Open Source Modeling

Textual DSLs with oAW Xtext

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

Bernd Kolb
b.kolb@kolbware.de
www.kolbware.de

Peter Friese
peter.friese@itemis.de
oaw.itemis.eu

Copyright is held by the authors/owners. – OOPSLA 2008, October 19–23, 2008, Nashville, Tennessee, USA. – ACM 978-1-60558-220-7/08/10.

About us

- **Markus Völter**

  Markus Völter works as an independent consultant and coach for software technology and engineering. He focuses on software architecture, model-driven software development and domain specific languages as well as on product line engineering. Markus is (co-) author of many magazine articles, patterns and books on middleware and model-driven software development. He is a regular speaker at conferences world wide. Markus can be reached at voelter@acm.org or via www.voelter.de

- **Bernd Kolb**

  Bernd Kolb focuses on model-driven software development and Eclipse technologies. As a consultant, he has worked in different domains from tooling for automotive embedded systems to enterprise Java applications. Bernd is an active committer for the open source projects openArchitectureWare and Eclipse. He is a regular speaker at conferences and has written a number of articles as well as co-authored a book. He can be reached via b.kolb@kolbware.de or at www.kolbware.de.

- **Peter Friese**

  Peter Friese is a software architect for itemis in Kiel. He is a committer for the open source projects openArchitectureWare, Eclipse, FindBugs, and AndroMDA. As a software engineer and software architect, Peter has worked on a variety of industry projects in different domains such as banking, aerospace and transport. Peter is a regular speaker at various conferences and has written a number of articles on topics such as Eclipse, Spring, and model-driven software development. He can be reach at peter.friese@temis.de or via www.peterfriese.de

© Bernd Kolb, Markus Voelter, Peter Friese 2008

CONTENTS

© Bernd Kolb, Markus Voelter, Peter Friese 2008

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, Peter Friese 2008

EMP Core: EMF

- 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)

© Bernd Kolb, Markus Voelter, Peter Friese 2008

EMP Concrete Syntax: GMF and TMF

- 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, Peter Friese 2008

EMP Concrete Syntax: Model Transformation Tools

- **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

© Bernd Kolb, Markus Voelter, Peter Friese 2008

EMP GMT openArchitectureWare: An integrated toolkit for MDSD

- 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

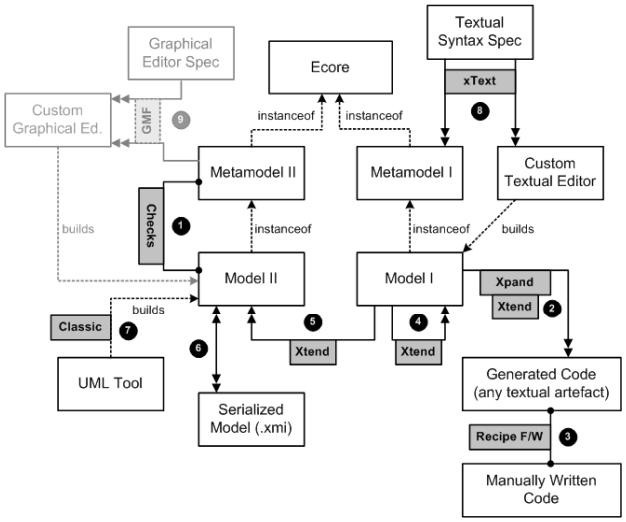
© Bernd Kolb, Markus Voelter, Peter Friese 2008

EMP GMT openArchitectureWare: oAW 5

- **Version 5** currently under development:
- **Migration** of oAW Components to Eclipse Modeling proper:
  - Xpand, Xtend, Check → Model-to-Text (M2T)
  - Xtext → Textual Modeling Framework (TMF)
  - Workflow → Modeling Workflow Engine (MWE)
- **More features**
  - New **backend:** a VM for M2T (and maybe more languages)
  - **Mixing** Languages: Xpand, Xtend, Check in one file
  - Xtext: Grammar **Fragments** and the stuff from the 4.3 beta
  - **Generic** Tree/Form Editor, mixing editors
  - Wizard Framework

© Bernd Kolb, Markus Voelter, Peter Friese 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)

