Best Practices for Model-Driven Development

OOPSLA 2007 Tutorial

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

www.mdsd-buch.de

www.mdsd-book.org

About me

- Independent Consultant
- Based out of Göppingen, Germany
- Focus on
  - Model-Driven Software Development
  - Software Architecture
  - Product Lines

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

CONTENTS

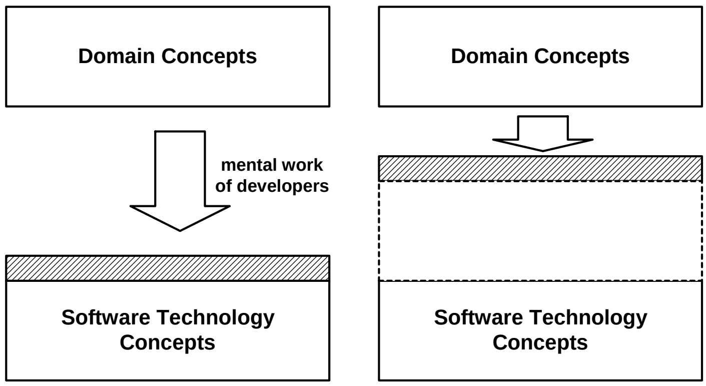
- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don't Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

CONTENTS

- **What is MDSD?**
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don't Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

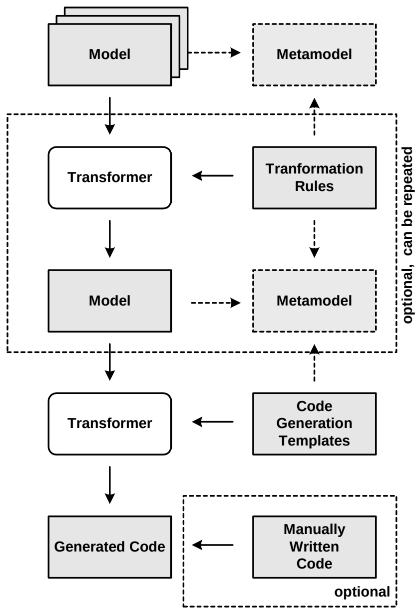
Model-Driven Software Development

- Model-Driven Software 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**.



How MDSD works

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



Reasons for MDSD

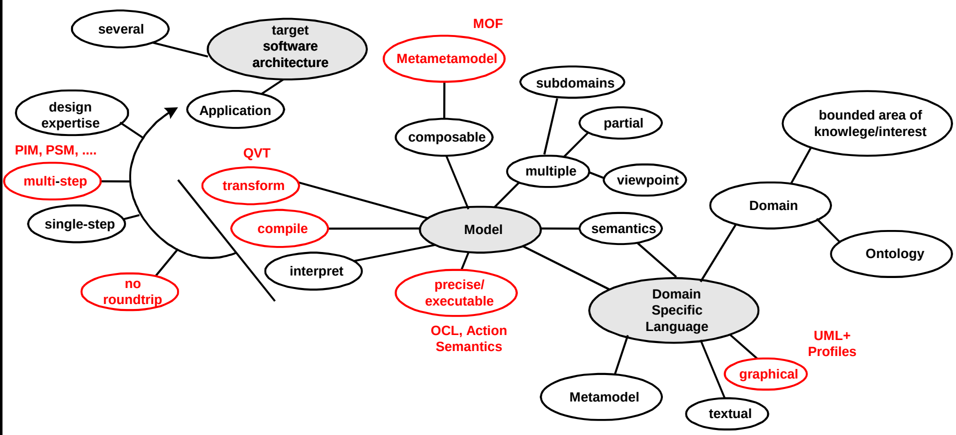
- Software Development is too **complex** and too **expensive** (now, this is a really new finding ☺) ...

  ... because:
  - There is **too little reuse**
  - **Technology changes** faster than developers can learn
  - Knowledge and practices are **hardly captured explicitly** and made available for reuse
  - Domain experts cannot understand all the **technology stuff** involved in software development

- MDSD aims at attacking some of these problems. We shall see how on the following slides.

MDSD Best Practices

MDSD Core Concepts and MDA



MDSD Core Values

- We prefer to validate **software-under-construction** over validating software requirements
- We work with **domain-specific assets**, which can be anything from models, components, frameworks, generators, to languages and techniques.
- We strive to **automate software construction** from domain models; therefore we consciously distinguish between building software factories and building software applications
- We support the **emergence of supply chains for software development**, which implies domain-specific specialization and enables mass customization

Other related approaches

- Microsoft's Software Factories:
  Focus on Reuse, Efficient Development, DSLs
- Domain-Specific (Visual) Modelling:
  Focus on (Visual) DSLs
- Generative Programming:
  Focus on Efficiency, "Automatic Manufactoring", Software System Families
- Language-Oriented Programming:
  Focus on DSLs instead of Frameworks, incl. Editor/Debugger Support

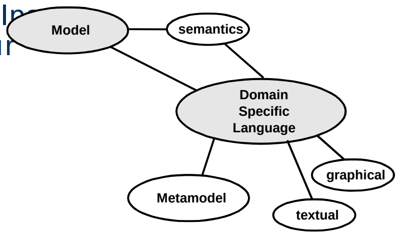
→ **all basically the same** ☺

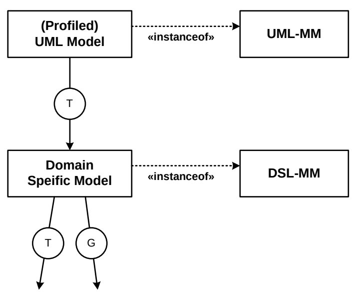
CONTENTS

- What is MDSD?
- **Custom Metamodel**
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don‘t Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

Custom Metamodel

- Building your own meta model helps you **understand and clarify** your domain‘s concepts.
- A **customized** general-purpose meta-model will always contain a lot of unnecessary complexity. (think UML Profile)
- If you use a general purpose language (such as UML) on which to build your DSL, **consider it concrete syntax!**
- You should still have a domain-specific metamodel the first step must be a **transformation** from the GP language to the custom metamodel.





Custom Metamodel II

- Why is this important? Basically, because the GP metamodel is typically **very complicated** (UML ☺)
  - Constraint checking can be more specific in a DS metamodel
  - Model modifications are much easier (try to **write** to the UML metamodel!)
  - Subsequent transformation/code generation is also much simple
  - And you are able to easily **change the concrete syntax** to something more appropriate without the need to change your backend

CONTENTS

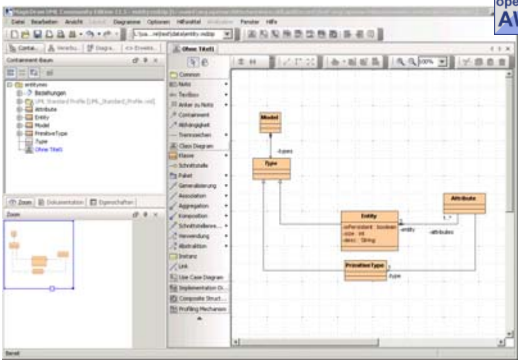
- What is MDSD?
- Custom Metamodel
- **Take care of your Metamodel**
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don't Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

Use a suitable editor

- The meta model is the **central asset** that defines the semantics of your domain and your DSL(s).
- Make sure it is described using a **scalable means**, such as a textual DSL or a UML tool
  - The EMF tree editors don‘t scale!
  - The Ecore Editor provided with GMF also does not really scale...

Using UML to define meta models

- One approach is to use a UML tool (one which supports Eclipse UML2 export) and **transform** the model into an Ecore meta model.
- An alternative is to use a **suitable textual notation** (make sure you can distribute the model over several files...!)



oAW **uml2ecore**

- Ecore File
- Name Management (qualified, namespaces)
- Various constraints

How do I come up with a good metamodel?

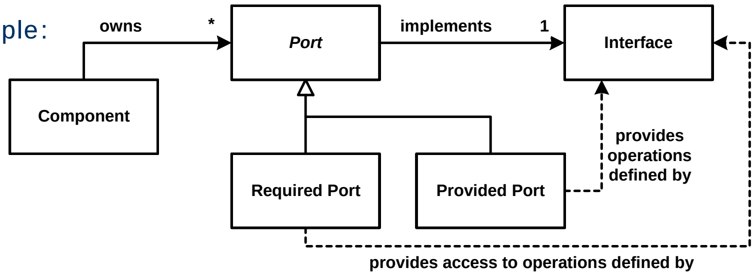
- **Incrementally!**
- Based on **experience** from previous projects, and by „mining“ domain experts.
- A very good idea is to start with a (typically) very well known domain: the **target software architecture** (platform) → Architecture-Centric MDSD

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:



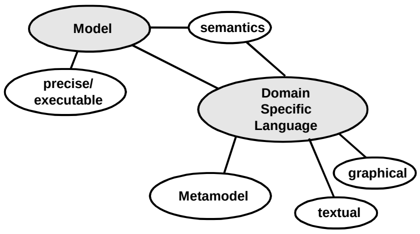
  - A component owns any number of ports.
  - Each port implements exactly one interface.
  - There are two kinds of ports: required ports and provided ports.
  - A provided port provides the operations defined by its interface.
  - A required port provides access to operations defined by its interface.

