MD* Best Practices

Markus Voelter

Independent/itemis AG

www.voelter.de
voelter@acm.org

About

Markus Voelter
Independent Consultant/itemis

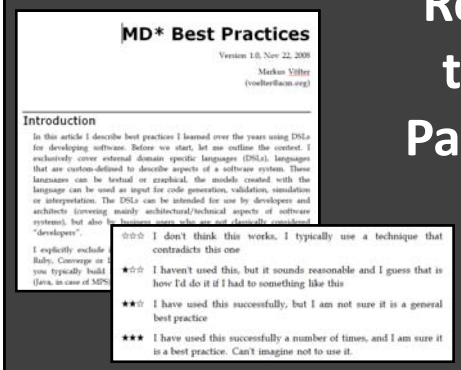
Software Architecture
DSLs & MDSD
Product Lines

About

Markus Voelter
Independent Consultant/itemis

http://www.voelter.de
voelter@acm.org
skype: schogglad

Read the Paper!



MD* Best Practices

Version 1.0, Nov 22, 2008

Markus Völter
(voelter@acm.org)

Introduction

In this article I describe best practices I learned over the years using DSLs for developing software. Before we start, let me outline the context. I exclusively cover external domain specific languages (DSLs), languages that are custom-defined to describe aspects of a software system. These languages can be textual or graphical, the models created with the language can be used as input for code generation, validation, simulation or interpretation. The DSLs can be intended for use by developers and architects (covering mainly architectural/technical aspects of software systems), but also by

☆☆☆ I don't think this works. I typically use a technique that contradicts this one

★☆☆ I haven't used this, but it sounds reasonable and I guess that is how I'd do it if I had to something like this

★★☆ I have used this successfully, but I am not sure it is a general best practice

★★★ I have used this successfully a number of times, and I am sure it is a best practice. Can't imagine not to use it.

http://www.voelter.de/data/articles/DSLBestPractices-Website.pdf

1 Designing the Language

Limit Expressiveness

Limit Expressiveness

Limit Expressiveness

Configuration

select
…from config space

… Properties
… Feature Models

Limit Expressiveness

Customization

compose
… creatively

… Vocabulary
… Sentences
… Box and Line

Limit Expressiveness

Precision

precisely what

… facts
… declarative
… domain experts can!

Limit Expressiveness

Algorithmic Completeness

formally how

… automation
… execution
… in model processors
… developers can!

Limit Expressiveness

Use a **3GL** if necessary
Generate APIs, Hooks

Graphical vs. Textual

Notation, ...

Domain Users
care deeply
about notation!

„UI“ for the language

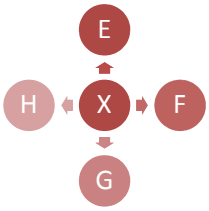
Grapical vs. Textual

Graphical

Sequence/Flow

Grapical vs. Textual

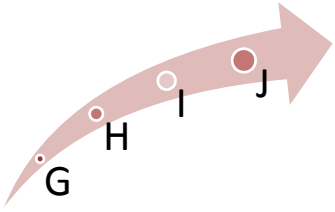
Graphical



Relationships

Grapical vs. Textual

Graphical



Timing

Grapical vs. Textual

Textual

In all other cases!

Grapical vs. Textual

Textual

Real simple…

- …CVS/SVN Integration
- … Diff/Merge
- … Build automation
- … Model Migration

Grapical vs. Textual

Text + Visualization

- Model-to-Model
- Automatic Layout
- Read-Only
- Drill-Down

Viewpoints

Viewpoints

$$system = \sum_{i=1}^{n} viewpoint_i$$

Viewpoints

$$viewpoint_i \Rightarrow (abstractions_i, notations_i)$$

Viewpoints

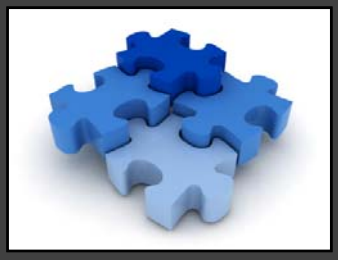
Well-defined Dependencies and Connection Points

Viewpoints

Try to avoid **overlap** and the need for **synchronization.**

Viewpoints

Partitioning



Partitioning

MD* tools do not **scale** arbitrarily!

