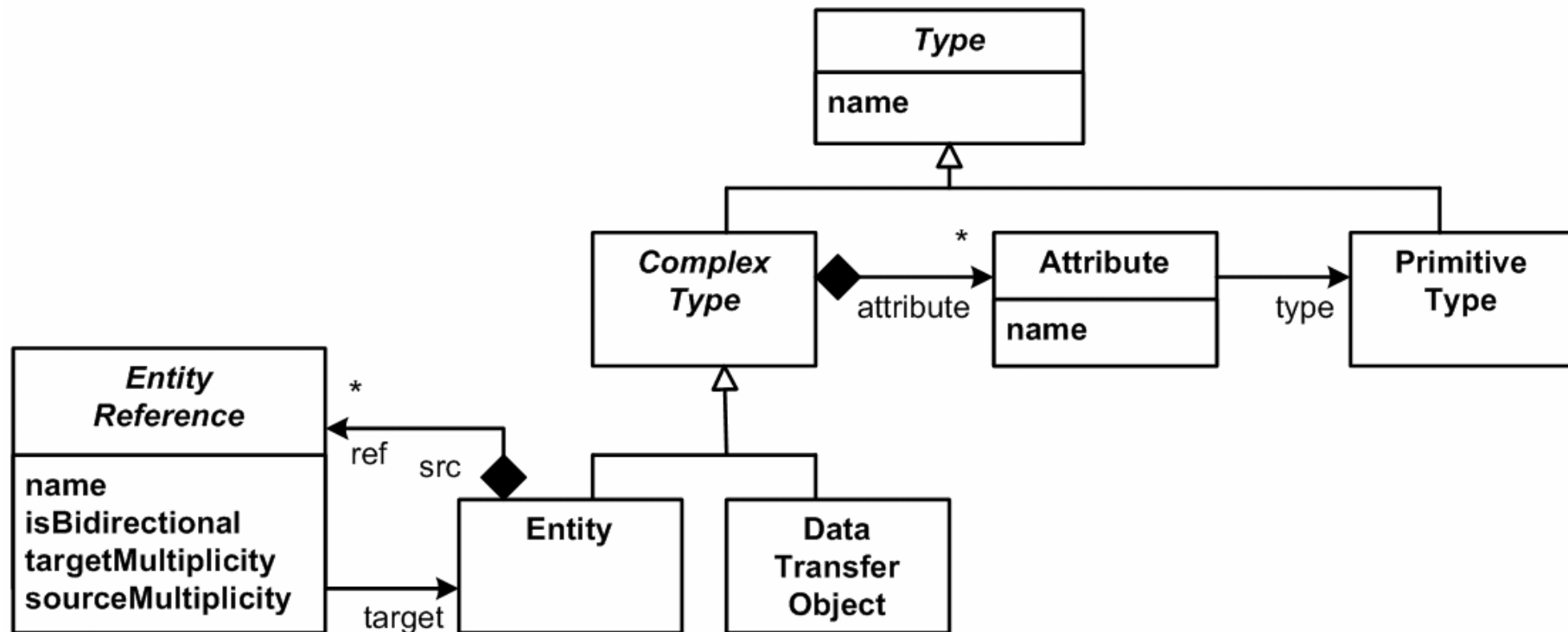


Type Metamodel



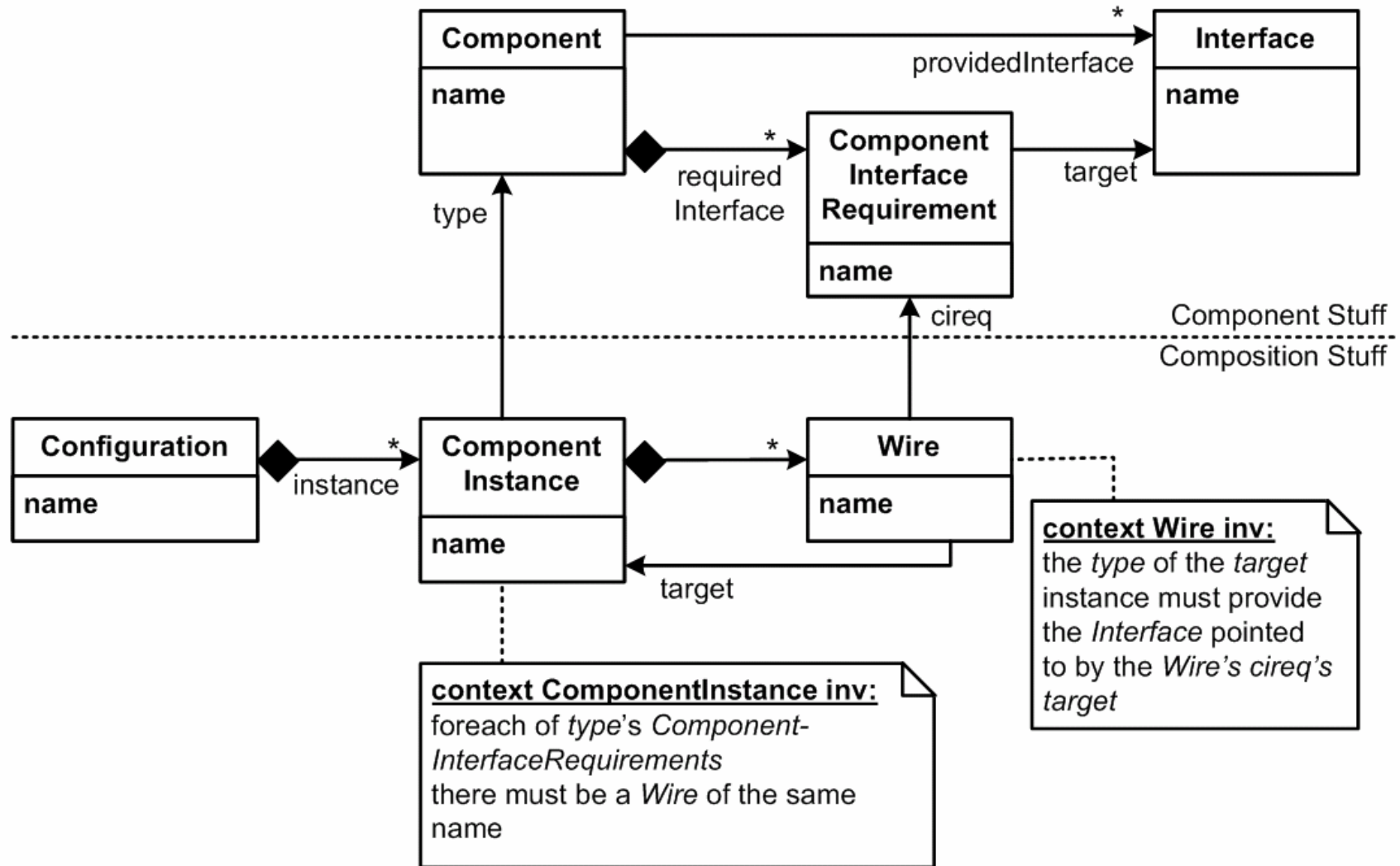


Type Metamodel II (Data)