CONTENTS

- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- **Checks First & Separate**
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don't Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

Define constraints in separate artefacts

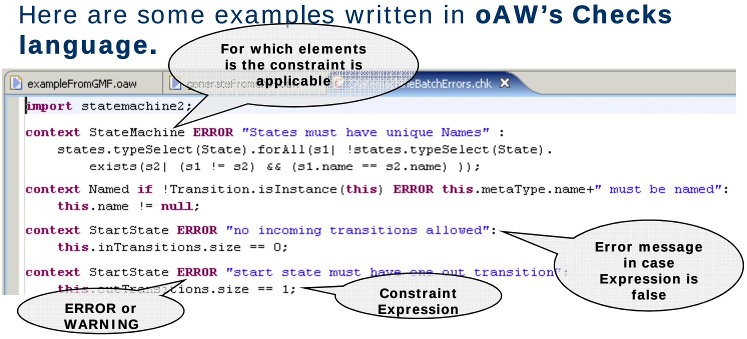
- There's no point in **transforming a „buggy" model** into something else.
- A buggy model is a model where the **constraints** defined as part of the metamodel **do not hold**.



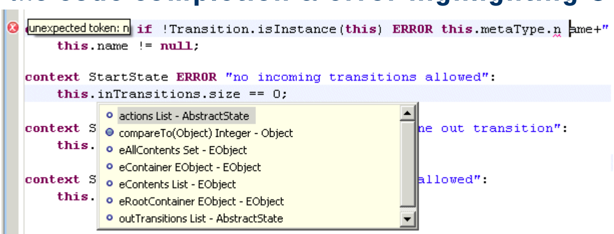
- Make sure you have such constraints!
- Make sure they are **not part of the transformation**:
  - Would make transformation more complicated
  - If you have several transformations from the same model, you'd need to implement the checks several times
- Make constraint checking a **separate, and early** step in the transformation workflow

Using oAW's Check language to define constraints

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

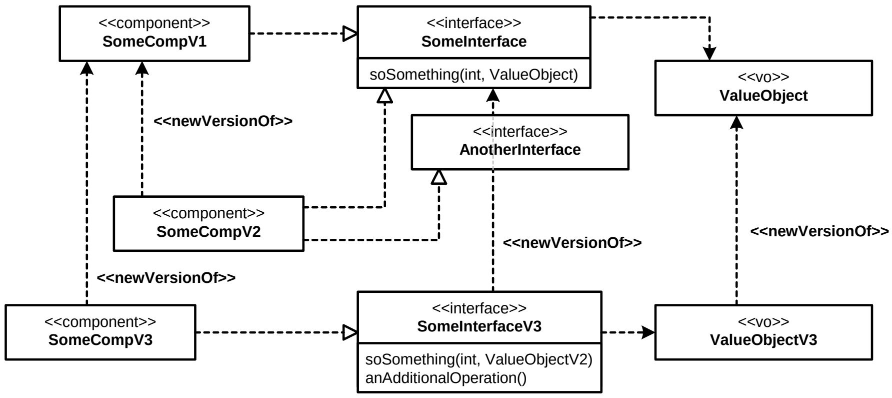


- Note the **code completion** & **error highlighting** ☺



Constraints can handle non-trivial things

- More complex constraints: Versioning and Evolution

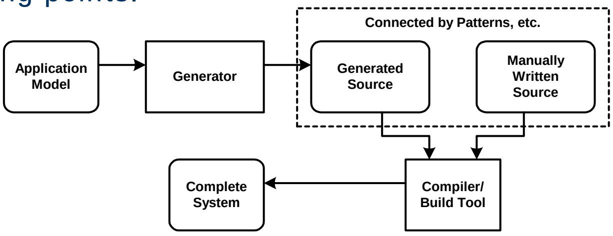


CONTENTS

- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- **Care about Generated Code**
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don't Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

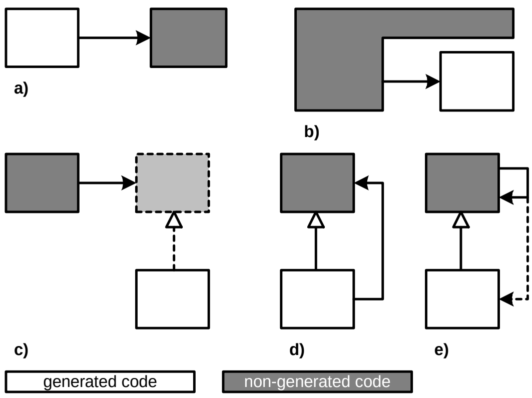
Separate Generated and Non-Generated Code

- Keep generated and non-generated code in **separate files.**
- **Never modify generated code.**
- Design an architecture that clearly defines **which artifacts are generated, and which are not.**
- Use **suitable design approaches** to “join” generated and non-generated code. Interfaces as well as design patterns such as factory, strategy, bridge, or template method are good starting points.



Code Integration using Patterns and Idioms

- A) Generated code can **call** non-generated code contained in libraries
- B) A non-generated framework can **call** generated parts.
- C) **Factories** can be used to „plug-in“ the generated building blocks
- D) Generated classes can also **subclass** non-generated classes.
- E) The base class can also contain abstract methods that it calls, they are implemented by the generated subclasses (***template method*** pattern)



Produce Nice-Looking Code ... whenever possible!

- PRODUCE NICE-LOOKING CODE ... WHEREVER POSSIBLE!
- When designing your code generation templates, also **keep the developer in mind** who has to – at least to some extent – work with the generated code, for example
  - When verifying the generator
  - Or debugging the generated code
- Using this pattern helps to **gain acceptance** for code generation in general.
- Examples:
  - Comments
  - Use pretty printers/code formatters
  - Location string („generated from model::xyz“)

Believe in Re-Incarnation

- The final, implemented application should be built by a build process that includes **re-generation of all generated/transformed parts**.
  - ...which includes more than just code – see LEVERAGE THE MODEL
- **As soon as there is one manual step**, or one line of code that needs to be changed after generation, then sooner or later (sooner is the rule) the generator will be abandoned, and the code will become business-as-usual.
- Note that this pattern **does not receommend to generate as much stuff as possible.**
  - You should use a rich domain specific platform,
  - And use existing frameworks and platform where possible

Leverage the Model

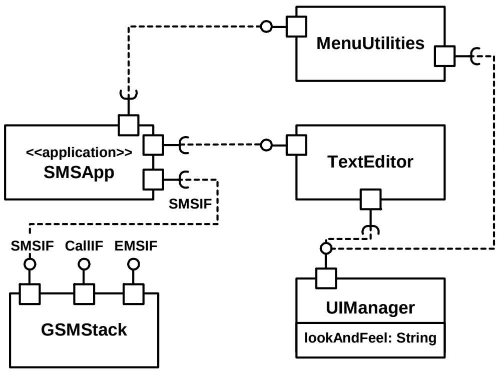
- The information captured in a model should be **leveraged to avoid duplication** and to minimize manual tasks.
- Hence you may **generate much more than code:**
  - build scripts
  - packaging and deployment files
  - infrastructure configuration files
  - test data and UIs
  - ...
- Find the right balance between the **effort required for automating manual tasks** and the effort of **repetitively performing manual tasks**

CONTENTS

- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- **Managing the Architecture**
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don't Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

Managing the Architecture

- It is relatively easy check architectural constraints (such as dependencies) **on the level of models**.
- However, if the model analysis tells you that everything is ok (no constraint violations) it must be ensured that the **manually written code does not compromise** the validity of the constraints.
- E.g. how do you ensure that there are no more dependencies in the code than those that are described in the model?



Managing the Architecture II

- The programming model shown below is bad:

```
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

Managing the Architecture III

```
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

Relationship Programming Model / Model

- The programming model must be **true** to the model and the constraints checked therein:
  - If certain constraints on the model hold
  - Then the programming model must ensure that these constraints can't be violated in the "real" code
- Example:
  - constraints, saythere are no illegal dependencies in the model…
  - The programming model must then be sure that no illegal dependencies can be created in the manually written code
- If this is not the case, **constraint checks in the model don't help** you much!