Partitioning

Partitions

… separate resources
… != logical structure
… units of check in/out
… processable separately

Partitioning

cross Partition ~~X~~ references

… lazy
… by proxy
… by name w/ linker

Partitioning

Partition

… not transparent
… part of language design
… referencable elements
… „include path“

The Fallacy of Generic Languages

Fallacy of Generic Lang.

You can model everything with UML somehow!

Fallacy of Generic Lang.

Problem

Shoehorning domain abstractions into the generic language

Fallacy of Generic Lang.

Problem

Sidetracked by existing abstractions and notations

Fallacy of Generic Lang.

Notations/ Abstractions extensible via Profiles

Very Limited Tool Support!

Fallacy of Generic Lang.

Meta Model Complexity!

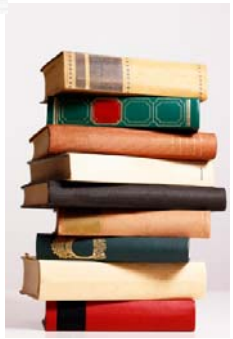
Fallacy of Generic Lang.

But don‘t reinvent the wheel either.

Fallacy of Generic Lang.

Where are **standards** useful?

Fallacy of Generic Lang.



People have to learn **underlying concepts** anyway.

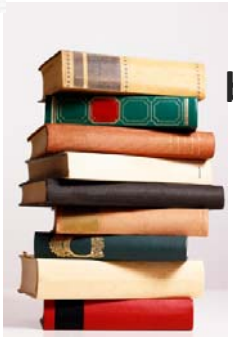
Fallacy of Generic Lang.

Is UML with a profile still a **standard language**?

Fallacy of Generic Lang.

On which **meta level** do I want to standardize? M2 (UML), M3 (MOF)?

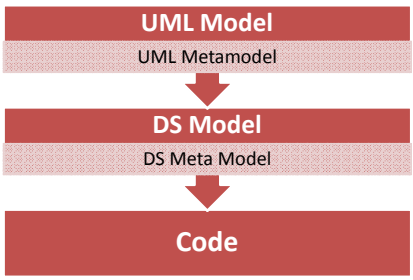
Fallacy of Generic Lang.



Isn't a **DSL** based on **MOF** as „standard" as a profile based on UML?

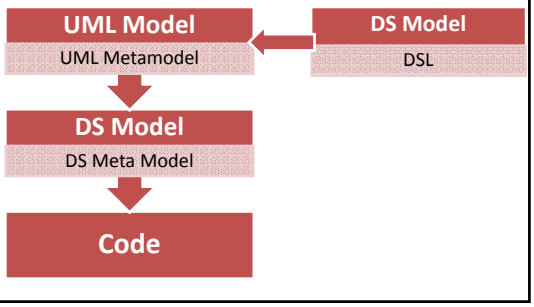
Fallacy of Generic Lang.

Introduce an **intermediate** language



Fallacy of Generic Lang.

separate **viewpoints** UML + DSL



Learn from 3GLs

Learn from 3GLs

DSL != 3GL

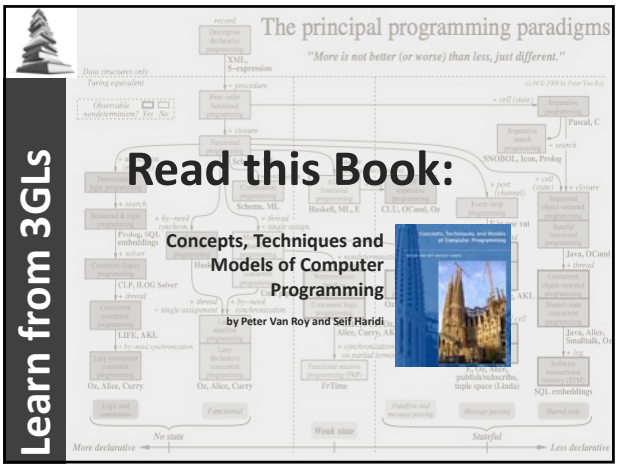
But:

Learn from 3GLs

We can learn:

- Namespaces
- Scoping
- Visibility
- „Abstract"
- Type Systems
- Specialization …

Learn from 3GLs



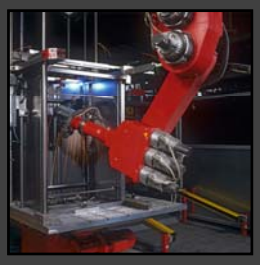
Read this Book:

Concepts, Techniques and Models of Computer Programming

by Peter Van Roy and Seif Haridi

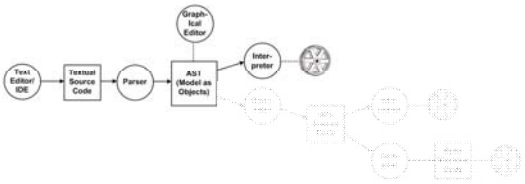
2 Processing the Models

Interpretation vs. Generation



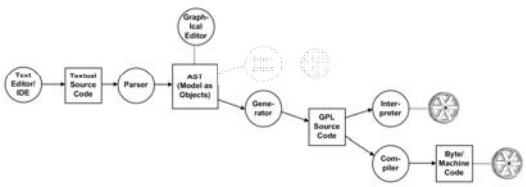
Interpretation vs. Generation

Interpretation



Interpretation vs. Generation

Generation



Interpretation vs. Generation

Generation

resulting code
can be easily
inspected

Interpretation vs. Generation

Generation

resulting code
can be easily
debugged

Interpretation vs. Generation

Generation

resulting code
can be
optimized
and more
efficient

Interpretation vs. Generation

Generation

Templates can be
derived
from existing code

Interpretation vs. Generation

Generation

work around
limitations
of target language

Interpretation vs. Generation

Generation

no changes
to target environment

(leaves no trace)

Interpretation vs. Generation

Interpretation

faster
turnaround
no regeneration
test
build
deploy

Control Manually Written Code

Control Manual Code

After code generation
how do you make sure
developers follow
all the required
procedures?

Control Manual Code

procedures?

… subclass
… overwrite
… naming conventions

Control Manual Code

Compiler Errors
are not enough.

wrong
abstraction
level!

Control Manual Code

generate **checks**
against the **code base**
evaluated by the **IDE**

Control Manual Code

```
if (false) {
  GeneratedBaseClass x =
    new ManualSubclass();
}
```

Make Code True to Model

Code true to the Model

Analyses on the model can verify all kinds of **properties** about the **system.**

Code true to the Model

Analyses on the model can verify all kinds of **properties** about the **system.**

Iff the code is **true** to the model

Code true to the Model

Use a clever **programming model** that does not allow **violations.**

Code true to the Model

generate the **configuration** for **architecture analysis** tools.

Model-2-Model To Simplify Generators

M2M to Simplify Generators

Reducing template
complexity
is important.

M2M to Simplify Generators

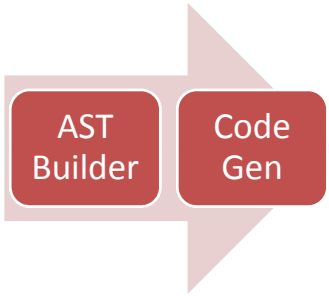
Reducing tempalte
complexity
is important.

Separation of Concerns
is the way to go.