© Bernd Kolb, Markus Voelter, Peter Friese 2008

CONTENTS

© Bernd Kolb, Markus Voelter, Peter Friese 2008

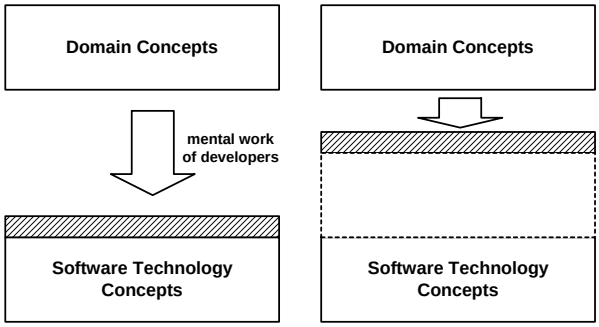
Domain Specific Languages

- A Domain Specific Language (DSL) is a **focused, processable language** for describing a specific **concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.
- DSLs can be **classified** in many ways:
  - Configuration vs. Customization
  - Internal vs. External
  - Graphical vs.textual
- We concentrate on **external textual customization** DSLs

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Core Concepts

- DSLs are 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**.



© Bernd Kolb, Markus Voelter, Peter Friese 2008

DSL Categories

- There is a large variety of DSL flavours.
- DLS can be **categorized** along the following dimensions:
  - Domain Selection
  - Expressiveness
  - Concrete Syntax
  - Execution
  - Integration
  - Tool Support

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Why DSLs are useful

- Aspects/Concerns/Viewpoints of a system can be **modeled formally**, independent of others
  (avoiding the legacy system problem)
- Formal models can be **processed automatically** by tools (analysis/check, transformation, generation/Interpretation) potentially leading to
  - **technology independence** of implementation code
  - More **efficient** development
  - More **reusable** systems
  - Better software **quality**

© Bernd Kolb, Markus Voelter, Peter Friese 2008

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-specfic constraint checks** and **language support** in editors

  → **This talk is about how leverage these advantages**

© Bernd Kolb, Markus Voelter, Peter Friese 2008

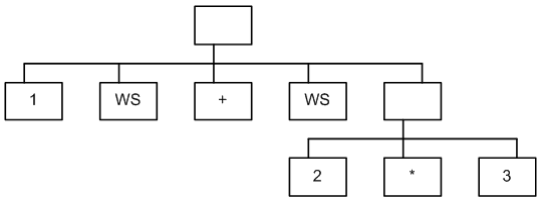
Why maybe not?

- Sometimes it is useful to **show things graphically**: Dependency Structures, Temporal Aspects, Signal Flow
- In this case there are several **alternatives**:
  - **Visualize** a subset of the model, e.g. using graphviz or prefuse (typically read-only, but superb auto layouting)
  - Use **different viewpoints**, some of them using graphical editor (requires cross-refs between textual and graphical models)
  - Edit the same model **textually and graphically** (textual format is used as the serialization format for the graphical editor; requires writeability of textual model!)

© Bernd Kolb, Markus Voelter, Peter Friese 2008

How it works in general (I)

- 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**

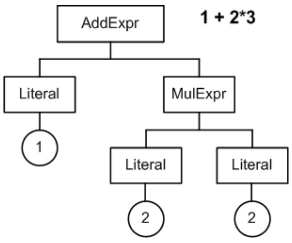


- **Parser Generator:** Generate a parser from a **grammar**

© Bernd Kolb, Markus Voelter, Peter Friese 2008

How it works in general (II)

- 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

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Challenges

- 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

© Bernd Kolb, Markus Voelter, Peter Friese 2008

CONTENTS

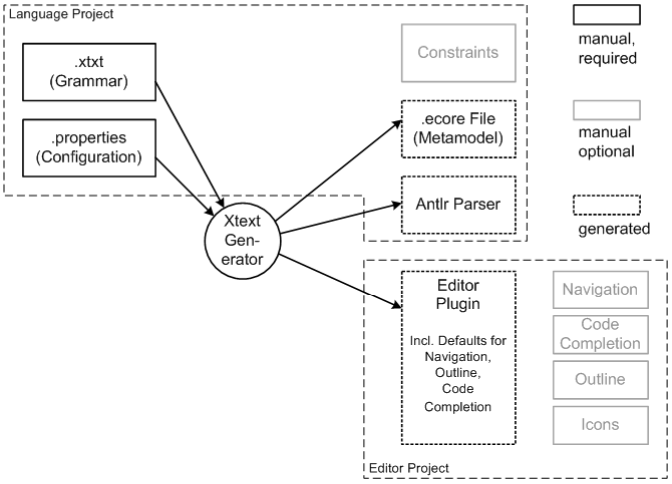
© Bernd Kolb, Markus Voelter, Peter Friese 2008