Relationship Programming Model / Model II

- **Conformance** of the manually written code to guidelines implied by the generator (and thus, by the constraints) can be checked by using
  - **compiler tricks** such as static if-false blocks that cast types around or “call” methods

```
public class SCMComponentBase ... {
  static {
    if ( false ) {
      SCMComponentBase i = (SCMComponentBase)
           (new SCMBusinessComponent());
    }
  }
}
```

  - **subsequent checks** check the manually written code for consistency with the guidelines/programming model
    → **Active Programming Model**, Recipe Framework

CONTENTS

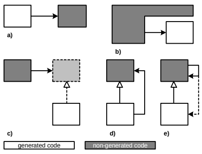
- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- **Active Programming Model**
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don‘t Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

Guiding developers beyond the generator run

- 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 models

- 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 and the **Recipe Framework**

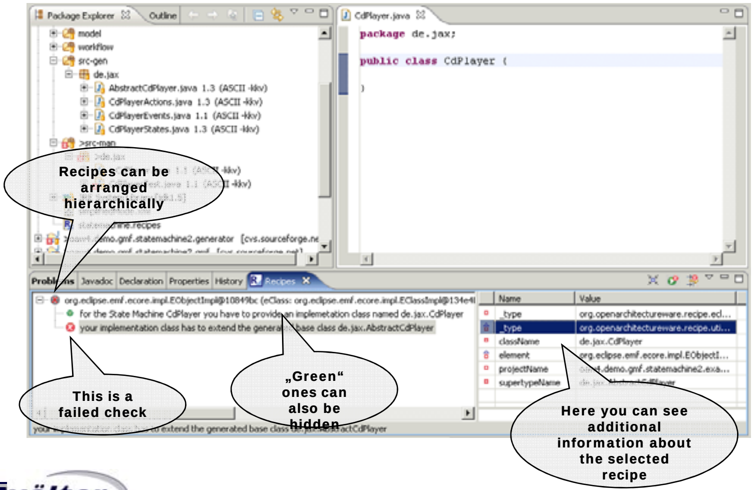


Relationship Programming Model / Model III

- The **openArchitectureWare RecipeFramework** can be used to subsequently check manually written code
  - During the generator run, we generate the generated code;
  - in addition, based on the model, **we instantiate checks** that need to be verified later on the manually-written code
  - In the IDE, the **failed checks are shown** to the user hinting at “problems” with the manualy code that need to be fixed.
  - Once a problem is fixed, the complaint goes away.
  - For many failed checks, a “fix this” button can be activated to **fix the problem automatically**.
- A fairly small number of such Checks can get you a long way...

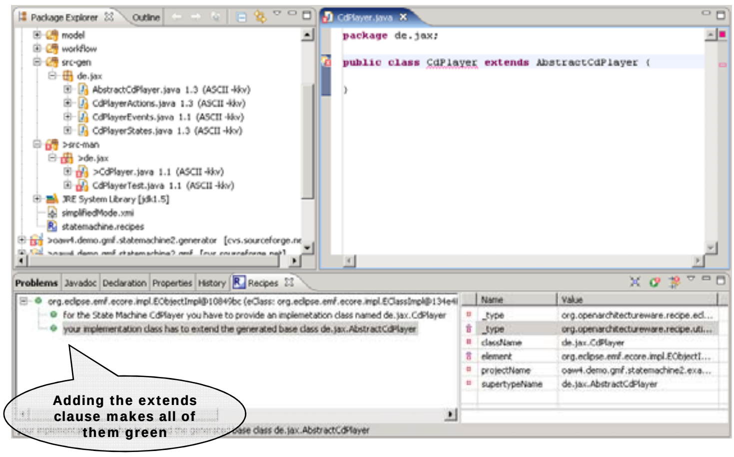
Recipe Framework

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



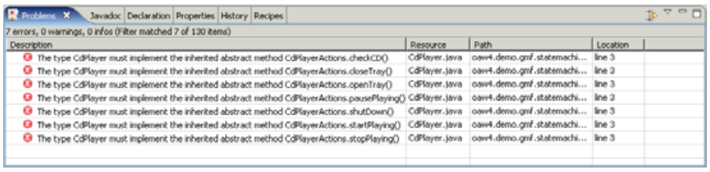
Recipe Framework II

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



Recipe Framework III

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



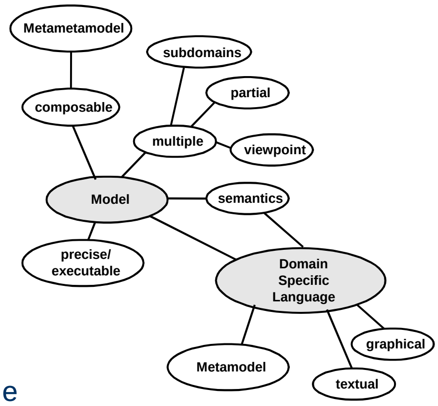
- I finally implement them sensibly, & everything is ok.
- The Recipe Framework & the Compiler have **guided me through the manual implementation steps.**
  - If I didn't like the compiler errors, we could also add recipe tasks for the individual operations.
  - oAW comes with a number of **predefined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

CONTENTS

- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- **Multiple Viewpoints**
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don't Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

Multiple Viewpoints

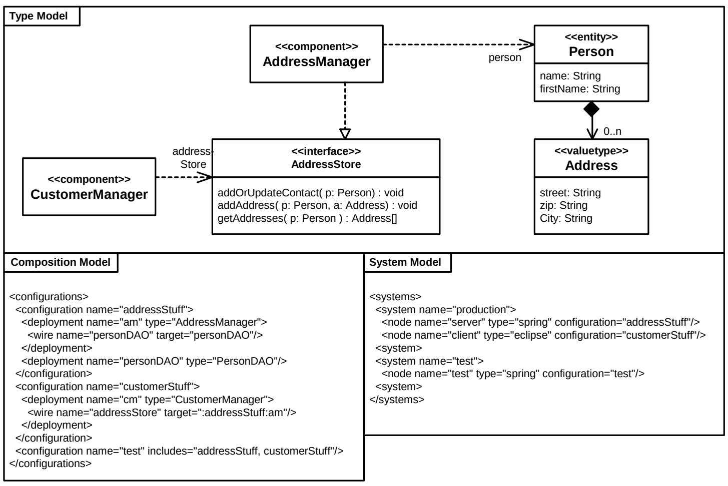
- Complex Systems typically consist of **several aspects, concerns or viewpoints**.
- Often (though not always) these are described by different people at different times in the development process.
- In most cases**, different** forms of **concrete syntax** are suitable for these different viewpoints.



- Therefore, provide **separate models** for each of these viewpoints.

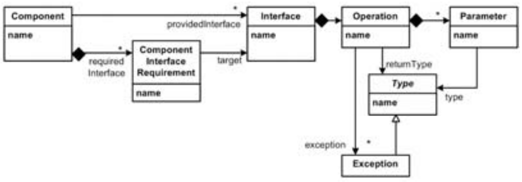
Viewpoints: Component-Based Development Example

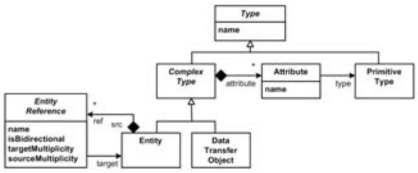
- **Type Model**: Components, Interfaces, Data Types
- **Composition Model**: Instances, “Wirings”
- **System Model**: Nodes, Channels, Deployments



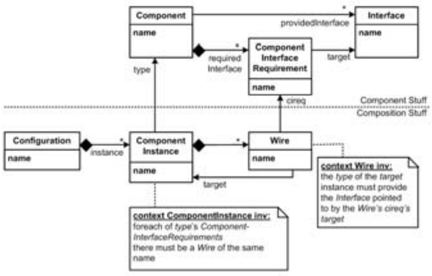
CBD Meta Models for the three viewpoints

Types

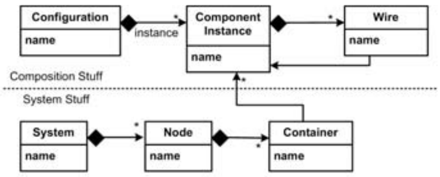




Composition

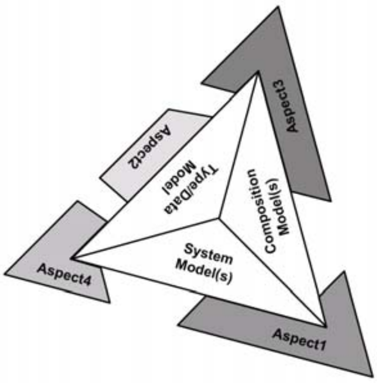


Deployment



Additional Viewpoints: 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

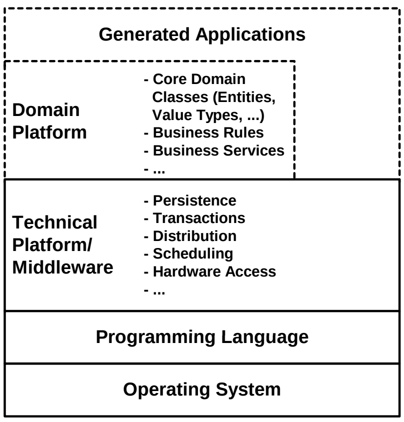


CONTENTS

- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- **Rich Platform**
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don‘t Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

Rich Domain-Specific Platform

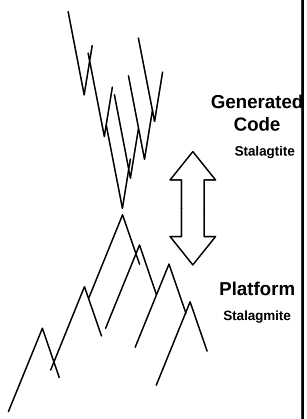
- Define a **rich domain-specific** application platform consisting of
  - Libraries
  - Frameworks
  - base classes
  - interpreters, etc.
- The transformations will “generate code” **for this domain-specific application platform**.
- As a consequence, the trans-formations **become simpler**.