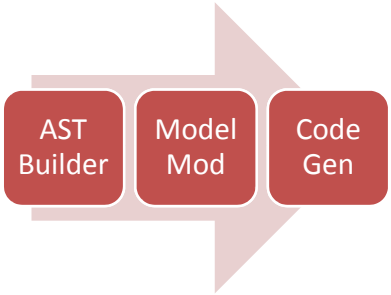
M2M to Simplify Generators

Instead of putting
complex logic
into the templates
put it into an M2M
that runs
before code gen.

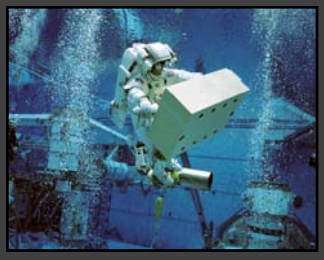
M2M to Simplify Generators



M2M to Simplify Generators



Model-2-Model
For Simulation and Proof



M2M for Simulation and Proof

Many
useful formalisms
already exist.

M2M for Simulation and Proof

Many
useful formalisms
already exist.

Simulation
Proofs
Properties

M2M for Simulation and Proof

Use an **M2M**
for this if possible.

Often the input is **XML**
so you actually
„generate code“

Cascading

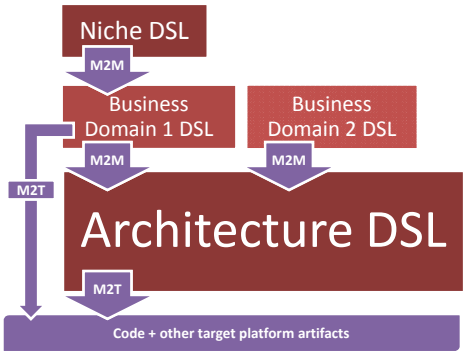
Cascading

PIM?
PSM?
PSSM?

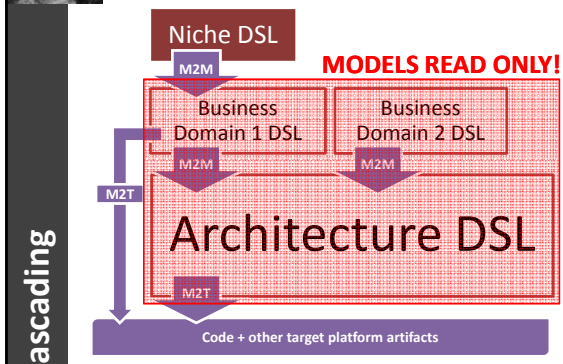
Cascading

Bottom Up
Works Better!

Cascading

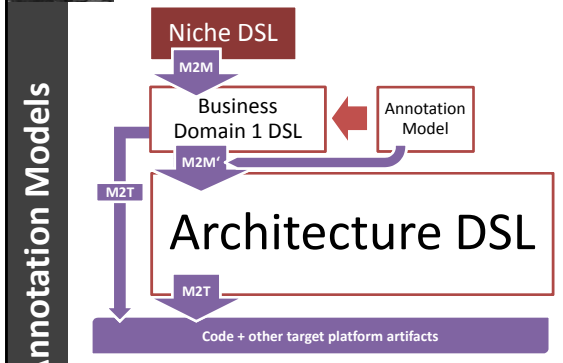


Cascading



Annotation Models

Annotation Models



Annotation Models

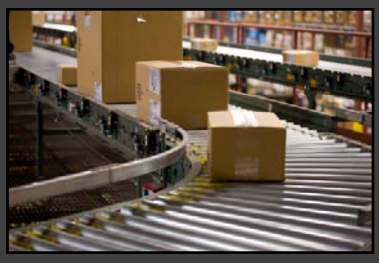
Annotation Model
references
elements in base model.

Transformation
takes additional information into account.

Annotation Models

Make sure the
annotation model
only captures
Exceptions
from the default in the templates.

Classify Behaviour



Action Semantics Languages ?

useful! But…

Classify!

