DSL Best Practices
illustrated with Eclipse Tools

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

• Independent Consultant

• Based out of Heidenheim, Germany

• Focus on
  • Model-Driven Software Development
  • Software Architecture
  • Middleware

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

When working with „generic“ languages such as UML, always transform to your own metamodel first
A DSL always consists of
• Abstract syntax (Metamodel)
• Concrete syntax
• Semantics

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 easier
Take care of your Metamodel

The meta model is the central asset. It will grow over time. Make sure you use appropriate means to model and manage the metamodel.
Take Care of your Metamodel

• 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...
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...!)

- Ecore File
- Name Management (qualified, namespaces)
- Various constraints
Checks First & Separate

Before you do anything else with the model (transformation, generation) make sure you check constraints – these must not be part of the transformation to avoid duplication.
Checks First & Separate

- 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 have the checks several times.

- Make constraint checking a separate, and early step in the transformation workflow
### Checks First & Separate II

Here are some examples written in oAW’s Checks language.

For which elements is the constraint applicable?

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Expression</th>
<th>Error message in case Expression is false</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WARNING</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Example: StateMachine**
  ```java
  context StateMachine ERROR "States must have unique Names":
  states.typeSelect(State).forall(s1) !states.typeSelect |State|.
  exits(s2) (s1 != s2) & (s1.name == s2.name) |});
  ```

- **Example: Transition**
  ```java
  context Named if !Transition.isInstanceOf(this) ERROR this.metaType.name+" must be named":
  this.name != null;
  ```

- **Example: StartState**
  ```java
  context StartState ERROR "no incoming transitions allowed":
  this.inTransitions.size == 0;
  ```

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

- **Code completion & error highlighting**

Note the code completion & error highlighting 😊
• More complex constraints: Versioning and Evolution

```mermaid
graph LR
  SomeCompV1[Component]
  SomeInterface[Interface]
  soSomething(int, ValueObject)
  SomeCompV2[Component]
  SomeInterface[Interface]
  AnotherInterface[Interface]
  ValueObject[ValueObject]
  SomeCompV3[Component]
  SomeInterfaceV3[Interface]
  soSomething(int, ValueObjectV2)
anAdditionalOperation()
  ValueObjectV3[ValueObject]
```
Multiple Viewpoints

Use several models to describe a system from several viewpoints – each viewpoint will have a suitable concrete syntax and metamodel
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.
Multiple Viewpoints II: CBD Example

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

![Diagram of Type, Composition, and System Models](image.png)

<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>

<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>
Multiple Viewpoints III: CBD Example Metamodels

Types

Composition

Deployment
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
Architecture First

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.
A successful system is built based on a **well-defined architecture**, often along the lines of the illustration below.

Various parts/layers of this stack can be generated, or developed with meta-model and generator support.

Use **Model-2-Model Transformations** to implement higher layers based on the abstractions provided by lower layers.
Architecture First II

**Functional Domain 1**
- MDSD Infrastructure
- Domain 1 Model
  - Functional Domain 1

**Functional Domain 2**
- MDSD Infrastructure
- Domain 2 Model
  - Functional Domain 2

**Input Models**
- MDSD Infrastructure

**Output Models**

**Basic Technical**
- MDSD Infrastructure

**Code for Target Platform**
Architecture First III: Generated Stuff

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

DSL Best Practices
Architecture First IV: 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.

```xPand
DEFINE SwitchBasedImpl FOR StateMachine
FOREACH states.typeSelect|State) AS s
    public static final int s.constant
ENDFOREACH
```
DSL Best Practices

Architecture First V: Code Generation

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

---

```xPand
<IMPORT simpleParser.
<EXTENSION templates::GeneratorUtil.

<DEFINE file FOR StateMachine
  <FILE basePath="/\"Abstract\"/name.toFirstUpper/\"java\"/
    <NAME spacePackage="/\">

  abstract class <implBaseClass> extends StateMachine
    
    <statesInName()| current>
    
    boolean terminated = false;

    public void handleEvent({eventInName()| event})
      if { terminated | throw new RuntimeException(\"this sm is terminated\") };

      switch { currentState }
      
        <FOREACH states AS >
          case <shortStateId>() :
            <FOREACH s.transitions AS >
              if { event == s.event.eventId(this) }
                <DEFINED executeTransition(this)>
                break;
              
              <DEFINED handleIllegalTransition>
              
            <ENDFOREACH>
            break // break out if no suitable transition has been found!
            
          <ENDFOREACH>

          public <statesInName()| currentState()>
            return currentState;

        <ENDFOREACH>

<ENDDEFINITION>

<DEFINE handleIllegalTransition FOR StateMachine
<ENDDEFINITION>

<DEFINE executeTransition(StateMachine sm) FOR Transition
  <FOREACH actions AS >
    this.<a.getMethodName()>()
  <ENDFOREACH>

  currentState = s.to.stateId(sm);
<ENDDEFINITION>
```

---

*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*
Extendible Metamodel

When generating/transforming models, you often need additional properties on your metaclasses, or whole even new metaclasses; make sure you can add them, without touching the metamodel itself!
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
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
Active Programming Model

You should restrict the freedom of developers ...
making the code more consistent and structured.
Help developers write correct code!
Active Programming Model

• 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, Recipe Framework
There are various ways of integrating generated code with non-generated code:

a) 

b) 

c) 

d) 

e) 

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

- Recipes can be arranged hierarchically

  - "Green" ones can also be hidden

  - This is a failed check

  - Here you can see additional information about the selected recipe
I now add the respective \textit{extends} clause, & the message goes away – automatically.
Now I get a number of compile errors because I have to **implement the abstract methods** defined in the super class:

![Image](image.png)

- 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.
Managing the Architecture

MDSD can help to make sure an architecture is used consistently and „correctly“ in larger teams
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 modeled in the model?
The programming model shown below is bad:

```java
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

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

```java
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
Graphical vs. Textual Syntax

Textual DSLs are often neglected in the MDSD/MDA space. Graphical DSLs are often ignored in other circles. When do you use which flavour?
Graphical vs. Textual Syntax

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

![Diagram of an editor built with Eclipse GMF with a state machine model and various panes: Tool Palette, Overview Pane, Model Element Properties.]