Xtext

- 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**

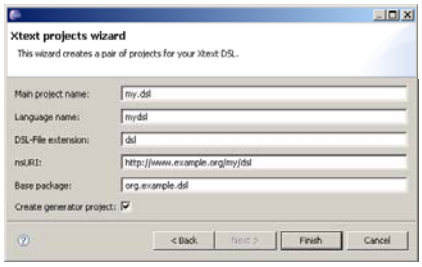
© Bernd Kolb, Markus Voelter, Peter Friese 2008

Xtext Overall Process



© Bernd Kolb, Markus Voelter, Peter Friese 2008

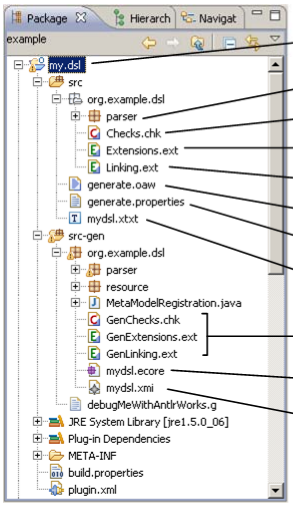
Creating a new Xtext Project I: Xtext Project Wizard



- **Main Project Name:**
  - name of language project
  - *.editor* project contains editor
- **Language Name:** package name of generated meta model
- **DSL File Extension:** file extension used to associate editor
- **nsURI:** full namespace URI of the generated meta model
- **Base Package:** all code will be in or below this package
- **Code Gen Project:** See last section of this presentation

© Bernd Kolb, Markus Voelter, Peter Friese 2008

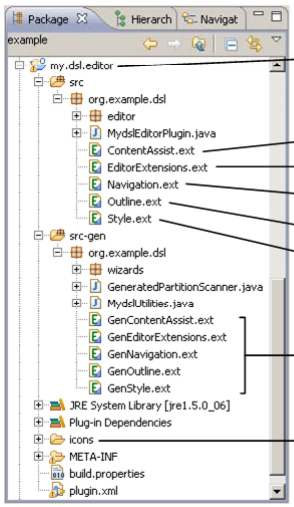
Creating a new Xtext Project II: Created Language Project



- **Language Project**
- Contains a **parser** to be used in standalone programs
- Put your language's **constraint checks** here
- This contains your metamodel's **extension** functions
- Define custom **linking** functionality here
- Run this to **generate** metamodel, editor, and everything else
- Contains a number of **properties** (e.g. those you put into the wizard)
- Specify your **grammar** rules here
- Generated „base files" - don't change.
- Generated **metamodel**
- Generated **Xtext Rules** file

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Creating a new Xtext Project III: Created Editor Project



- **Editor Project**
- Customize **Code Completion** here
- Customize **outline labels** and **icons** here
- Define **go to definition** and **find references** here
- Provide customized **outline structure** and **viewpoints** here
- Customize **keyword font style** here
- Generated „base files" - don't change.
- Put **customized icons** (for the outline view) here

© Bernd Kolb, Markus Voelter, Peter Friese 2008

CONTENTS

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Intro to Example I: Problem

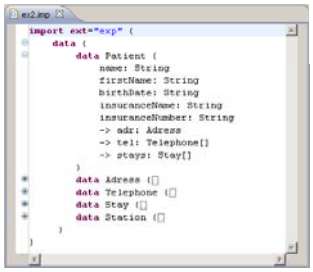
- DSL for **importing data:** common problem in large enterprises (e.g. hospitals)
- **Problem:** read data like the one on the right into Java Beans
  - Example is about patients, stays, wards, insurance data
- **Record Structure:** <Type> <Fd1>,<Fd2>, … <Fdn>
- **Different Record Types** have different fields

