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

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

Markus Völter
voelter@acm.org
www.voelter.de
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**
How MDSD improves Software Quality

Conceptual Architecture: Starting Points

- Independent parts
  - Communicating Processes
  - Event/Message-bases Systems
  - Data Flow
    - Batch Sequential
    - Pipes and Filters
  - Data Centered
    - Repository
    - Blackboard
- Workflow
- Call and Return
  - Procedural
  - Object oriented
  - Layers
- Virtual Machine
  - Interpreters
  - Rule-Based
  - Microkernel
- Components
- Plug-Ins

Markus Völter
Formalization of Architecture in MDSD using Metamodels

if a component B is a new version of a component A, then B has to have the same interfaces, additional provided interfaces, fewer required interfaces or new version of interfaces of A.

A new version of an interface has to have the same return type and the same parameters - or parameters with subtypes.

If a component B is a new version of a component A, then B has to have the same interfaces, additional provided interfaces, fewer required interfaces or new version of interfaces of A.
Multi-Viewpoint models

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

Component
  name

Component Interface Requirement
  name

providedInterface

Interface
  name

Operation
  name

Parameter
  name

required Interface

target

returnType

type

exception

Exception

Type
  name
How MDSD improves Software Quality

Type Metamodel II (Data)
Composition Metamodel

**Context ComponentInstance inv:**
- Foreach of type's Component-InterfaceRequirements
- There must be a Wire of the same name

**Context Wire inv:**
- The type of the target instance must provide the Interface pointed to by the Wire's cireq's target
System Metamodel

Composition Stuff

Configuration
  name

Component Instance
  name
  instance

Wire
  name

System Stuff

System
  name

Node
  name

Container
  name
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
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.
How MDSD improves Software Quality

Talk Metamodel II

- 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

<table>
<thead>
<tr>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain Platform</td>
</tr>
<tr>
<td>- Core Entities</td>
</tr>
<tr>
<td>- Core Valuetypes</td>
</tr>
<tr>
<td>- Business Rules</td>
</tr>
<tr>
<td>- Business Services</td>
</tr>
<tr>
<td>Technical Platform/Middleware</td>
</tr>
<tr>
<td>- Persistence</td>
</tr>
<tr>
<td>- Transactions</td>
</tr>
<tr>
<td>- Distribution</td>
</tr>
<tr>
<td>- Scheduling</td>
</tr>
<tr>
<td>- Hardware Access</td>
</tr>
<tr>
<td>Programming Language</td>
</tr>
<tr>
<td>Operating System</td>
</tr>
</tbody>
</table>
Three Basic Viewpoints – Generated Stuff

• What can be generated?
  • Base classes for component implementation
  • Build-Scripts
  • Descriptors
  • Remoting Infrastructure
  • Persistence
  • …
How MDSD improves Software Quality

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.
How MDSD improves Software Quality

Code Generation II

- 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**.

---

**Example xPand Code**

```xpand
// Opens a File
<IMPORT simpleSM>
<EXTENSION templates:GeneratorUtil>

// Define file FOR StateMachine
<FILE basepath="/\Abstract" name="toFirstUpper\" extension="java\">
  namespace = basePackage || \$

  abstract class simpleBaseClass
    
    <states\$name\$> current;
    The Boolean terminated = false;
    
  public void handleEvent (String\$name\$ || \$event\$) {
    if (terminated || three new RuntimeException( "this sm is terminated!" ) ;
      switch (currentState) {
        FOREACH states AS native:
          case <.shortStateId\$>
            FOREACH s.transitions AS $trans
              if ($event == s.event.eventId(this))
                EXPAND executeTransition(this);
                break;
          }
          EXPAND handleIllegalTransition;
          ENDFOREACH;
          break // break out if no suitable transition has been found!
          ENDFOREACH.

        public String\$name\$ || \$get\$Current\$State\$
          return currentState;

        ENDDEFINITION

        DEFINE handleIllegalTransition FOR StateMachine
          ENDDEFINITION

        DEFINE executeTransition(StateMachine sm) FOR Transition
          FORSTATE actions AS native:
            this.sa.methodName()();
          ENDFOREACH;
          currentState = sm.to.stateId(sm);
          ENDDEFINITION
```
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
Managing Architecture