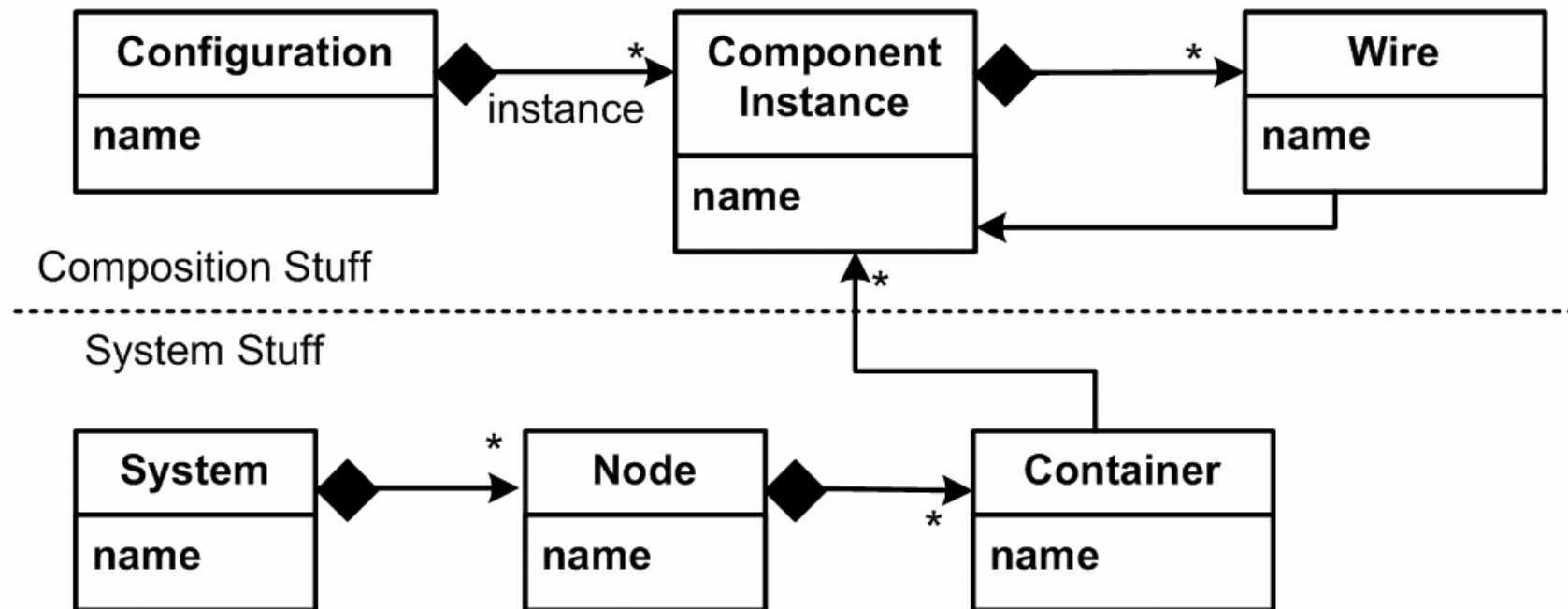


Composition Metamodel





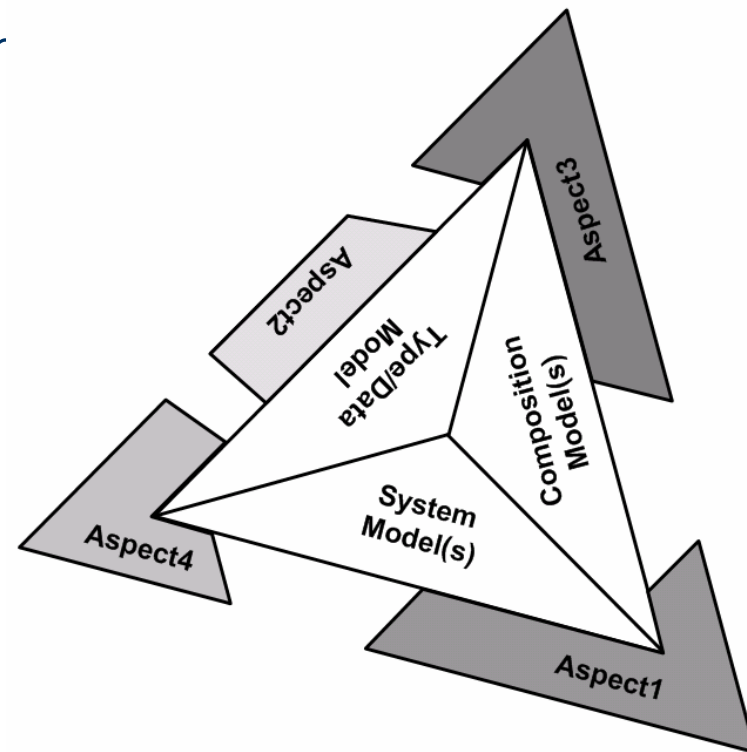
System Metamodel





Aspect Models

- Often, the described three viewpoints are not enough, **additional aspects** need to be described.
- These go into **separate aspect models**, each describing a well-defined aspect of the system.
 - Each of them uses a suitable DSL/syntax
 - The generator acts as a weaver
- Typical **Examples** are
 - Persistence
 - Security
 - Forms, Layout, Pageflow
 - Timing, QoS in General
 - Packaging and Deployment
 - Diagnostics and Monitoring





