Open Source Modeling

Generating Code with Xpand

Bernd Kolb
b.kolb@kolbware.de
http://www.kolbware.de

Markus Voelter
voelter@acm.org
www.voelter.de
Introduction to Modeling
Different stages
  ▪ Creating the metamodel
  ▪ Creating a model
  ▪ Checking a model
  ▪ Transforming a model
  ▪ Generating code
  ▪ Xpand
  ▪ Xtend
  ▪ AOP
  ▪ Best Practices
Introduction to Modeling
Different stages
- Creating the metamodel
- Creating a model
- Checking a model
- Transforming a model
- Generating code
Xpand
Xtend
AOP
Best Practices

© Bernd Kolb, Markus Voelter 2008
Eclipse Modeling is the umbrella project for all things modeling that happen on the Eclipse platform:

The Eclipse Modeling Project focuses on the evolution and promotion of model-based development technologies within the Eclipse community by providing a unified set of modeling frameworks, tooling, and standards implementations.

- Eclipse Modeling is not formally related to OMG, but implements several of their standards.
- It is fair to say that many leading edge modeling tools are hosted/developed at Eclipse Modeling.
- Everything Open Source under the Eclipse Public License

© Bernd Kolb, Markus Voelter 2008
The **Eclipse Modeling Framework** (EMF) serves as the foundation: It provides the **Ecore Metametamodel** and frameworks and tools around it for tasks such as

- Editing
- Transactions
- Validation
- Query
- Distribution/Persistence (CDO, Net4j, Teneo)
The Graphical Modeling Framework (GMF) is used for building custom graphical editors based on meta models defined via EMF.
- It is currently in version 2.x
- Proven technology, used in many industrial-strength systems
- Based on Eclipse GEF

The Textual Modeling Framework is used for building custom textual editors.
- Project is currently being set up
- Will be populated initially from oAW Xtext and INRIA TCS.
- **UML 2.x**: An implementation of the UML 2 meta model based on EMF
  - UML 2 Tools: GMF editors for the UML 2 models

- **OCL**: APIs for OCL expression syntax for implementing queries and constraints.

- **EODM**: implementation of the Ontology Definition Metamodel (ODM) on EMF with tooling around it
- **M2M (Model-to-Model)**: delivers an extensible framework for model-to-model transformation languages, with an exemplary implementation of the QVT Core language.
  - **ATL**: M2M language from INRIA

- **M2T (Model-to-Text)**: focuses on technologies for transforming models into text (code generation)
  - **JET**: provides code generation framework & facilities that are used by EMF.
  - **Xpand**: oAW’s code generation engine, part of M2T

© Bernd Kolb, Markus Voelter 2008
 EMP GMT openArchitectureWare: An integrated toolkit for MDSD

- **Version 4.2.1** is current
- **Proven track record** in various domains & project contexts
  - e.g., telcos, internet, enterprise, embedded realtime, finance, ...
- **www.openarchitectureware.org** and **eclipse.org/gmt/oaw**
- Integration with Eclipse:
  - Uses EMF as a basis
  - Graphical editors based on GMF
  - All editors and tooling based on Eclipse
- Supplier of components for Eclipse Modeling:
  - Xpand, Xtend, Check → M2T, Xtext → TMF

© Bernd Kolb, Markus Voelter 2008
EMP GMT openArchitectureWare II: Overview on a Slide

1. Model Verification using Constraints
2. Integrating Generating Code with Manually Written Code
3. Code Generation
4. Model Modification/Completion
5. Model-to-Model Transformation
6. Loading/Storing Models
7. Model Editing based on Custom-built Textual Editors
8. Model Editing using UML Tools
9. Model Editing based on Custom-built Graphical Editors (based on Eclipse GMF, not on openArchitectureWare)
Introduction to Modeling

Different stages
- Creating the metamodel
- Creating a model
- Checking a model
- Transforming a model
- Generating code

Xpand
Xtend
AOP
Best Practices
There are several ways of developing the meta model.