- DSLs and Frameworks are two sides of the same coin

Code Generation vs. Platform

- There is no point in generating 100% of an application's code. You might want to generate 100% for a certain part/aspect, but other code will always be **reused from a platform**.
- The ratio of generated code and platform code varies
  - From system to system
  - And also in one system over time
  - If the **platform gets too complicated** or too slow, generate more code.
  - If the **generator gets too complicated or generates lots of identical code**, move it to the platform

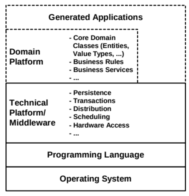


- Generated code is often **framework completion code** – DSLs make frameworks easier to use!

CONTENTS

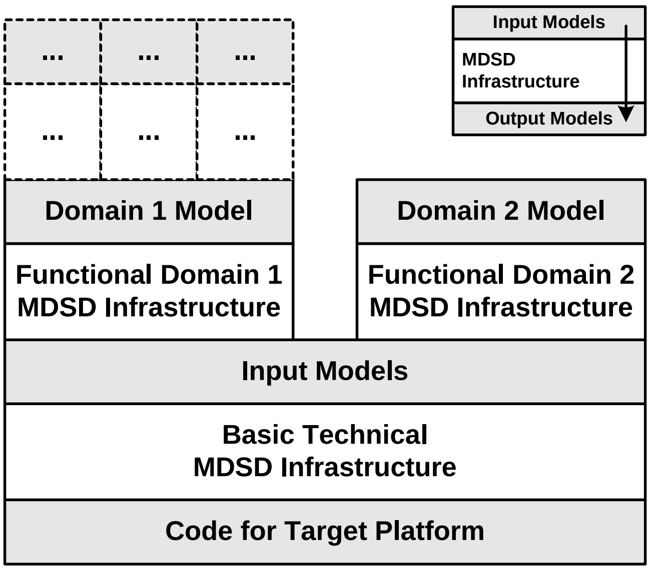
- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- **Cascading MDSD**
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don't Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

Architecture First

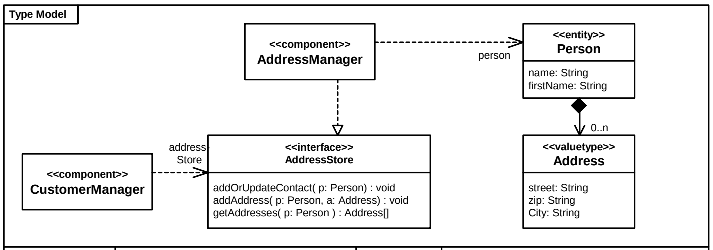


- A successful system is built based on a **well-defined architecture**, often along the lines of the illustration on the right.
- Various parts/layers of this stack can be generated, or developed with metamodel and generator support.
- It has proven useful to **start with the lower layers** to lay a stable foundation:
  - Often, the software architecture is **better understood** than the application logic (by the developers)
  - The architecture is **fairly general** and can be **reused** in many projects
  - More specific layers can be **cascaded on top** of that using model-to-model transformations

Partitions / Layers / Cascading



Example for Cascading I

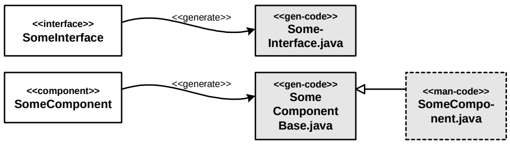


Composition Model

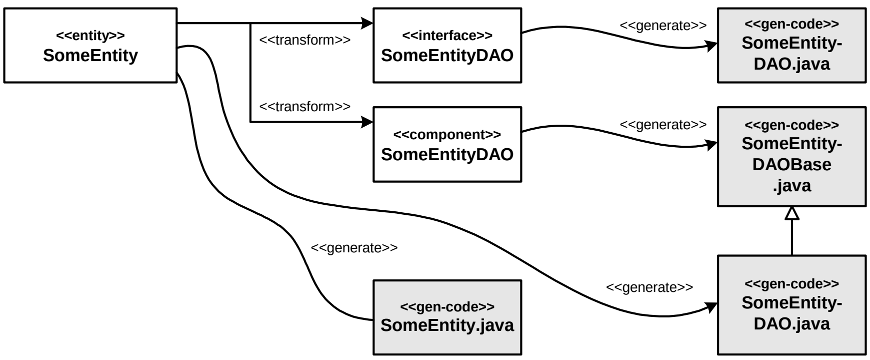
```
<configurations>
  <configuration name="addressStuff">
    <deployment name="am" type="AddressManager">
      <wire name="personDAO" target="personDAO"/>
    </deployment>
    <deployment name="personDAO" type="PersonDAO"/>
  </configuration>
  <configuration name="customerStuff">
    <deployment name="cm" type="CustomerManager">
      <wire name="addressStore" target=":addressStuff:am"/>
    </deployment>
  </configuration>
  <configuration name="test" includes="addressStuff, customerStuff"/>
</configurations>
```

System Model

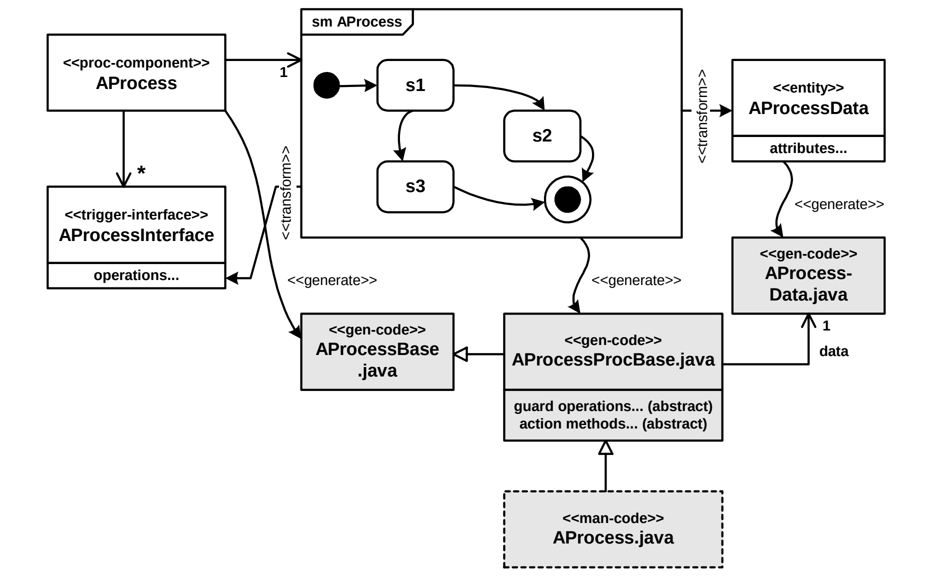
```
<systems>
  <system name="production">
    <node name="server" type="spring" configuration="addressStuff"/>
    <node name="client" type="eclipse" configuration="customerStuff"/>
  <system>
  <system name="test">
    <node name="test" type="spring" configuration="test"/>
  <system>
</systems>
```



Example for Cascading II



Example for Cascading III



CONTENTS

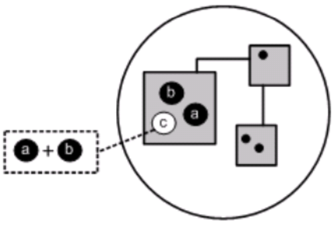
- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- **Extendible (Meta)model**
- Graphical vs. Textual Syntax
- Don‘t Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

Extendible Metamodel

- Assume you want to **generate code for Java** from a given model. You‘ll need all kinds of **additional properties** on your model elements, such as:
  - Class::javaClassName
  - Class::package
  - Class::fileName
- If you add these to your domain metamodel, you‘ll **pollute the metamodel** with target platform-specific properties.
- This gets even worse if you generate for **several targets** from the same model...
- Therefore allow **metaclasses to be annotated** with additional (derived) properties **externally**.
  - Somewhat like open classes/AOP/C#3.0 extension methods

Extension Functions

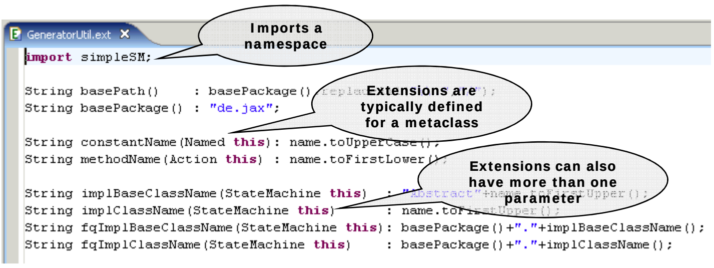
- Define a **set of functions** that calculate derived properties.
  - Depending on the tooling, they can be **accessed as if they were properties.**
  - Defined in a separate file, the original meta model does not need to be changed.