*These rectangles are to demo decorations 😊*
This is a textual editor for the same metamodel.

- **Literals have become keywords**
- **Constraints are evaluated in real time**

```plaintext
statemachine CdPlayer {
    // initial state
    state Off {
        shutdown
        powerSwitchPressed -> Off
    }
    state Open {
        openTray
        openClosePressed -> On
        powerSwitchPressed -> Off
        closeTray
    }
    /* composite state */
    statemachine On {
        checkCD
        openClosePressed
        powerSwitchPressed
        // children
        state Stop {
            stopPlaying
            playPressed -> Play
        }
        state Play {
            startPlaying
            stopPressed -> Stop
            pausePressed -> Pause
        }
        state Pause {
            pausePlaying
            stopPressed -> Stop
            pausePressed -> Play
        }
    }
}```
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)
Don‘t Duplicate – Transform!

Direct Model-to-Code Transformation is often not enough, since you‘ll either have to duplicate stuff into code generation templates or you have to add “obvious” stuff to your models. Neither is desirable.
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.
Don't Duplicate – Transform! II

• 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
Don't Duplicate – Transform! III

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

```java
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.
Partitions/Layers/Cascading

Architecture can be nicely layered and architected to be as small and consistent as possible
Partitions/Layers/Cascading

- Functional Domain 1
  - MDSD Infrastructure
  - Domain 1 Model
- Functional Domain 2
  - MDSD Infrastructure
  - Domain 2 Model

Input Models

Basic Technical MDSD Infrastructure

Code for Target Platform
**Partitions/Layers/Cascading II**

**Type Model**

- **AddressManager**
- **AddressStore**
- **CustomerManager**

**Composition Model**

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

**System Model**

```xml
<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>
```

**Valuetype Model**

- **Address**

```xml
<entity>
  Person
  name: String
  firstName: String
</entity>
```

```xml
<valuetype>
  Address
  street: String
  zip: String
  city: String
</valuetype>
```

**Generate Model**

```xml
<generate>
  SomeComponent
</generate>
```

```xml
<generate>
  SomeComponent.java
</generate>
```

```xml
<man-code>
  SomeComponent.java
</man-code>
```

**Interface Model**

```xml
<interface>
  SomeInterface
</interface>
```

```xml
<generate>
  SomeInterface.java
</generate>
```

**Component Model**

```xml
<component>
  SomeComponent
</component>
```

```xml
<generate>
  SomeComponent.java
</generate>
```
Partitions/Layers/Cascading III

- `SomeEntity.java` generated from `SomeEntity` entity.
- `SomeEntityDAO.java` generated from `SomeEntityDAO` interface.
- `SomeEntityDAOBase.java` generated from `SomeEntityDAO` component.
- `SomeEntityDAO.java` generated again from `SomeEntityDAOBase.java`.

The diagram shows the relationship and transformation between entities, interfaces, and components in a software architecture.
Partitions/Layers/Cascading IV

<<proc-component>>
AProcess

<<trigger-interface>>
AProcessInterface

<<transform>>
* operations...

<<transform>>

sm AProcess

1

s1

s2

s3

<<trigger-interface>>
AProcessInterface

<<entity>>
AProcessData

<<generate>>

attributes...

<<generate>>

<<generate>>

<<gen-code>>
AProcessBase.java

guard operations... (abstract)
action methods... (abstract)

<<man-code>>
AProcess.java

1 data

<<gen-code>>
AProcessProcBase.java

<<gen-code>>
AProcessData.java

<<generate>>

<<entity>>
AProcess

<<transform>>

1

<<gen-code>>
AProcessProcBase.java

<<man-code>>
AProcess.java
Configuration over Composition

Architecture can be nicely layered and architected to be as small and consistent as possible.
Configuration over Composition

• **Structural Variations**
  Example Metamodel

- **Base**
  name: String

- **Date**
  0..n Attribute
  type: String

- **Entity**
  id: long

  0..n **DependentObject**

• **Non-Structural Variations**
  Example Feature Models

  Dynamic Size, **ElementType**: int, Counter, Threadsafe

  Static Size (20), **ElementType**: String

  Dynamic Size, Speed-Optimized, Bounds Check

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

Architecture can be nicely layered and architected to be as small and consistent as possible.
Specific Implementation DSLs

- 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!**
• 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
  -> JRuby, but that’s another talk 😊
Thanks!

Please ask questions!
Some advertisement 😊

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

  Völter, Stahl:
  **Modellgetriebene Softwareentwicklung**
  Technik, Engineering, Management
  dPunkt, 2005
  [www.mdsd-buch.de](http://www.mdsd-buch.de)

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

  **Model-Driven Software Development**
  Wiley, Q2 2006