**EMF**
- EMOF implementation
- Roughly similar to the capabilities of UML Class Diagram
- No bidirectional associations, only unidir. references

**UML2**
- Reference implementation of the OMG’s UML2 spec
- UML2 Class Models can be transformed into an EMF meta model

**GMF Ecore demo editor / Ecore Tools**

**Xtext Grammar**
EMF’s **default reflective editor** can be used out of the box to create instances of any meta model

- Generic,
- But not very usable or scalable
Different stages – creating the model (II)

- **TMF – The Textual Modeling Framework**
  - Textual Syntax can be defined for a meta model
  - Textual models are parsed into instances of this meta model
Different stages – creating the model (III)

- GMF – The Graphical Modeling Framework
- Define custom graphical editors for your meta model
- Use the editor to “draw” instances of the meta model
Validation

- Validation is very important!
  - Constrain the models over what Ecore can express directly
  - Fail early in the development process – more semantics!
  - Report errors and warnings to the user
  - Only run transformations and generators on a valid model

- OCL
  - Implementation for OMG’s spec
  - No nice editor yet

- Check
  - Part of M2T’s Xpand
  - OCL like with some useful extensions
Model Transformations are used to
- Transform a model to an instance of another metamodel
- To modify (enrich or simplify) the existing model

Eclipse’ solutions to that problem are
- ATL – the Atlas Transformation Language
- QVT (Operational) – An implementation of OMG’s spec.
- Xtend – oAW’s transformation language
  (also used for other tasks within oAW)
Typically you want to “execute” your model
- Build an interpreter
- Build a generator

M2T contains 3 Template languages
- Jet – Java Emitter Templates
- (MOF Script) – Just created, not yet finished. OMG spec
- Xpand – oAW Template Language

The rest of the talk focusses on Xpand
Introduction to Modeling
Different stages
  - Creating the metamodel
  - Creating a model
  - Checking a model
  - Transforming a model
  - Generating code

- Xpand
- Xtend
- AOP
- Best Practices

© Bernd Kolb, Markus Voelter 2008
Xpand

- Designed specifically for **code** (actually, text) **generation**
- **OO template engine** supporting template polymorphism
- French quotation marks as **escape characters**
- **Embedded Expressions** based on Xtend (OCL-like)
- Support for **Recursion**
- Support for Template Variability using **AOP**
- Metamodel-aware **Editor** with Code Completion and Syntax Highlighting
- **Debugger**
- **Import**: make meta models visible to template file
- **Define**: define a new template
- **File**: open a file into which output is written
- **Expand**: Call another template for one or several elements
- **Foreach**: iterate over coll., generate code for each element
- **If**: conditional template code
- **Extension**: import an extension file for use in the templates
- **Error**: report an error
- **Let**: define a temporary variable
- **Protected** regions: well, you should not use this... 😊
- **Rem**: define comments
- **Around**: advice other templates (AO)
Xpand Editor knows your metamodel (code completion, validation)

You can work on different types of meta-metamodels
- EMF, UML2, Java, ...

When writing a template you have to specify the metamodel you are using
- This is done using the import syntax
- At the moment you are importing packages using their name
  - In the future you’ll be able to use a NS_URI like notation
- A **Template file** contains any number of **templates**
- A template is specified using the **DEFINE** keyword
- A template has a **name** and is defined for a certain **type** (from your meta model)
- It is possible to have **several templates with the same name** but different types (i.e. Polymorphism if those are subtypes)
- Optionally, a template may have **additional parameters**.
  - **Polymorphic dispatch** over all parameters (multi methods)
The `FILE` statement is used to specify the output file where the contents of a template and its called templates goes to.

The filename might be an expression.
- **EXPAND** is used to **invoke another template**

- You have to specify the **target object**
  (if called for this, the **FOR** part can be omitted)

- It is also possible to call a template **FOREACH** element in a **collection** (support for **SEPARATOR**)
- **FOREACH** is used to iterate over a collection
- The current element from the list is assigned to a **local variable** (AS-part)
- Optionally you may specify a **SEPARATOR** and **ITERATOR**

```java
// A constant for each state in the statemachine
<FOREACH relevantStates() AS t ITERATOR i>«
public static final int <s.constantName()> = <i.counter0>;
<ENDFOREACH>

// A constant for each transition the
<FOREACH states.transitions AS t ITERATOR
public static final int <t.constant>
<ENDFOREACH>

// A constant for each event the statemachine
```
- *IF* is used to add **conditions** to your template
- You may have any number of *ELSE IF* statements
- And one *ELSE*
- **LET** is used to define a local variable (assign once)
- Not used very often – better use an extension

```xml
<DEFINE jumper FOR RecordHandler->
  <IF jump != null>
    <LET jump.nextRecord AS next>
      <next.handlerMethodName()>{ current, rest };
    </LET>
    <ELSE>
      return;
    </ELSE>
  </IF>
</DEFINE>
```
- **REM** starts a comment (**ENDREM** ends it)
- There is **no single line comment**
- Used to **stop the code generation** and report an error to the user
  - E.g., used to report an “abstract” template call
Xpand supports **protected regions**
- However other solutions are more suitable (see later)
- Some files (e.g. the Manifest) do not support PRs