- Disadvantage: Since the extensions are functions, **you cannot store additional information** with the model; you can only calculate derived values from information already in the model.
- **Tooling:** using oAW’s Xtend facility you can access the “derived properties”, i.e. the functions almost as if they were regular properties: you have to use () after the name

Extendible Metamodel II

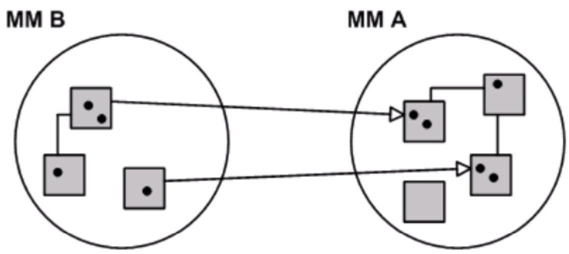
- One can **add behaviour to existing metaclasses** using oAW's **Xtend** language.



- 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

Specialization

- Create a **new meta model**, **extending** classes defined in some other (base) meta model.
  - Useful to **specialize a complete language** and work with that new language in your system.
  - A typical candidate for extension is the UML meta model.



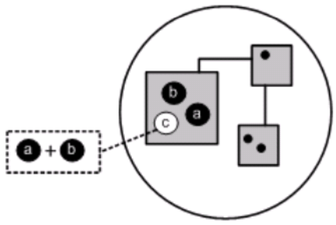
- Disadvantage: you **cannot remove items** you do not need in your language from the base meta model.
  - This is an especially serious problem with complex base meta models such as UML.

Specialization II

- Tooling:
  - Ecore **does not provide** a means to have one meta model package "**extend**" another one. You can only extend meta classes.
  - This means you have to define a new meta model package, and **reference meta classes in another one** to have your new classes extend the original ones.
  - Your meta classes will use the **new package's name for qualification**. The old meta classes (those "inherited" from the original meta model) will still be available under in the old package.
  - Thus, you have to work with **two meta model packages**. This can be a problem in some tool environments.

Extension Functions

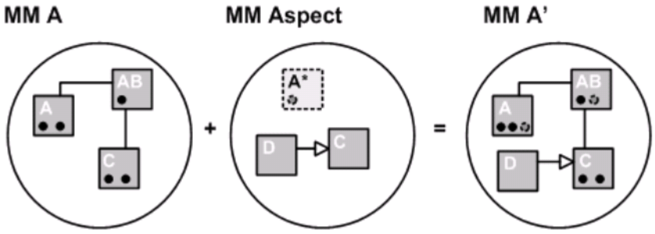
- Define a **set of functions** that calculate derived properties.
  - Depending on the tooling, they can be **accessed as if they were properties.**
  - Defined in a separate file, the original meta model does not need to be changed.



- Disadvantage: Since the extensions are functions, **you cannot store additional information** with the model; you can only calculate derived values from information already in the model.
- **Tooling:** using oAW's Xtend facility you can access the "derived properties", i.e. the functions almost as if they were regular properties: you have to use () after the name

Weaving

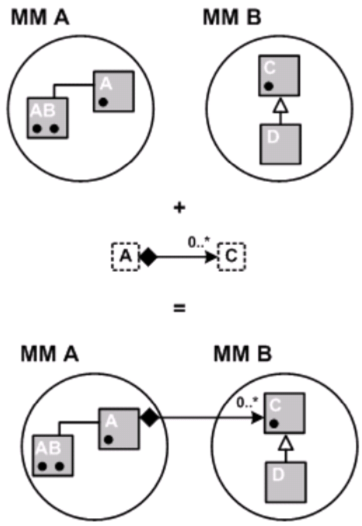
- You use an aspect weaver to **weave additional properties, relationships or meta classes** into the base meta model.
  - Depending on the weaver, you can add new properties, new relationships and also new meta classes.



- **Tooling:** We use oAW's XWeave.
  - The aspect elements are actually physically woven into the original model, physically altering its structure.
  - The result of the weaving process is an updated model.
  - Subsequent tooling cannot tell the difference between a woven model and a "normal" model.

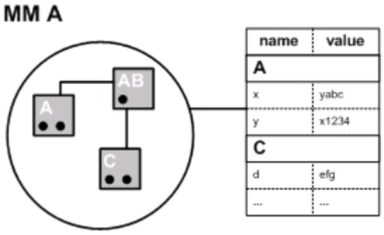
Joining

- You take two or more existing meta models and **add relationships joining them**.
  - The meta models keep their own identities.
  - Subsequent tools must be able to work with several meta models.
  - The two (or more) partial models do not need to know about the other ones.
- **Tooling:** oAW comes with a join facility called XJoin.



Dynamic Properties

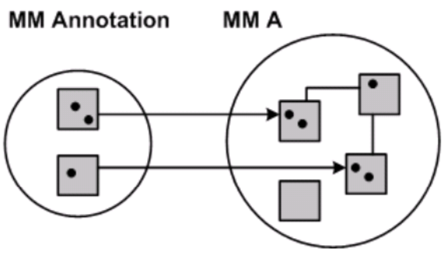
- Associate a set of **name-value pairs** with a meta model element.
  - This allows the storage of all kinds of additional information with model elements.
  - The values can be primitive values or even additional model fragments.



- **Tooling:** oAW provides a library that can store any number of name-value pairs with any model element.
  - The value can be anything, including a model fragment.

Annotation

- **External models** that store additional information about a model element of the original model.
  - In order to establish the relationship with the original model, the annotation meta model either contains a **reference** to the target meta class, or **references the target by some unique (typically qualified) name or ID**.



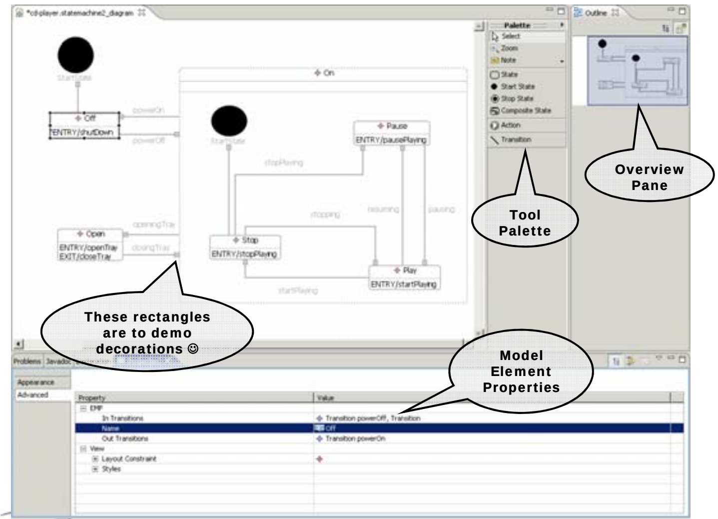
- **Tooling:** In EMF, a model can reference elements in another model by using inter-resource references.

CONTENTS

- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- **Graphical vs. Textual Syntax**
- Don‘t Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

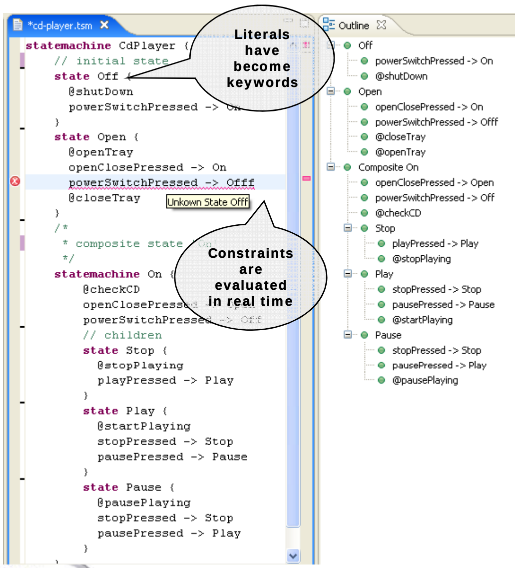
Graphical vs. Textual Syntax

- This is an example of an editor **built with Eclipse GMF**, based on a metamodel for state machines.