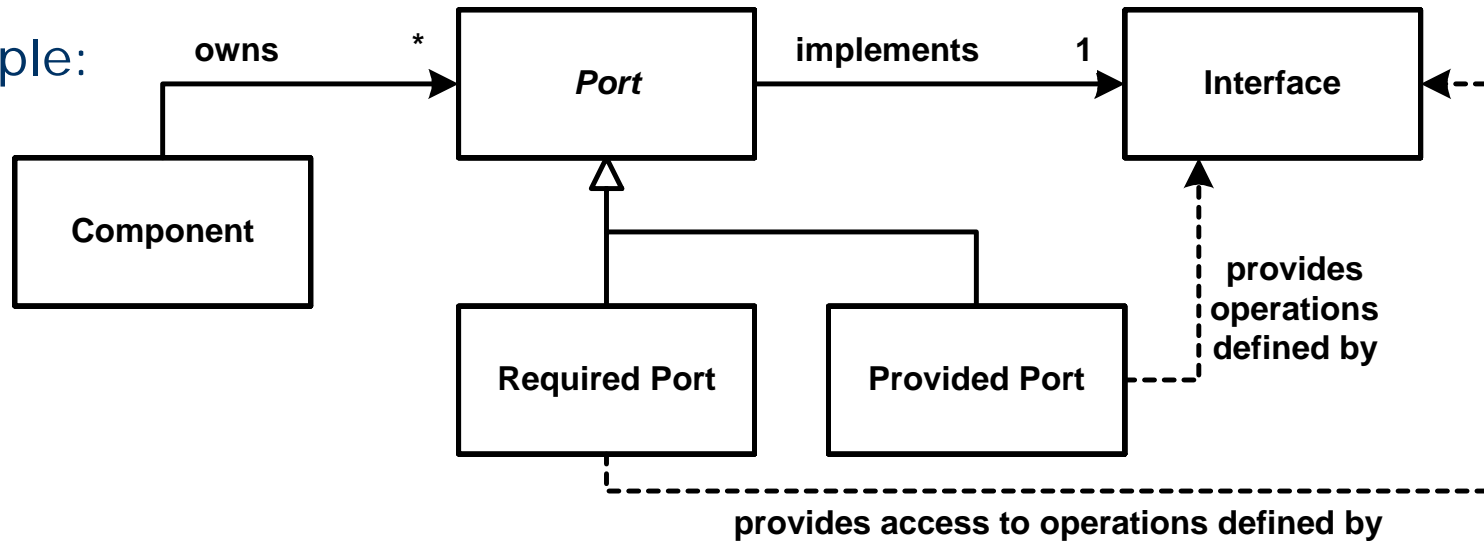
Talk Metamodel

- In order to **continuously improve and validate** the FORMAL META MODEL for a domain, it has to be **exercised** with domain experts as well as by the development team.
- In order to achieve this, it is a good idea to use it during discussions with stakeholders by **formulating sentences** using the concepts in the meta model.
- As soon as you find that you **cannot express something using sentences** based on the meta model,
 - you have to reformulate the sentence
 - the sentence's statement is just wrong
 - you have to update the meta model.



Talk Metamodel II

- Example:



- A component owns any number of ports.
- Each port implements exactly one interface.
- There are two kinds of ports: required ports and provided ports.
- A provided port provides the operations defined by its interface.
- A required port provides access to operations defined by its interface.

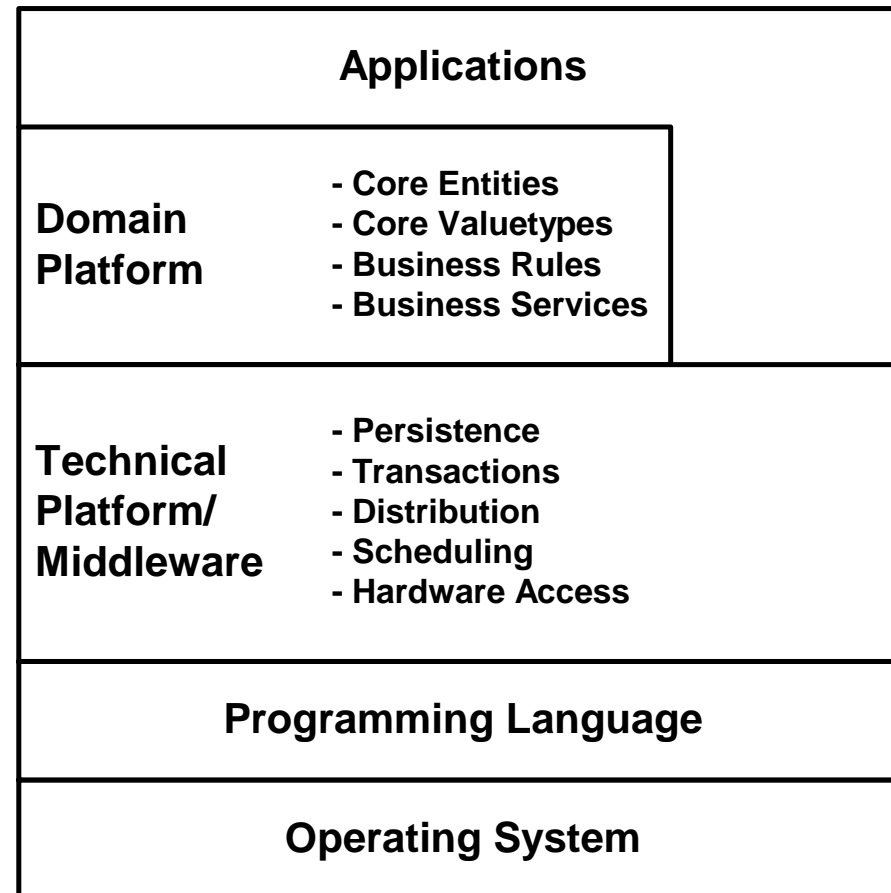


Technical Architecture

You can generate all the „adaption code“ to run the system on a given platform – you don't need to care about these things when implementing business logic



Technical Architecture - Blueprint





Three Basic Viewpoints – Generated Stuff

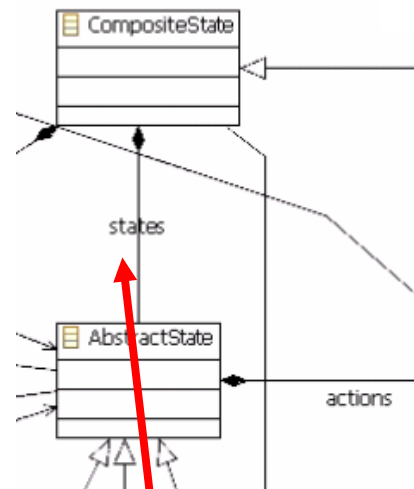
- What can be generated?
 - Base classes for component implementation
 - Build-Scripts
 - Descriptors
 - Remoting Infrastructure
 - Persistence
 - ...



Code Generation



- Code Generation is used to **generate executable code** from models.
- Code Generation is **based on the metamodel** & 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»
  
```



Code Generation II

```

Actions.xpt  Statemachine.xpt x
«IMPORT simpleSM»
«EXTENSION templates::GeneratorUtil»

«DEFINE file FOR StateMachine»
  «FILE basePath()+"/Abstract"+name.toFirstUpper()+".java"»
  package «basePackage ()»;

  abstract class «implBaseClassName» «Name ()» {
    «statesEnumName ()» currentS
    state boolean terminated = false;

    public void handleEvent( «eventsEnumName ()» event,
      if ( terminated ) throw new RuntimeException( "this sm is terminated!" );

    switch ( currentState ) {
      «FOREACH states AS s-»
      case «s.shortStateId()»:
        «FOREACH s.transitions AS t-»
          if ( event == «t.event.eventId(this)»
            «EXPAND executeTransition(this)»
            break;
          «EXPAND handleIllegalTransition»
        «ENDFOREACH»
      break; // break out if no suitable transition has been found!
    «ENDFOREACH»

    public «statesEnumName ()» getCurrentState(
      return currentState;
    }
  }
«ENDEFFINE»

«DEFINE handleIllegalTransition FOR StateMachine»
«ENDEFFINE»

«DEFINE executeTransition(StateMachine sm) FOR Transition»
  «FOREACH actions AS a-»
  this.«a.methodName ()» ();
«ENDFOREACH»
  currentState = «to.stateId(sm)»;
«ENDEFFINE»

```

Namespace & Extension Import

Opens a File

Name is a property of the State-Machine class

Iterates over all the states of the State-Machine

Calls another template

Extension Call

Template name

Like methods in OO, templates are associated with a (meta)class

- 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**.



Code Generation III



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

```

GeneratorUtil.ext x
import simpleSM;

String basePath() : basePackage()
String basePackage() : "de.jax";