Xpand marks start of protected region via a **comment**

A **unique ID** is used to identify a protected region

You have to tell the generator which folders should be considered when searching for “old” PRs

Content in the PR acts as **default implementation**
- It is good practice to **keep a generator simple**
  - Complex (and/or reused) expression should not be put into templates

- Xpand can **delegate these parts to another language**
  - Xtend is used for this purpose
  - You have to import the extension files you’d like to use

- oAW comes with a set of **utility extensions**. They can be found in the `stdlib`
Finally you have to **execute** the generator

**Option 1 – From Java Code:** Creating an instance of
`org.openarchitectureware.xpand2.Generator / org.eclipse.xpand2.Generator`

**Option 2: Using the Workflow** (Recommended)
Contents

- Introduction to Modeling
- Different stages
  - Creating the metamodel
  - Creating a model
  - Checking a model
  - Transforming a model
  - Generating code
- Xpand
- Xtend
- AOP
- Recipes

© Bernd Kolb, Markus Voelter 2008
- It is good practice to **keep a generator simple**
  - Complex (and/or reused) expression should not be put into templates

- Xpand can **delegate these parts to another language**
  - Xtend is used for this purpose
  - You have to import the extension files you’d like to use

- oAW comes with a set of **utility extensions**. They can be found in the **stdlib**
- **OCL-like expression language** used throughout oAW
  - Can be used in constraint checks, model transformations and generators
  - **Add “methods”** to meta types (Java calls are possible if necessary)
  - **path** expressions, **set** operations, (some) **higher order** functions
  - **Polymorphism** (multiple dispatch)
  - **Tool support** (syntax highlighting, code completion, debugger)

```java
import imp;

extension dataimport::dsl::GenExtensions reexport;

Collection[Instance] instancesInScope| RecordHandler this :
  parentHandler() != null ? instances.union( parentHandler()).instancesInScope() : instances;

parentHandler( RecordHandler this ) :
  eContainer != null ? ( RecordHandler.isInstance(eContainer) ? ((RecordHandler)eContainer) : null ) :
null;
```

© Bernd Kolb, Markus Voelter 2008
CONTENTS

- Introduction to Modeling
- Different stages
  - Creating the metamodel
  - Creating a model
  - Checking a model
  - Transforming a model
  - Generating code
- Xpand
- Xtend
- AOP
- Best Practices
AOP is an important mechanism to
- adapt existing transformations and generators by “advising” before/after/around existing templates non-invasively
- Define variants of generators in the context of PLE.

Aspects can be added to **Xtend** functions and **Xpand** templates.
We do **not** want to **change** the **original workflow**
- However we have to tell the generator where to find **aspects**
- Therefore a workflow component can have an **ID**
- To this **ID** you can contribute **advices**
CONTENTS

- Introduction to Modeling
- Different stages
  - Creating the metamodel
  - Creating a model
  - Checking a model
  - Transforming a model
  - Generating code
- Xpand
- Xtend
- AOP
- Best Practices

© Bernd Kolb, Markus Voelter 2008
- 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.
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)
Recipes can be arranged hierarchically.

This is a failed check.

“Green” ones can also be hidden.

Here you can see additional information about the selected recipe.
Adding the extends clause makes all of them green.
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 recommend to generate as much stuff as possible.
  - You should use a RICH DOMAIN-SPECIFIC PLATFORM,
  - And SELECT FROM BUILD, BUY OR OPEN SOURCE

© Bernd Kolb, Markus Voelter 2008
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")
It is important that the templates are **nice and readable**

However the **generated code** is important, too

By default a Xpand **statement** would result in **a new line**
  - This can be prevented by either
    - setting an option in the generator
    - Adding a hyphen at the end of a statement

Xpand has support for **post processors**
  - A simple interface
  - Formatters available for Java, XML (and C/C++)
Thanks for your attention!

Questions?

Bernd Kolb
b.kolb@kolbware.de
http://www.kolbware.de

Markus Voelter
voelter@acm.org
http://www.voelter.de

© Bernd Kolb, Markus Voelter 2008