… state based
… business rules
… mathematics

… or a specific DSL
… 3GL code

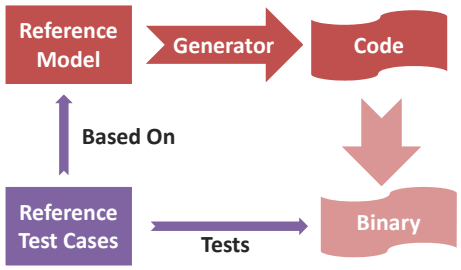
Don‘t forget Testing

Limited Expressiveness.
Reduced Need For Tests.

Constraint Checks.
A Form of Test.

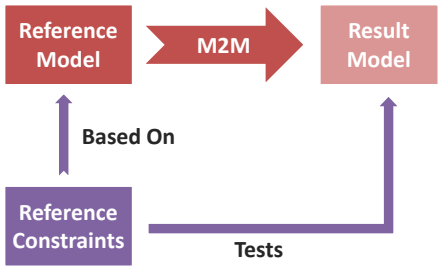
Don't Forget Testing

Testing Generators



Don't Forget Testing

Testing Transformations



Don't Forget Testing

Testing Metware

Reference Model

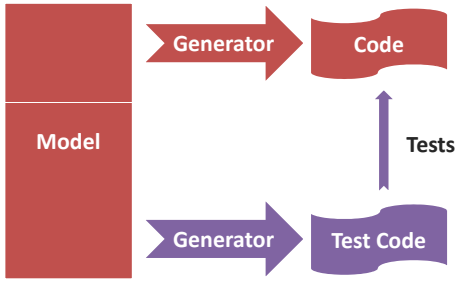
Reference Test Cases

Reference Constraints

… maintaned!
… by metaware developers

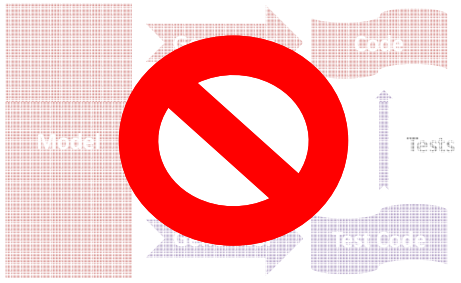
Don't Forget Testing

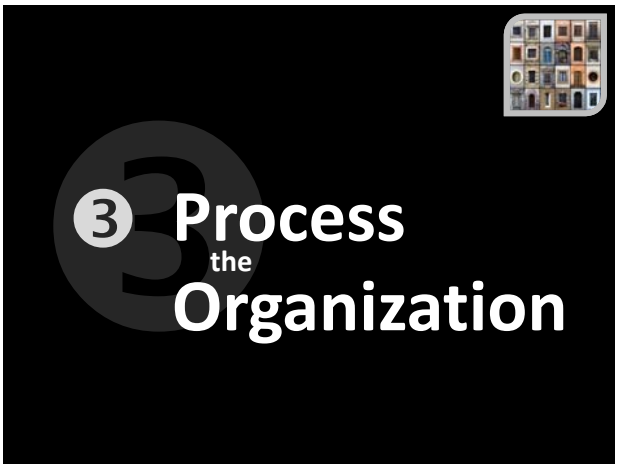
Self-Fulfilling ….!