String constantName(Named this): name.toUpperCase();
String methodName(Action this) : name.toFirstLower();

String implBaseClassName(StateMachine this) : ""
String implClassName(StateMachine this) : name.toFirstLower();
String fqImplBaseClassName(StateMachine this): basePackage()+"."+implBaseClassName();
String fqImplClassName(StateMachine this) : basePackage()+"."+implClassName();
  
```

Imports a namespace

Extensions are typically defined for a metaclass

Extensions can also have more than one parameter

- 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



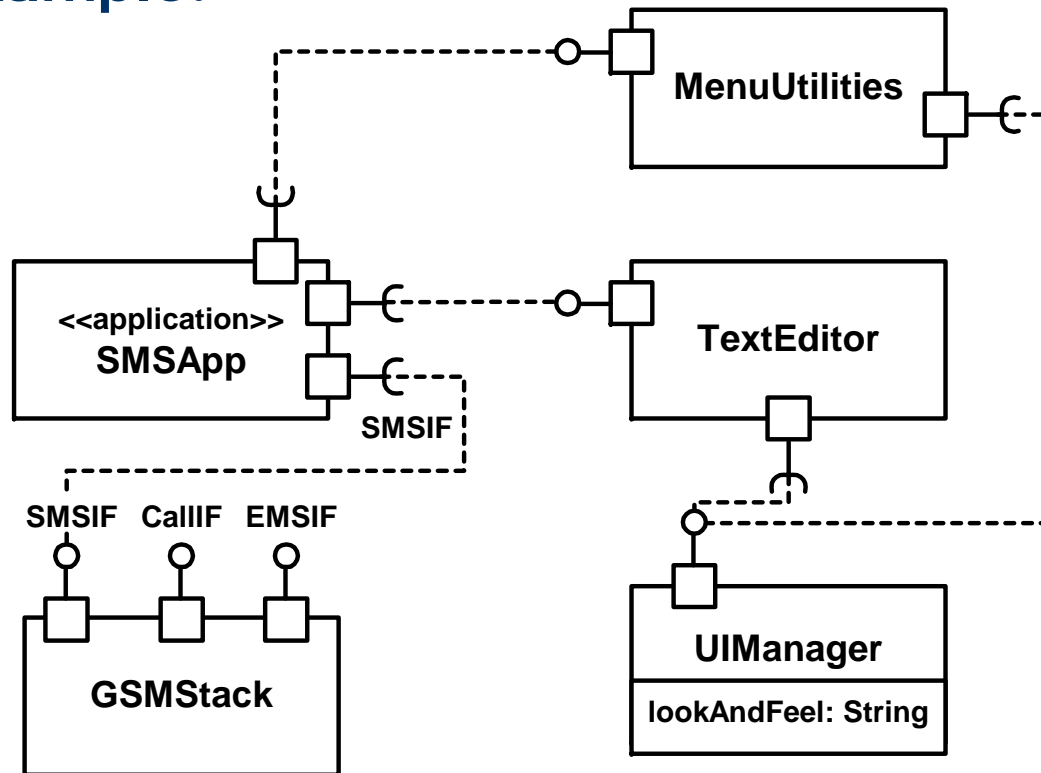
Managing Architecture

MSD can help to make sure an architecture is used consistently and „correctly“ in larger teams



Architecture „Enforcement“ using MSD

- **Example:**



- **Problem:** How do you **ensure** that developers can actually only reference (use) those components, which are declared as being used in the model?



Typical Solution, without MDSO

```
public class SMSAppImpl {
    public void tueWas() {
        TextEditor editor =
            Factory.getComponent("TextEditor");
        editor.setText( someText );
        editor.show();
    }
}
```

- **Problems:**

- Developers can lookup, use, and thus, depend on whatever they like
- Developers are not guided (by IDE, compiler, etc.) what they are allowed to access and what is prohibited



Improved Solution, with MSDS

```
public interface SMSAppContext extends ComponentContext {
    public TextEditorIF getTextEditorIF();
    public SMSIF getSMSIF();
    public MenuIF getMenuIF();
}
```