Graphical vs. Textual Syntax II

- This is a textual editor for the same metamodel



Graphical vs. Textual Syntax III: Comparison

- **Both kinds** of editors…
  - Can be built on the same meta model
  - Can verify constraints in real time
  - Will write ordinary EMF models
- **Graphical Editors**
  - are good to show structural relationships
- **Textual Editors**
  - are better for „algorithmic“ aspects
  - Integrate better with CVS etc. (diff, merge)

CONTENTS

- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- **Don't Duplicate – Transform!**
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

Motivating Example – State Machines

- Consider you want to generate a **state machine implementation for C++ and Java**:
  - You have a model of a state machine,
  - And you have two sets of templates – one for C++, one for Java
- Assume further, that you want to have an ***emergency stop* feature** in your state machines (a new transition from each ordinary state to a special stop state)
  - You can either add it manually to the model (which is tedious and error prone)
  - Or you can modify the templates (two sets, already...!) and hard-code the additional transitions and state.
- Both solutions are not satisfactory.
- **Better Alternative:** Use a Model-Modification to add these transitions and state automatically

Motivating Example – State Machines II

- The **model modification** shows how to add an dditional state & 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

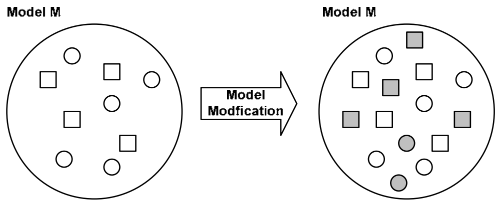
„create extensions“ guarantee that for each set of parameters the *identical* result will be returned.

Therefore createShutDown() will always return the same element.

No code generation templates need not be modified for the new feature to work

M2M: Model Modifications

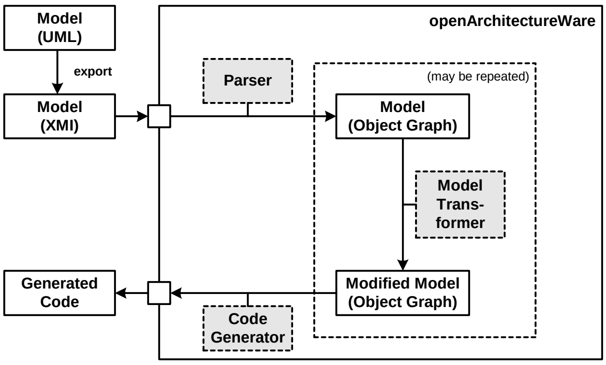
- An existing model is **modified “in place”.**



- Implications of model modification
  - An existing model is **enhanced at generation time**, by adding elements
  - The model is based on the same metamodel before and after the modification
  - Little initial implementation overhead (e.g. using Java code)

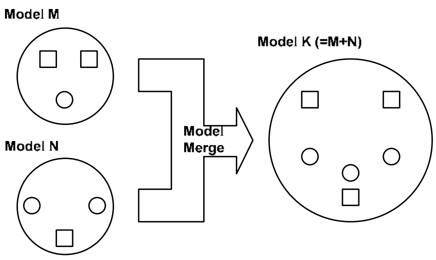
Don‘t Duplicate – Transform!

- M2M Transformations should be kept **inside the tool**, use them to **modularize the transformation** chain.
  - Never ever modify the result of a transformation manually
- Use **example models** and **model-specific constraints** to verify that the transformation works as advertised.



M2M: Model Merging

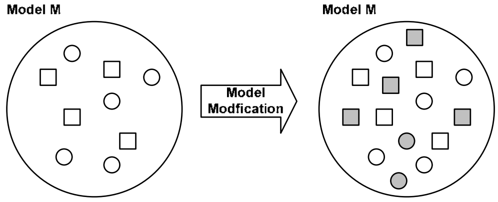
- Several models are **merged** with each other.



- Implications of model merging
  - Typically **easy to implement** (no actual transformation)
  - Meta models are obviously the same
  - Useful if models need to be **modularized** (team issues, performance, ...) and then put together for a complete build

M2M: Model Modifications

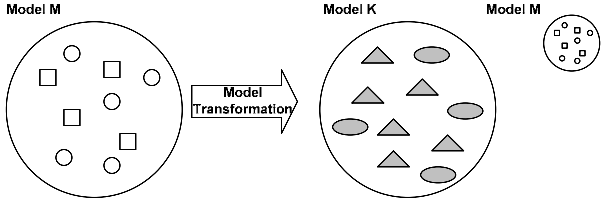
- An existing model is **modified "in place".**



- Implications of model modification
  - An existing model is **enhanced at generation time**, by adding elements
  - The model is based on the same metamodel before and after the modification
  - Little initial implementation overhead (e.g. using Java code)

M2M: Model Transformations

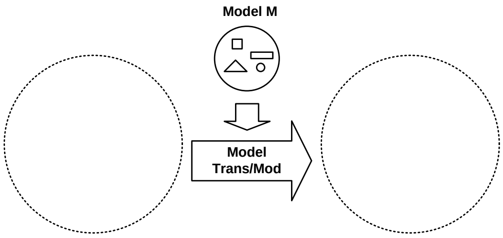
- A model is **transformed into another model**; the input model is left unchanged.



- Implications of model transformations
  - clean separation: **separate models, separate metamodels**
  - different domains can evolve independently
  - identical copy operations must be programmed explicitly
  - runtime and memory overhead

M2M: Mixin Models (aka Markup Models)

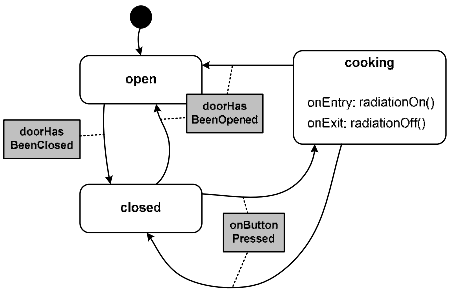
- The modification or transformation needs to be **parameterized**.



- Implications of mixin models
  - Provide **additional (mark up) information** about how a given model should be processed in a modification or transformation
  - Obviously used **together with the other forms**

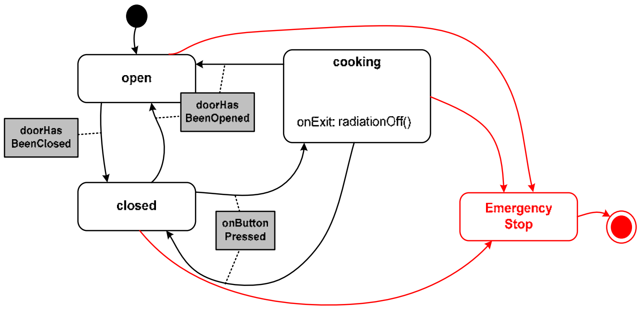
M2M: Model Weaving

- This is **like model merging**, but with the additional ability to **specify pointcuts**.
- Here is a model of a simple state machine. It serves as the base model, i.e. aspect models will be woven into it.



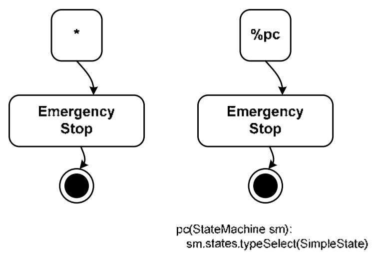
M2M: Model Weaving II

- This is the **desired result** of the aspect weaving process.
  - We want to add an emergency shutdown feature to the original state machine.
  - That means, from each normal state, we want to have a transition to a newly added Emergency Stop state.



M2M: Model Weaving III

- These are **two aspect models** that accomplish this task.
- The left one uses **the asterisk to select all instances** of the metaclass denoted by the rounded rectancle (i.e., *SimpleStates*).
- The right model uses a **pointcut expression** to achieve the same goal. The expression is referenced via the special form ***%expressionName*** and is defined elsewhere.
  - In this case, the expression also selects all instances of the metaclass *SimpleState*, making the two aspect models similar in effect.