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

```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
Improved Solution, with MDSD

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

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) ![Diagram a]

b) ![Diagram b]

c) ![Diagram c]

d) ![Diagram d]

e) ![Diagram e]
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!
Here’s an error that suggests that I extend my manually written class from the generated base class:
I now add the respective *extends* clause, & the message goes away – automatically.

Adding the extends clause makes all of them green.
Now I get a number of compile errors because I have to implement the abstract methods defined in the super class:

<table>
<thead>
<tr>
<th>Description</th>
<th>Resource</th>
<th>Path</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>The type CoPlayer must implement the inherited abstract method closeCD()</td>
<td>CoPlayer.java</td>
<td>oaw1r2.demo.gmf.stateachines ... Line 3</td>
<td></td>
</tr>
<tr>
<td>The type CoPlayer must implement the inherited abstract method closeKey()</td>
<td>CoPlayer.java</td>
<td>oaw1r2.demo.gmf.stateachines ... Line 3</td>
<td></td>
</tr>
<tr>
<td>The type CoPlayer must implement the inherited abstract method closePlayer()</td>
<td>CoPlayer.java</td>
<td>oaw1r2.demo.gmf.stateachines ... Line 3</td>
<td></td>
</tr>
<tr>
<td>The type CoPlayer must implement the inherited abstract method pausePlayer()</td>
<td>CoPlayer.java</td>
<td>oaw1r2.demo.gmf.stateachines ... Line 3</td>
<td></td>
</tr>
<tr>
<td>The type CoPlayer must implement the inherited abstract method shutdown()</td>
<td>CoPlayer.java</td>
<td>oaw1r2.demo.gmf.stateachines ... Line 3</td>
<td></td>
</tr>
<tr>
<td>The type CoPlayer must implement the inherited abstract method startPlayer()</td>
<td>CoPlayer.java</td>
<td>oaw1r2.demo.gmf.stateachines ... Line 3</td>
<td></td>
</tr>
<tr>
<td>The type CoPlayer must implement the inherited abstract method stopPlayer()</td>
<td>CoPlayer.java</td>
<td>oaw1r2.demo.gmf.stateachines ... Line 3</td>
<td></td>
</tr>
</tbody>
</table>

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.
How MDSD improves Software Quality

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**.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Expression</th>
<th>Error message in case Expression is false</th>
</tr>
</thead>
</table>
| ERROR or WARNING | `states.typeSelect(State).forAll(s1 | `states.typeSelect|State| .
|                          | exists(s2 | `s1 != s2 | & (s1.name == s2.name | });` |
| ERROR or WARNING | `if !Transition.isInstance(this)` | `ERROR this.metaType.name | “ must be named”`: |
| ERROR or WARNING | `this.name != null;` |  |
| ERROR or WARNING | `StartState` | `“no incoming transitions allowed”`: |
| ERROR or WARNING | `StartState` | `“start state must have one out transition

Note the **code completion & error highlighting 😊**
Partitions/Layers/Cascading

Architecture can be nicely layered and architected to be as small and consistent as possible.
How MDSD improves Software Quality

Levels of MDSD

- Input Models
  - MDSD Infrastructure
  - Output Models

<table>
<thead>
<tr>
<th>Domain 1 Model</th>
<th>Domain 2 Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Domain 1 MDSD Infrastructure</td>
<td>Functional Domain 2 MDSD Infrastructure</td>
</tr>
</tbody>
</table>

- Basic Technical MDSD Infrastructure
- Code for Target Platform
How MDSD improves Software Quality

Levels of MDSD III – M2M Transformations

<table>
<thead>
<tr>
<th>Type Model</th>
<th>Composition Model</th>
<th>System Model</th>
</tr>
</thead>
</table>
| <<component>> AddressManager | <<interface>> AddressStore | addOrUpdateContact( p: Person) : void 
| |
| | | addAddress( p: Person, a: Address) : void 
| | | getAddresses( p: Person ) : Address[] 
| | | |
| | <<entity>> Person | name: String 
| | | firstName: String 
| | | |
| | <<valueType>> Address | street: String 
| | | zip: String 
| | | City: String 
| | | 0..n 
| <<component>> CustomerManager | <<generate>> SomeComponent.java |
| <<generate>> SomeComponent Base.java | <<generate>> SomeInterface.java |
Levels of MDSD III – M2M Transformations II

How MDSD improves Software Quality

<<entity>>
SomeEntity

<<transform>>
SomeEntityDAO

<<interface>>
SomeEntityDAO

<<generate>>
SomeEntity-DAO.java

<<transform>>
SomeEntityDAO

<<generate>>
SomeEntity-DAOBase.java

<<generate>>
SomeEntityDAO.java

<<generate>>
SomeEntity.java

<<generate>>
SomeEntity-DAOBase.java
How MDSD improves Software Quality

Levels of MDSD III – M2M Transformations IV

<<generate>>
<<gen-code>>
AProcess

<<trigger-interface>>
AProcessInterface
operations...

<<transform>>

<<entity>>
AProcessData
attributes...

<<transform>>

<<gen-code>>
AProcessProcBase.java
guard operations... (abstract)
action methods... (abstract)

<<man-code>>
AProcess.java
How MDSD improves Software Quality

Levels of MDSD III – M2M Transformations III

openArchitectureWare

Model (UML) → export → Parser → Model (XMI) → Model (Object Graph) → Model Transformer → Modified Model (Object Graph) → Code Generator → Generated Code

(may be repeated)
The model modification shows how to add an additional state & some transitions to an existing state machine (emergency shutdown)

```java
import statemachine2;

extension statemachine2::constraint::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.mdsd-buch.de

• An very much updated translation is under way:
  **Model-Driven Software Development**, Wiley, Q2 2006
  www.mdsd-book.org