```
public class SMSAppImpl implements Component {
    private SMSAppContext context = null;
    public void init( ComponentContext ctx) {
        this.context = (SMSAppContext)ctx;
    }
    public void tueWas() {
        TextEditor editor = context.getTextEditorIF();
        editor.setText( someText ); editor.show();
    } }
}
```

- **Better, because:**
 - Developers can only access what they are allowed to...
 - ... and this is always in sync with the model
 - IDE can help developer (ctrl+space in eclipse)
 - Architecture (here: Dependencies) are enforced and controlled



The Programming Model

You can restrict the freedom of developers ...
making the code more consistent and structured



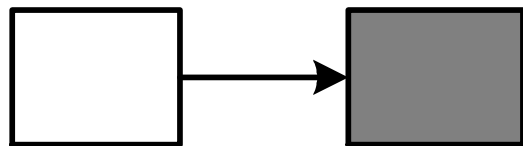
Problem

- You want to make sure developers have only **limited freedom** when implementing those aspects of the code that are not generated.
 - > well structured system
 - > keeps the promises made by the architecture
- An important challenge is thus: How do we combine **generated** code and **manually written** code in a controlled manner (and without using protected regions)?
- **Solution:** Patterns, Recipe Framework

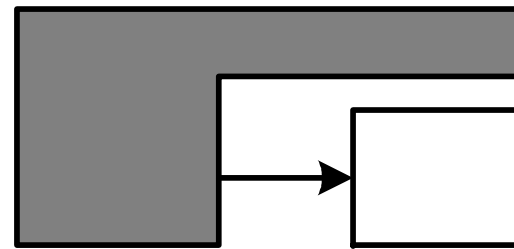


Integration Patterns

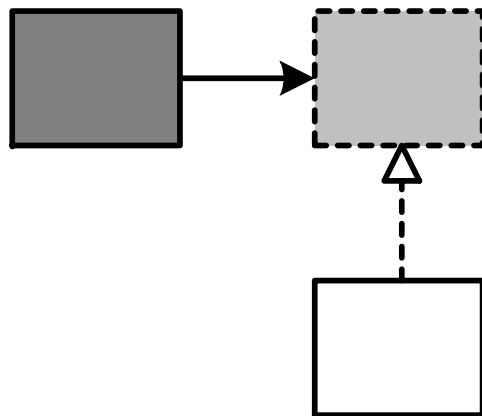
- There are various ways of integrating generated code with non-generated code



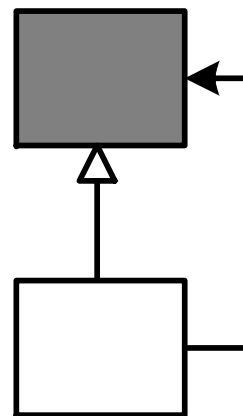
a)



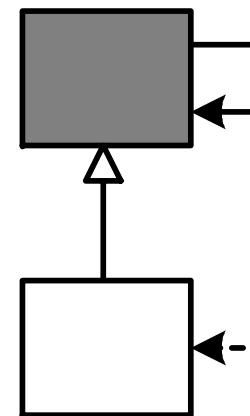
b)



c)



d)



e)

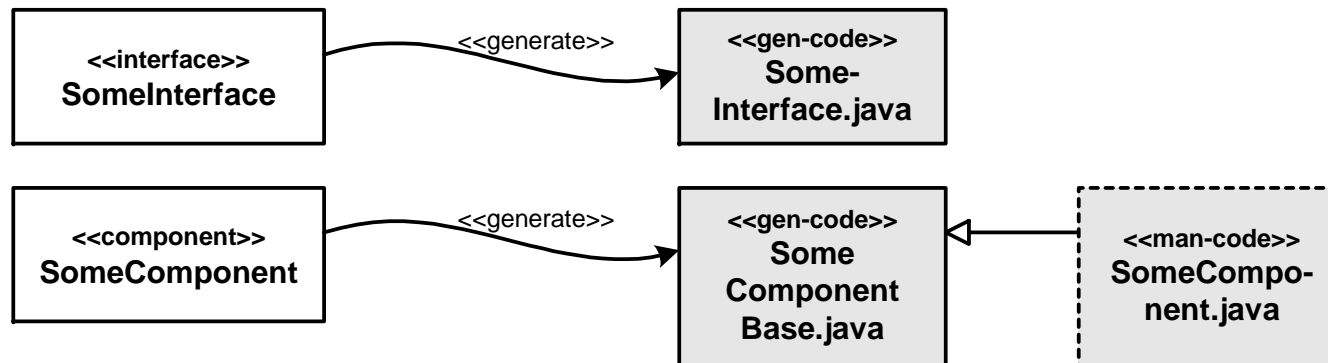
generated code

non-generated code



Component Implementation

- We have not yet talked about the **implementation code** that needs to go along with components.
 - As a default, you will provide the implementation by a **manually written subclass**

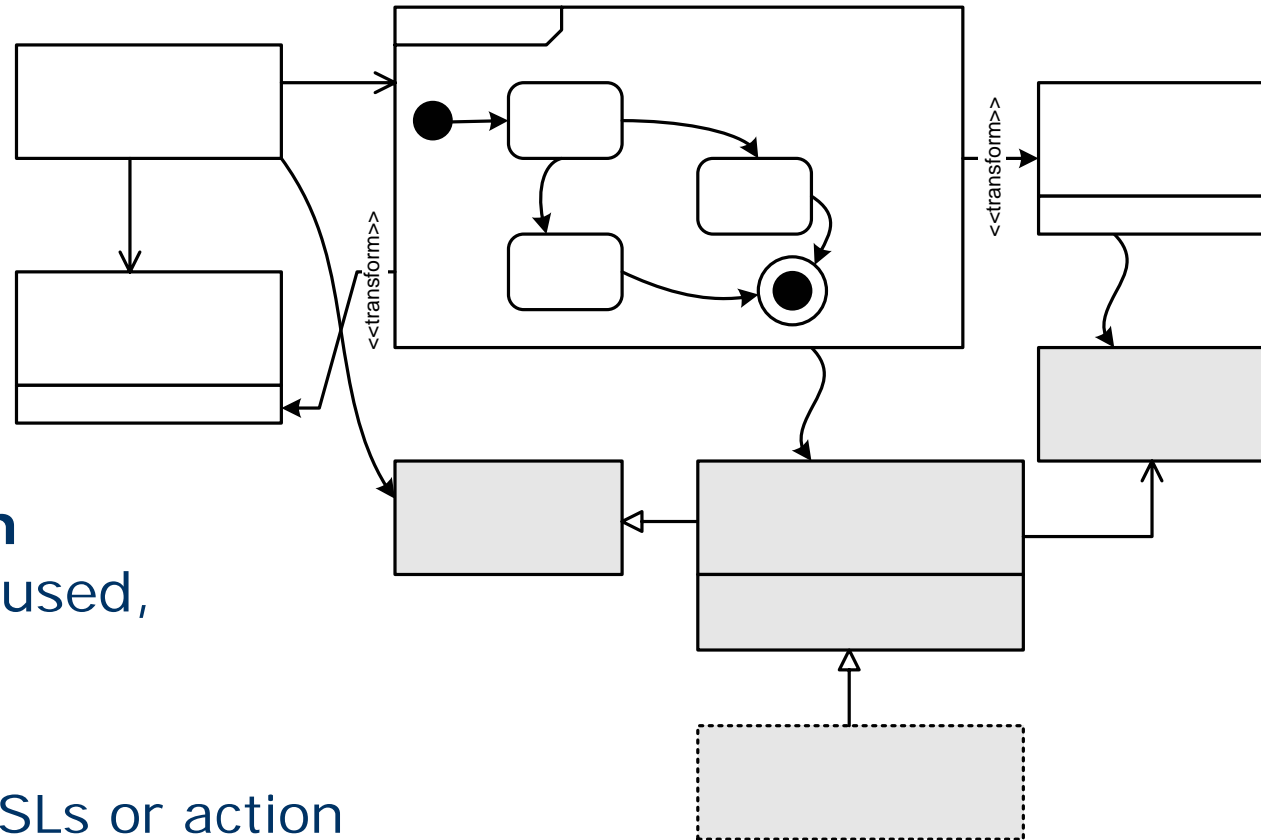


- However, for **special kinds of components** (“component kind” will be defined later) can use different implementation strategies -> **Cascading!**



Component Implementation II

- Remember the **example of the process components** from before:
- Various other **implementation strategies** can be used, such as:
 - Rule-Engines
 - “Procedural” DSLs or action semantics
- Note that, here, **interpreters** can often be used sensibly instead of generating code!





Recipes I



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

The screenshot shows the Eclipse IDE with the following components:

- Package Explorer:** Shows a project structure with packages like 'model', 'workflow', and 'src-gen'. Under 'src-gen', there is a package 'de.jax' containing several generated classes like 'AbstractCdPlayer.java'.
- Editor:** Displays the source code for 'CdPlayer.java', which is currently empty except for the package declaration 'package de.jax;'.
- Problems View:** Shows an error message: "your implementation class has to extend the generated base class de.jax.AbstractCdPlayer".
- Recipes View:** A table showing details for a selected recipe.

Callouts provide additional context:

- "Recipes can be arranged hierarchically" points to the Package Explorer.
- "This is a failed check" points to the error message in the Problems view.
- "Green ones can also be hidden" points to a green checkmark icon in the Problems view.
- "Here you can see additional information about the selected recipe" points to the Recipes view table.

Name	Value
_type	org.openarchitectureware.recipe.edi...
_type	org.openarchitectureware.recipe.uti...
className	de.jax.CdPlayer
element	org.eclipse.emf.ecore.impl.EObjectI...
projectName	4_demo.gmf.statemachine2.exe...
supertypeName	de.jax.AbstractCdPlayer



Recipes II



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

Adding the extends clause makes all of them green

Name	Value
_type	org.openarchitectureware.recipe.ecl...
_type	org.openarchitectureware.recipe.uti...
className	de.jax.CdPlayer
element	org.eclipse.emf.ecore.impl.EObjectI...
projectName	oaw4.demo.gmf.statemachine2.exa...
supertypeName	de.jax.AbstractCdPlayer



Recipes III



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

Description	Resource	Path	Location
✘ The type CdPlayer must implement the inherited abstract method CdPlayerActions.checkCD()	CdPlayer.java	oaw4.demo.gmf.statemachi...	line 3
✘ The type CdPlayer must implement the inherited abstract method CdPlayerActions.closeTray()	CdPlayer.java	oaw4.demo.gmf.statemachi...	line 3
✘ The type CdPlayer must implement the inherited abstract method CdPlayerActions.openTray()	CdPlayer.java	oaw4.demo.gmf.statemachi...	line 3
✘ The type CdPlayer must implement the inherited abstract method CdPlayerActions.pausePlaying()	CdPlayer.java	oaw4.demo.gmf.statemachi...	line 3
✘ The type CdPlayer must implement the inherited abstract method CdPlayerActions.shutdown()	CdPlayer.java	oaw4.demo.gmf.statemachi...	line 3
✘ The type CdPlayer must implement the inherited abstract method CdPlayerActions.startPlaying()	CdPlayer.java	oaw4.demo.gmf.statemachi...	line 3
✘ The type CdPlayer must implement the inherited abstract method CdPlayerActions.stopPlaying()	CdPlayer.java	oaw4.demo.gmf.statemachi...	line 3

- I finally implement them sensibly, & everything is ok.
- The Recipe Framework & 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.



Model Verification

Model Verification is an additional way of „testing“ a system, on a very „semantical“ level



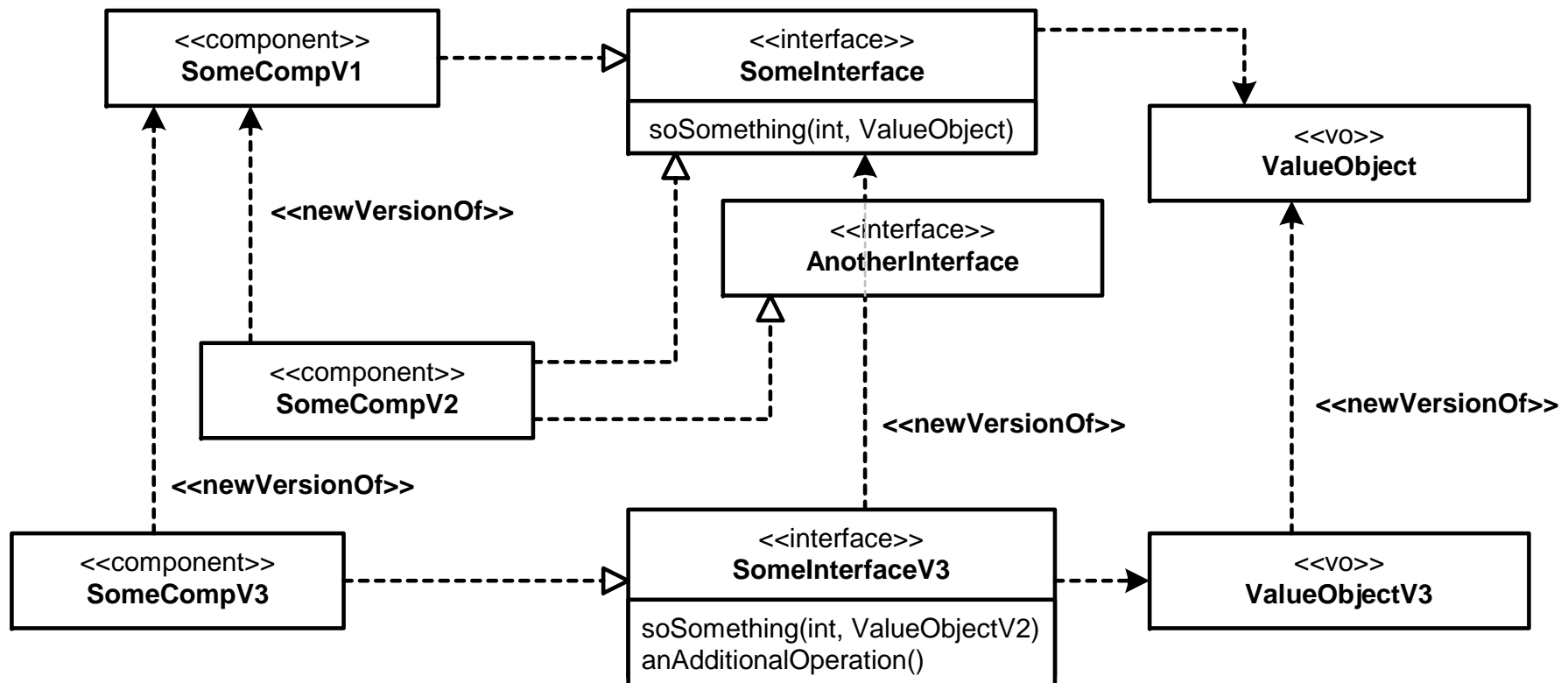
Additional Tests: Model Verification

- In many cases it is possible to **detect design errors already in the models**. This step is called **model verification**.
- The most „extreme“ form is to **interpret and simulate the whole model**; this is however, not simple to achieve, although there are „UML VMs“.
- However, it is easily possible to **verify design constraints** in the model **before** model transformation or code generation steps are done.



Additional Tests: Model Verification

- A really important aspect in our example system is **evolution of interfaces**:





Additional Tests: Model Verification



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