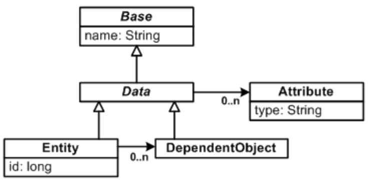


CONTENTS

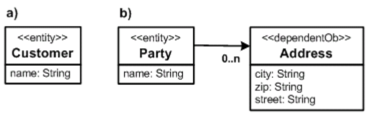
- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don't Duplicate – Transform!
- **Configuration over Composition**
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

Structural vs. Non-Structural Variability

- **Structural Variations**
  Example Metamodel



- Based on this sample metamodel, you can build a **wide variety of models:**

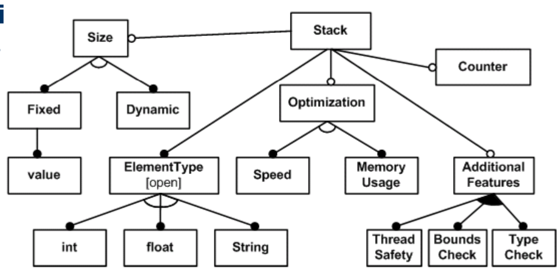


- **Non-Structural Variati**
  Example Feature Models

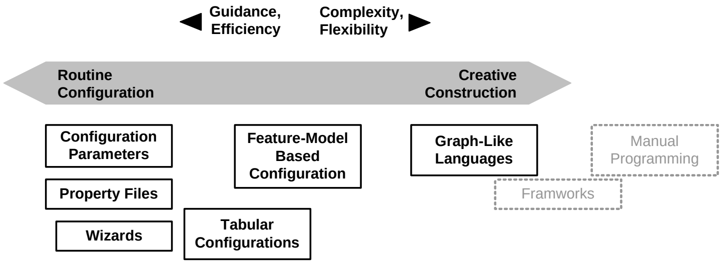
  Dynamic Size, ElementType: int, Counter, Threadsafe

  Static Size (20), ElementType: String

  Dynamic Size, Speed-Optimized, Bounds Check



Configuration and Creative Construction Languages



- This slide (adopted from K. Czarnecki) is **important for the selection of DSLs** in the context of MDSD **in general**:
  - The more you can move your DSL „form" to the configuration side, the simpler it typically gets.
  - We will see why this is especially important for behavior modelling.

C O N T E N T S

- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don't Duplicate – Transform!
- Configuration over Composition
- **Leverage Testing**
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

The Role of Testing in SW Development

- In all but very few cases, the **correctness of software cannot be verified** theoretically or formally.
- Thus the only way of verifying a system does what it should do is by **testing it extensively.**
- There are **different kinds** of things that can be tested:
  - Ensuring that the software **does what the developer wanted it to do**
  - Ensuring that what the developer programmed is actually what the system should do (i.e. **what the customer wants**)
  - Ensuring that the system **performs and scales** adequately
  - Ensuring that other **non-functional properties** work as specified (such as transactions, security, ...)
  - Ensuring that the **tools and technologies** used in the implementation **work together** well
- We will now look at each of these in the context of MDD.

Unit Testing

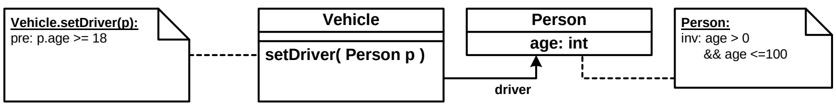
- Ensuring that the code does what the developer wants is called **Unit Testing**.
  - Tools such as JUnit provide a **framework to implement and repeatedly execute** unit tests
  - They are **written by the developer** as he develops his code.
  - Typically, they test **functionality**, not nun-functional properties
- You can always **write unit tests manually**, even if you use MDSD

Unit Testing II

- In the context of MDD, unit tests can be **generated from models**, too
  - Tests for **static properties** can be generated directly from the model.
  - For behavioral aspects, it should be a **different model** – because if tests are created from the same model as the implementation code, tests will always pass.
  - Additional Testcases can also be **generated from OCL expressions** (invariants, as well as pre- and postconditions).
  - When the code is generated, we can even **embed OCL constraint evaluation into the generated code** and check these at runtime.
- It is also possible to **generate input to tools** that verify/proof dynamic properties of models/systems

Unit Testing Example

- Consider the following model:



- This could result in the following code:

```
class Vehicle {
  ...
  public void setDriver( Person p ) {
    if (p.getAge() < 18 ) throw new ConstraintViolated();
  }
}
```

- A similar approach could be taken for the invariant in *Person*.
- In case of the invariant, it is easy to **automatically create a set of unit tests** that check ages like 0, 16, 78, 120, -1, 3.4 and see if the system behaves accurately.

Requirements Testing

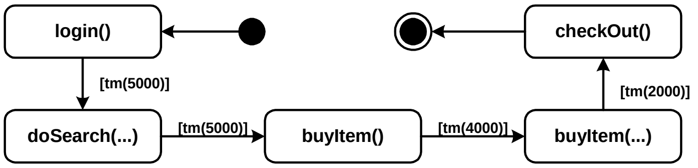
- Here we want to make sure that the **system does what the customer (or the requirements) say**.
- We use the **same technical approach** here as for unit testing. However, here the test cases are written by **domain experts** and not by the developer.
- If models are annotated with OCL constraints, they **are significantly more rich** that „typical“ requirements. A lot of test cases can be **generated** from these models.
- If we have a suitable, high-level modeling notation, **the domain expert can even specify test models himself**, or with some support by a technical person.
  - → A DSL for test specification, MD-Testing
- Because of the domain-specific notation, **developer / customer communication** about tests is simplified.

Performance and Scalability Testing

- This kind of testing basically works by **simulating a certain number of clients** and then **measuring response times and resource consumption.**
- Running such tests always requires a **setup of an environment similar to the production environment.** This is typically done **manually**, although some deployment artifacts can be generated from models.
- The simulated clients **can often be generated completely**. The input is basically
  - Which **operations** to call
  - At which **sequence** and **intervals**
  - In how many parallel **threads** or **processes**
  - And where to **store the timing measurements** and in which format

Performance and Scalability Testing Example

- A **statechart** can be used to specify this behaviour:



- Note that we do not care about **errors** and **functional testing** here. This is done in other tests!
- This statechart can be **code generated** into a client.
- An additional (textual) specification defines **how many parallel threads and processes** we have.
  - Tools for this task are also available outside MDD.

Additional Tests: Model Verification

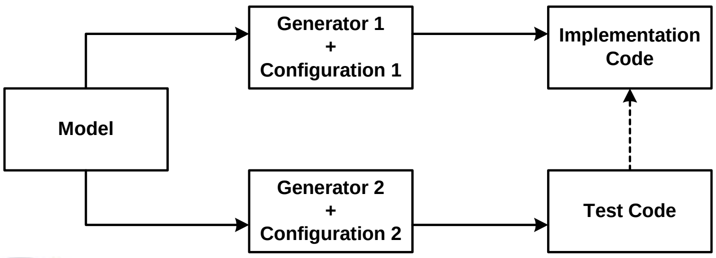
- In many cases it is possible to **detect design errors already in the models**. This step is called **model verification**.
  - Note that this kind of „testing“ is not available in classical development techniques – there are no semantically richer models
- It is easily possible to **verify modeling constraints** in the model **before** model transformation or code generation steps are executed.
- The most „extreme“ form is to **simulate certain aspects of the model** and proof certain properties.
  - Petri nets, for example, can be used to prove deadlock freedom in concurrent systems

Additional Tests: Generator Testing

- Many if not all of the previous statements on testing were based on the **assumption that the generator works fine.**
- Of course, **this has to be tested** also, at least in the early stages of the generator or the metamodel.
- Over time, however, the **generator will become a stable asset** that works reliably. Or you can buy one and trust it .... Just as you **trust C++/Java/etc. compilers**.
- If you have a cascaded generator, make sure you **test each step separately**.
  - In cases of M2M, this can be done by **writing test model-specific constraints**
  - In case of M2C, you should typically **test the semantics** of the code by running it and **writing unit tests** – testing the textual structure should be the last resort

Generator Testing: 2 Channel Concepts

- In safety-critical systems, the concept of **independent channels** is used
  - It is used to ensure that a failure in a system cannot go undetected by a second channel;
  - and to ensure that is is very unlikely that a failure does not affect both channels at the same time.
- The following diagram shows how to apply this idea to **testing generators**:



Generator Testing: 2 Channel Concepts II

- If **one generator** or configuration fails, **it is assumed that the other one does not fail** and will thus detect the failure.
- This **does not detect** failures in the model, of course. To detect those, we would need to **extend the 2 channel concept to include the model.**