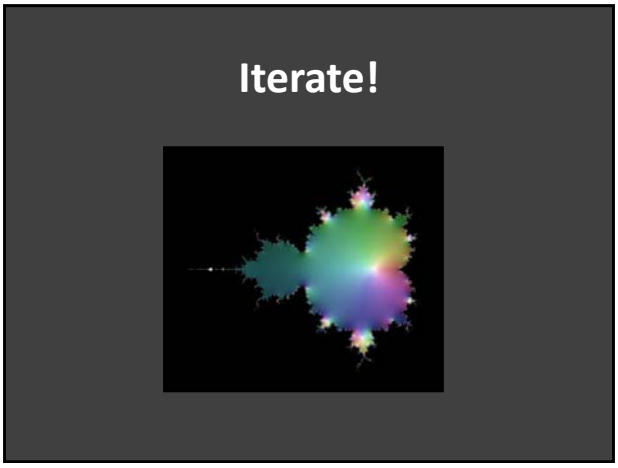


Don't Forget Testing

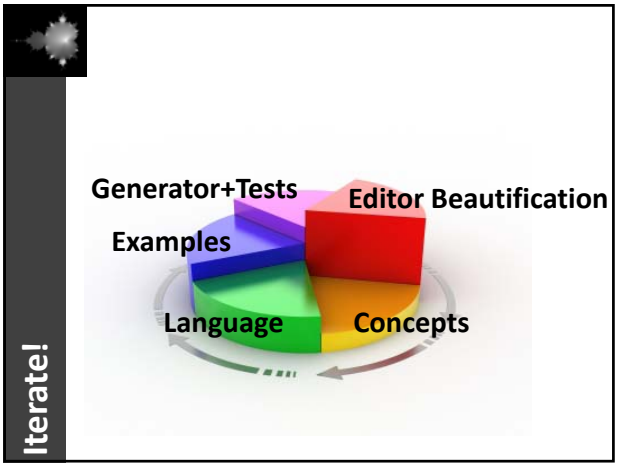
Self-Fulfilling ….!




3
Process
the
Organization


Iterate!

Waterfall is bad!
With or Without MD*
Iterate!


Generator+Tests
Editor Beautification
Examples
Language
Concepts
Iterate!

Co-Evolve Language
and Concepts

Co-Evolve Language & Domain

Understanding the Domain

Building The Language

Co-Evolve Language & Domain

Understanding the Domain

Building The Language

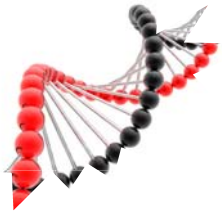
Co-Evolve Language & Domain

Building a language requires **Formalization**

Co-Evolve Language & Domain

Building a language requires **Formalization** requires you to **think** and **decide** about the domain.

Co-Evolve Language & Domain



requires frequent **Evolution!**

Co-Evolve Language & Domain

and flexible, agile **Tooling!**

Documentation is still necessary

Documentation still necessary

The DSL and the „programs“ are documentation.

Documentation still necessary

The DSL and the „programs“ are documentation.

Not Quite!

Documentation still necessary

Language Definition is not a **Teaching Tool!**

Documentation still necessary

Tutorials

- … Concepts
- … How to use Language
- … How to integrate manual code

Example-Driven!

Documentation still necessary

Language Definition captures **the WHAT** but not **the WHY**

Documentation still necessary

Rationales

… why the concepts?
… why we generate what we generate
… target platform decisions and idioms

Documentation still necessary

Different Media

Compatible Organization

Compatible Organization

MD* requires **cross-project** work.

Compatible Organization

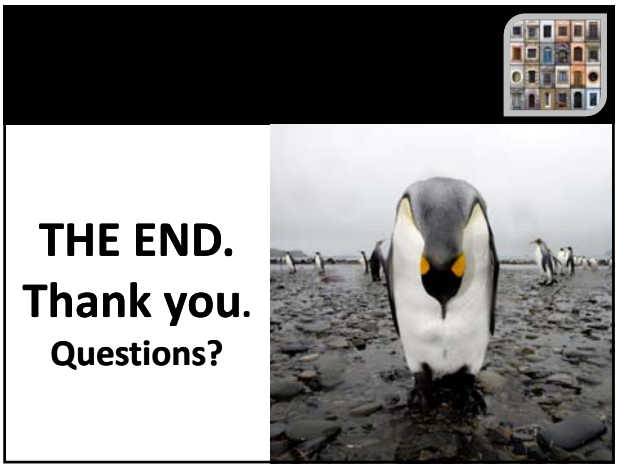
MD* requires **cross-project** work.

A strict **project-focused** organization does not work

Compatible Organization

Make **room & budget** for cross-cutting work.

Open Source?


THE END.
Thank you.
Questions?