```

import statemachine2;

context StateMachine ERROR "States must have unique Names" :
  states.typeSelect(State).forall(s1| !states.typeSelect(State).
    exists(s2| (s1 != s2) && (s1.name == s2.name) ));

context Named if !Transition.isInstance(this) ERROR this.metaType.name+" must be named":
  this.name != null;

context StartState ERROR "no incoming transitions allowed":
  this.inTransitions.size == 0;

context StartState ERROR "start state must have one out transition":
  this.outTransitions.size == 1;
  
```

For which elements is the constraint is applicable

ERROR or WARNING

Constraint Expression

Error message in case Expression is false

- Note the **code completion & error highlighting** 😊

```

unexpected token: n if !Transition.isInstance(this) ERROR this.metaType.n |ame+"
  this.name != null;

context StartState ERROR "no incoming transitions allowed":
  this.inTransitions.size == 0;

context S
  this.
  eAllContents Set - EObject
  eContainer EObject - EObject
  eContents List - EObject
  eRootContainer EObject - EObject
  outTransitions List - AbstractState

context S
  this.
  allowed":
  allowed":
  
```

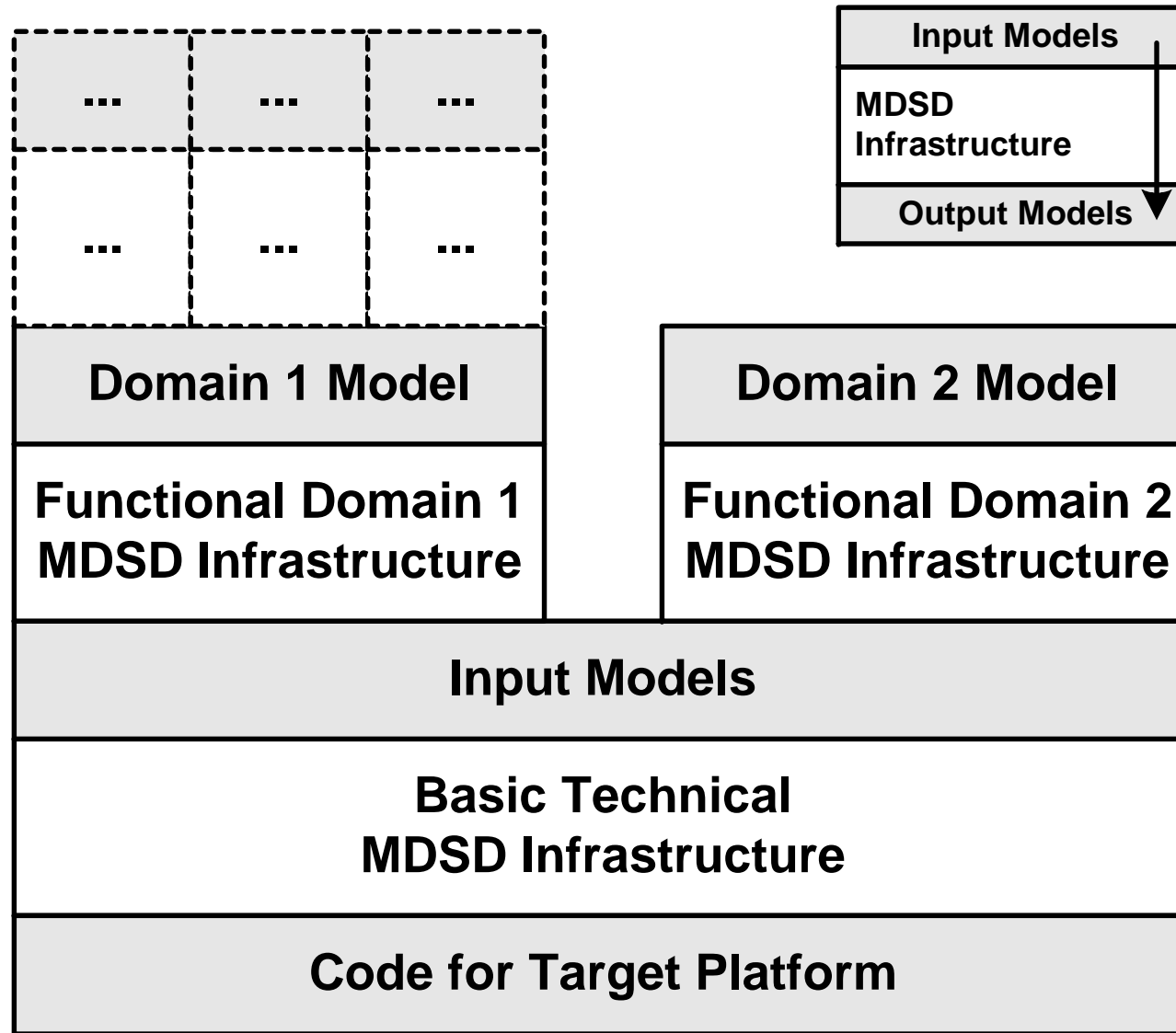


Partitions/Layers/Cascading

Architecture can be nicely layered and architected to be as small an consistent as possible

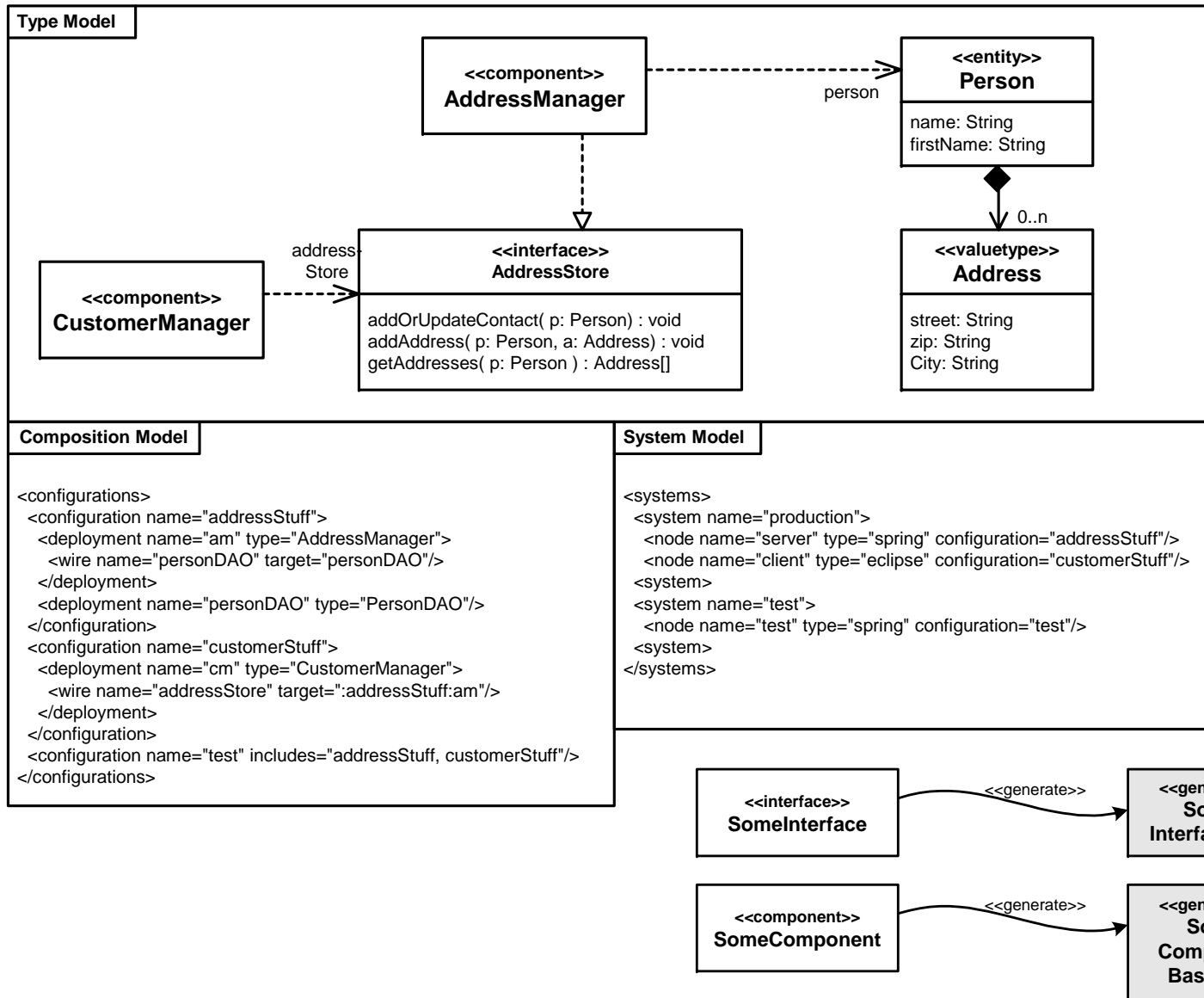


Levels of MDSD



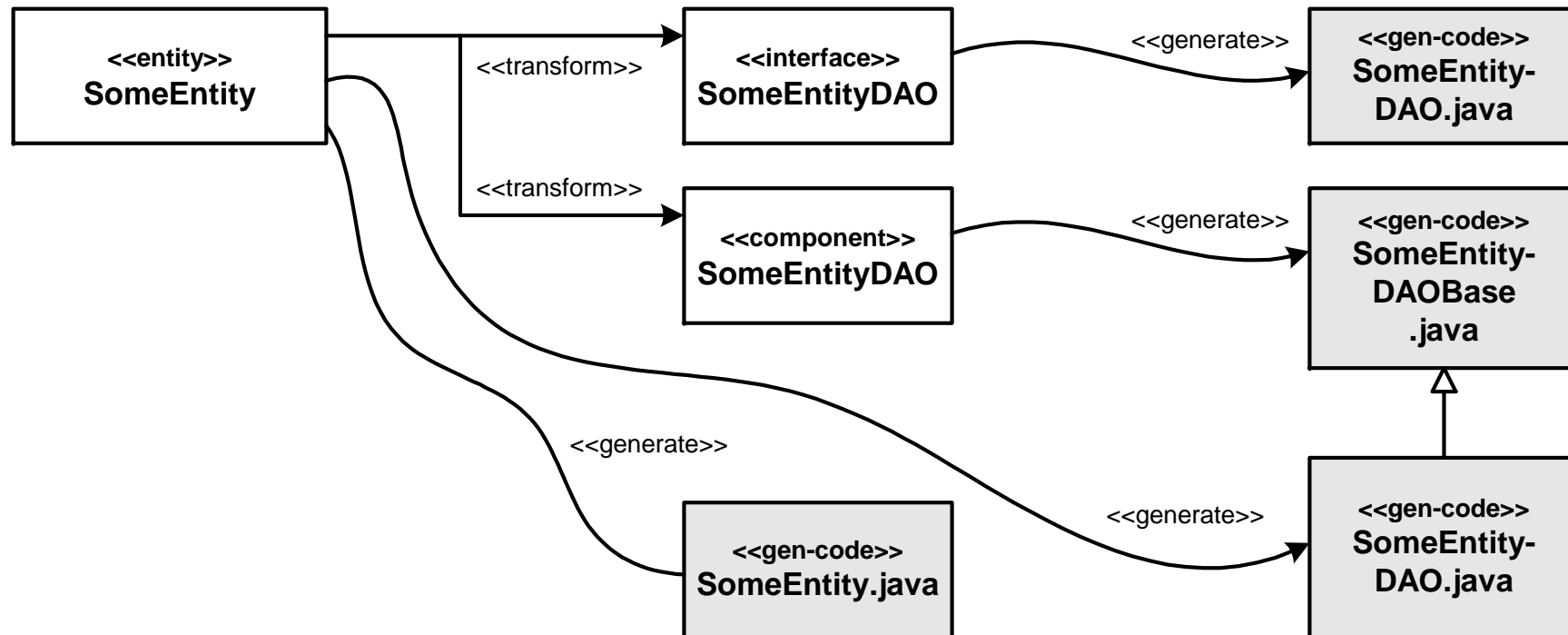


Levels of MDSD III – M2M Transformations



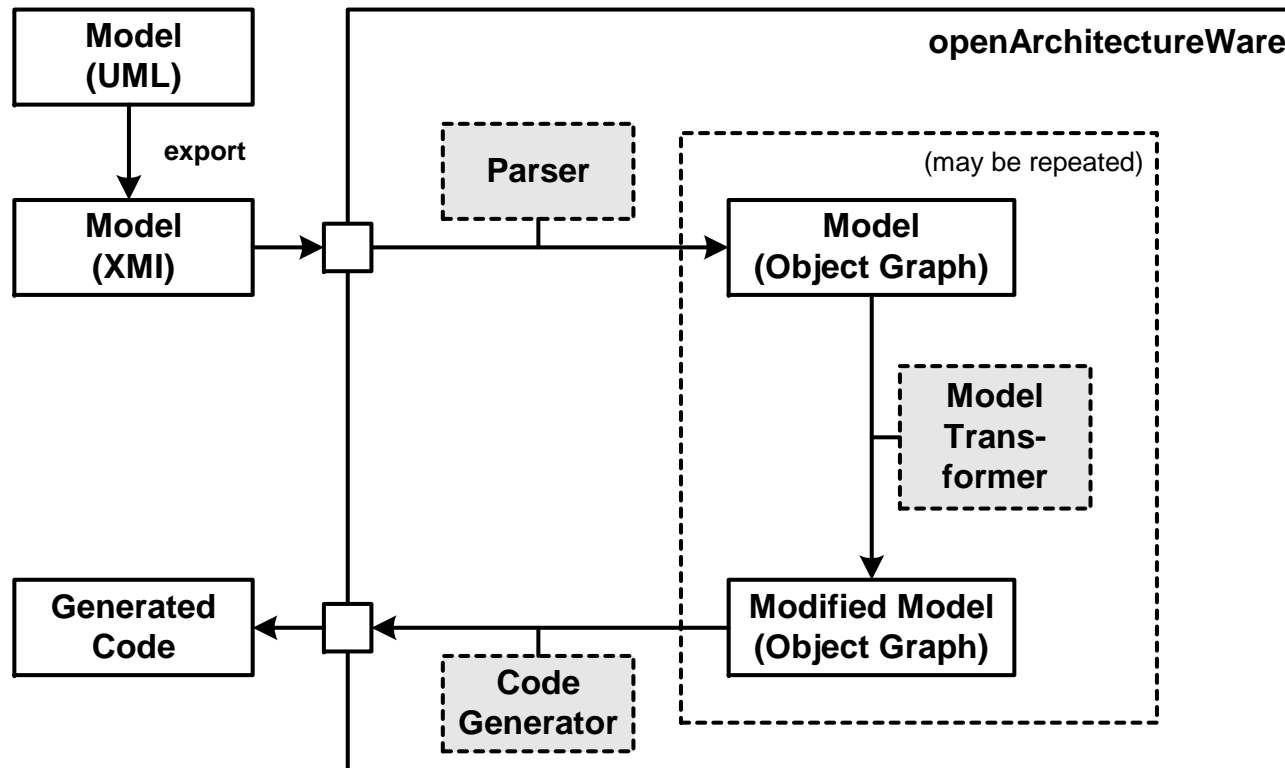


Levels of MDSD III – M2M Transformations II





Levels of MDSD III – M2M Transformations III





M2M Transformations



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

```

AddEmergencyShutdown.ext x
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.



Thanks!

Please ask questions!



Some advertisement ☺

- For those, who speak (or rather, read) german:

Völter, Stahl:

Modellgetriebene Softwareentwicklung

Technik, Engineering, Management

dPunkt, 2005

www.mdspd-buch.de



- An **very much updated** translation is under way:

Model-Driven Software Development,

Wiley, Q2 2006

www.mdspd-book.org