CONTENTS

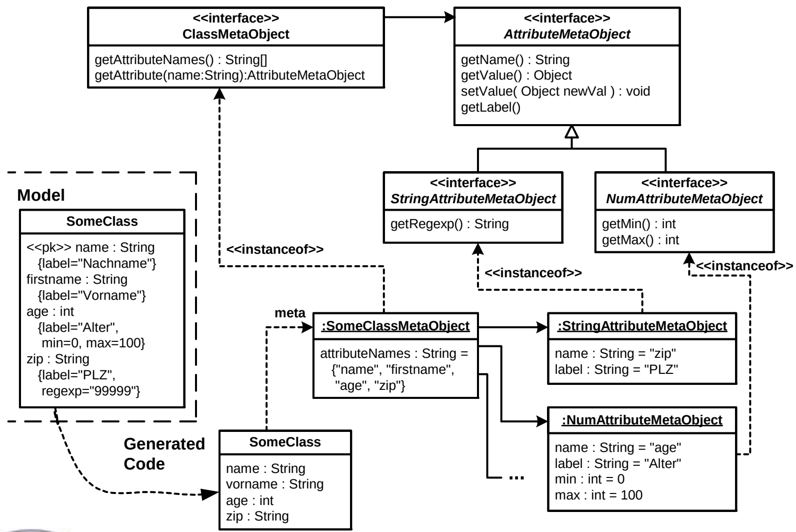
- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don't Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- **The Bridge to Frameworks**
- Behaviour Modeling
- Variant Management

Descriptive Metaobjects

- The generated application often needs **information about some model elements at run time** to control different aspects of the applicaton plaform.
- Use the information available at generation time to **code-generate meta objects** that describe the generated artifacts.
- Provide a means to **associate a generated artifact with its meta object.**
  - You add a *getMetaObject()* operation to the generated artifact.
  - You can also use a central registry that provides a lookup function *MetaRegistry.getMetaObjectFor(anArtefact)*. The implementation for the operations will be generated, too.
- Make sure the meta objects have a **generic interface** that can be accessed by the RICH DOMAIN-SPECIFIC PLATFORM.

Descriptive Metaobjects II

- Example:



Generated Reflection Layer

- You can even go one step further and **generate an "interpreter"**, a reflection layer that allows you to
  - "script" the system
  - build IDEs
- Since the reflection layer is **separate from the core classes,** it can be excluded from the „real" system for (performance reasons)

```
public interface RClass {
    // initializer – associates with
    // base-level object
  public setObject( Object o );
    // retrieve information about
    //the object
  public ROperation[] getOperations();
  public RAttribute[] getAttributes();
    // create new instance
  public Object newInstance();
}

public interface ROperation {
    // retrieve information about op
  public RParameter[] getParams();
  public String getReturnType();
    // invoke
  public Object invoke(Object params)
}

public interface RAttribute {
    // retrieve information about op
  public String getName();
  public String getType();
    // set / get
  public Object get();
  public void set( Object data );
}
```


CONTENTS

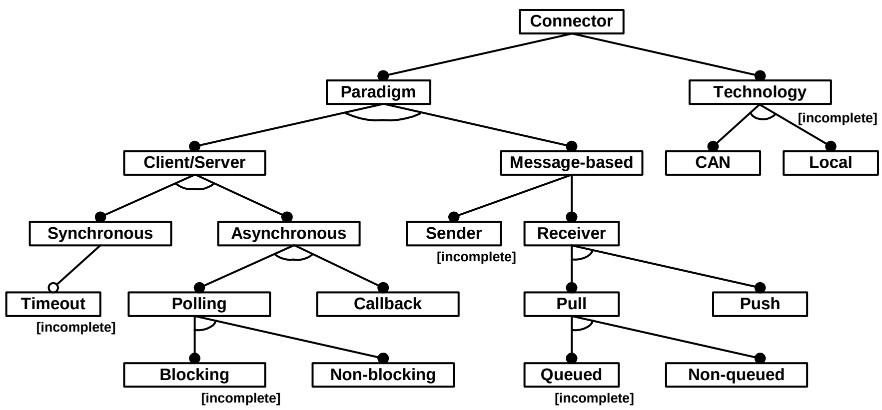
- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don't Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- **Behaviour Modeling**
- Variant Management

Behavioural Configuration

- The easiest way to model behaviour is to **reduce the behaviour to simple descriptive tags** if that is possible.
  - For example, to describe **communication between components**, if you are able to identify a **limited number** of well defined **alternatives** (synchronous, asynchronous, etc.), then the behaviour can be described by just **marking** it with the respective alternative.
  - You don't have to actually *describe* the behaviour, you just denote which alternative you need, and **the transformation or the code generator can make sure** the generated system does indeed behave as specified.
  - Selecting a valid option can be as easy as **specifying a certain property** or as complex as a **sophisticated selection based on a feature diagram**.

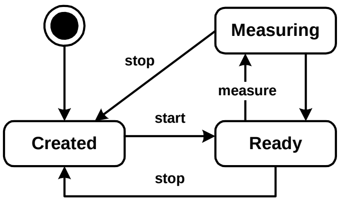
Behavioural Configuration II

- An example feature diagram for **configuration of communication behaviour** among components.



Using a specific formalism

- You can use a **well-known formalism** for specifying specific kinds of behaviour. Examples include
  - state charts or petri nets
  - first order predicate logic or business rule engines.
- Of course this approach only works in case the **required behaviour can** actually **be described** in the selected formalism.
- Advantages:
  - the **description** and the **semantics** of the behaviour is often quite clear
  - **editors and other tools** are available.
  - It is easy **to implement „engines“** for the particular formalism in order to execute the specifications.



- Within the constraints of the selected formalism, this approach already constitutes **creative construction**, not configuration.

Defining your own Formalism

- In case no formalism is readily available you may want to **invent your own.**
  - For example, in the insurance domain, you might want to use **textual languages** that specify verification constraints for insurance contracts.
- In that case you have to **define the formalism** (the language) **yourself**, and you have to build all the tooling. Writing engines might not always be easy because **it's not trivial to get the semantics** of the „invented“ formalism right.

```
PlausiGruppe SchuldnerGui <Schuldner> {
  Fehler "namePflichtfeld": name == null;
  Fehler "nameLaenge": name.length <3 || name.length > 50;
  Warnung "hausnummer": adresse.hausnummer ==
  Warnung "aktivaPassiva": bilanz.summeAktive
}
PlausiGruppe SchuldnerB2B <Schuldner> {
  Fehler "namePflichtfeld": name == null;
  Warnung "vornamePflichtfeld": vorname == nul
}
```

```
double ortsFaktor (Schuldner s):
  switch (s.adresse.stadt) {
    case "Pusemuckel": 0.5;
    default: 0.8;
};
betrag restWert (Forderung f):
  ortsFaktor (f.hauptSchuldner)
     * f.nominalwert;
```

Last reort: Turing-complete Language

- The last alternative you have is to use **existing Turing-complete languages**
  - such as a **3GL** or
  - **UML action semantics** languages
- Here you can specify any kind of behaviour - albeit using a very general language that is *not* domain-specific for the kind of behaviour at hand.

Integration with Structural Models

- It is always necessary to **associate a piece of behaviour with a structural element.**
- Structural „behaviour wrappers“ provide a natural point of **integration** between structural models and behavioural models.
- You should thus define certain **subtypes of structural elements** that implement their behaviour with a certain formalism, and not just allow developers to „implement“ the structural element. So, in case of components,
  - **process components** represent business processes; behaviour is modelled using state machines
  - **business rule components** capture (often changing) business rules; behaviour is modelled using predicate logic
  - **insurance contract calculation components** are implemented with a specific textual DSL.
  - And finally, 3GLs are used to **implement the beaviour** for the rest of the components; this should be a limited number.

CONTENTS

- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don‘t Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- **Variant Management**

Variant Management

- To make those possible, you'll need **model extension and weaving** – see above
  ➔ the oAW XWeave model weaver
- You also need **variants** of workflows, templates, transformations, constraints
  ➔ oAW supports the template, transformation and workflow aspects
- All of these "low-level" variation mechanisms must be **tied to a configuration model**
  ➔ oAW supports the use of any kind of model as a configuration model, specifically we support feature modeling tools (such as pure::variants)
- But that's another talk ☺

CONTENTS

- What is MDSD?
- Custom Metamodel
- Take care of your Metamodel
- Checks First & Separate
- Care about Generated Code
- Managing the Architecture
- Active Programming Model
- Multiple Viewpoints
- Rich Platform
- Cascading MDSD
- Extendible (Meta)model
- Graphical vs. Textual Syntax
- Don't Duplicate – Transform!
- Configuration over Composition
- Leverage Testing
- The Bridge to Frameworks
- Behaviour Modeling
- Variant Management

THE END.

Some advertisement ☺

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

  Völter, Stahl, Haase, Efftinge:

  Modellgetriebene Softwareentwicklung
  Technik, Engineering, Management
  2. Auflage

  dPunkt, 2007

  www.mdsd-buch.de

2nd Edition significantly updated

- A translation is available
  Model-Driven Software Development,
  Wiley, May 2006

  www.mdsd-book.org