EclipseCon 2008

Modeling Tutorial

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

Markus Voelter
voelter@acm.org
www.voelter.de
EclipseCon 2008

Setting up your machine

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

Markus Voelter  
voelter@acm.org  
www.voelter.de
Setting up your machine

- Extract your Eclipse
- Copy the links folder into the Eclipse directory
- Extract the lib.zip parallel to your Eclipse installation
- Start Eclipse
- Import the workspace.zip using the Import-Wizard
EclipseCon 2008

Modeling Tutorial

Intro to EMP

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

Markus Voelter
voelter@acm.org
www.voelter.de
Eclipse Modeling in a Nutshell

- 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.

© Bernd Kolb, Markus Voelter 2008
- **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.

- **XSD Infoset**: reference library that provides an API for use with any code that examines, creates or modifies W3C XML Schema.

© Bernd Kolb, Markus Voelter 2008
- **M2M (Model-to-Model)**: delivers an extensible framework for model-to-model transformation languages.
  - **ATL**: M2M language from INRIA
  - **QVT** implementations

- **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, to be part of M2T in the Ganymede release
An integrated, tested, one-stop toolkit for MDSD

Version 4.2.1 is current (4.3 beta for Xtext)

Proven track record in various domains & project contexts

Stable, productive and helpful developer, support and user communities

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

© Bernd Kolb, Markus Voelter 2008
EclipseCon 2008

Modeling Tutorial
Intro to MDSD

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

Markus Voelter
voelter@acm.org
www.voelter.de
Domain Driven 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.
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.
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.
• domain-experts to formally specify their knowledge
• different implementations
• capture knowledge about the domain, uncluttered.
• don’t want to bother with implementation details
• fan-out
• product lines and software system families
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 Manufacturing”, Software System Families

- Language-Oriented Programming:
  Focus on DSLs instead of Frameworks, incl. Editor/Debugger Support
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**

**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 (III)

- 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 (II)

- GMF – The Graphical Modeling Framework
- Define custom graphical editors for your meta model
- Use the editor to “draw” instances of the meta model
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
EclipseCon 2008

Modeling Tutorial

Code Generation

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

Markus Voelter
voelter@acm.org
www.voelter.de
Typically 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
- Designed for code (actually, text) generation only
- **OO template engine** supporting template polymorphism
- **French quotation marks** as escape characters
- Embedded **expressions** based on Xtende
- Support for **Recursion**
- Support for **Template Variability using AOP**
- **Metamodel aware Editor** with Code Completion and Syntax Highlighting
- Debugger
Xtend:

oAW Expression Language

- 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)
Thanks for your attention!

Questions?

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

Markus Voelter  
voelter@acm.org  
http://www.voelter.de
EclipseCon 2008

Modeling Tutorial

Workflow Engine

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

Markus Voelter
voelter@acm.org
www.voelter.de
A language for describing **sequential workflows**

It was designed to bind together the different model-processing tools

**Orchestrate the model processing chain**
- Read a model
- Check the model
- Modify the model
- Store the model
- Generate code from that model
What comes with MWE

- The **workflow engine** itself
- **Basic components** for reading and writing EMF models
- **API** for you to **integrate** your project with MWE

**Tool support**
- Editor with code completion (using reflection) and outline
- Run as...
- Ant task

**Debugger support**
- Debug as...
- Simple API to add a debugger for your project
public interface WorkflowComponent {

    public void checkConfiguration(Issues issues);

    public void invoke(WorkflowContext ctx, ProgressMonitor m, Issues i);

}

- **checkConfiguration**: validate the configuration of the workflow component; report issues if there are any

- **invoke**: execute your component’s behaviour
  - **WorkflowContext ctx**: access workflow state via
  - **ProgressMonitor m**: report progress
  - **Issues i**: report issues
<xml version="1.0"?><workflow>
  <component class="org.eclipse.emf.mwe.utils.Reader">
    <uri value="platform:/resource/project-name/pathToFile.xmi"/>
    <modelSlot value="mySlot"/>
  </component>

  <component class="org.eclipse.emf.mwe.utils.Writer"
            uri="platform:/resource/project-name/pathToOutputFile.xmi"
            modelSlot="mySlot"/>
</workflow>

public class Reader {
    //...
    public void setUri(String uri) {
        //...
    }
    //...
}
Why textual, external DSLs?

- A big barrier to adoption of DSLs is **tool integration**: how to integrate tool that stores its data in repositories?
- Textual DSLs completely removes this problem:
  - Human-readable models are stored, versioned, searched and diffed **like any other text file**, integrating with CVS and SVN.
  - Also, we know from working with traditional source code that **text is a useful representation** for even large systems.
- The big advantage compared to internal DSLs (in Ruby, Groovy) is that you can have **DSL-specific constraint checks** and **language support** in editors

→ This talk is about how leverage these advantages
Typically textual DSLs are build using one of the many parser generators (ANTLR, JavaCC, Lex/yacc), or even by a hand-writing a custom parser.

Parser: match text and try to create a parse tree

Parser Generator: Generate a parser from a grammar
Typically, transformed into an **Abstract Syntax Tree (AST)**
- No whitespace, reified nodes, often binary tree

The AST can be considered the **model**
- The node types of the AST act as the **metamodel**

Programs processing the “sentence” are typically written against the AST – they usually don’t care about the parse tree
Writing a parser is **non-trivial**

Using a *parser generator* makes it easier, but still not for everybody

Also: out of the box, a parser generator **only creates a matcher** and/or a **simplistic AST**. You still need to
- Transform the model into a form that is easily processable
- Create an editor with syntax highlighting, code completion, etc.

→ **A lot of work!** Only few people are willing to do that
Xtext makes this much more feasible.

Based on an EBNF grammar it generates
- ANTLR3-based parser
- EMF-based metamodel
- Eclipse editor with
  - syntax highlighting
  - code completion
  - customizable outline
  - code folding
  - real-time constraint checking

Xtext is part of openArchitectureWare / Eclipse Modeling
Thanks for your attention!

Questions?

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

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