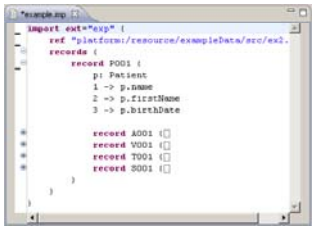
```
P001 Voelter,Markus,*14.02.1974
A001 Grabenstrasse 4, 73033, Goeppingen
V001 Futuristic Insurance,123456767
T001 49,07161,12334454
T001 49,171,8601869
S001 01.02.2004,IN25
S002 03.02.2004,CH52
S003 05.02.2004
S001 22.03.2005,NE02
S002 25.03.2005,NE04
S002 27.03.2005,NE05
S003 28.03.2004
P001 Somebody,Else,*23.05.1960
A001 Ziegelaecker 11, 89520, Heidenheim
V001 Some Other Insurance GmbH,124345
T001 49,07321,973344
S001 23.02.1999,IN25
S003 02.03.1999
P001 Person,Else,*23.05.1960
A001 Ziegelaecker 11, 89520, Heidenheim
V001 Some Other Insurance GmbH,124345
T001 49,07321,973344
S001 23.02.1999,IN25
S003 02.03.1999
```

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Intro to Example II: Solution Approach

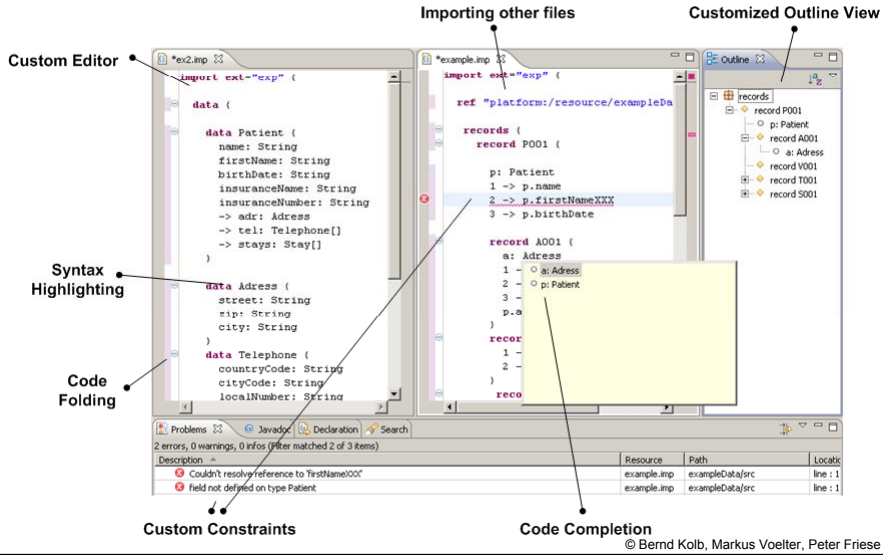
- Define **data structures** (à la Java Beans)
  - Entities
  - Attributes, References (:1, :n)
- Define **parser**
  - Checks record order
  - Instantiates data structures
  - Maps fields to attributes
  - Assembles references
- Backend: **Generate** Java code for **beans** as well as **parser**





© Bernd Kolb, Markus Voelter, Peter Friese 2008

Intro to Example III: Solution Tooling



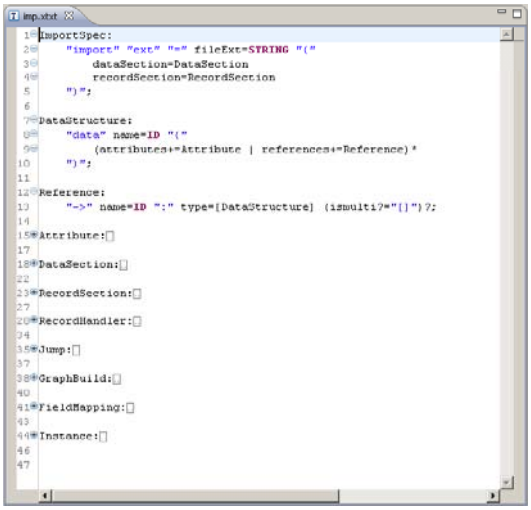
© Bernd Kolb, Markus Voelter, Peter Friese 2008

CONTENTS

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Defining The Grammar

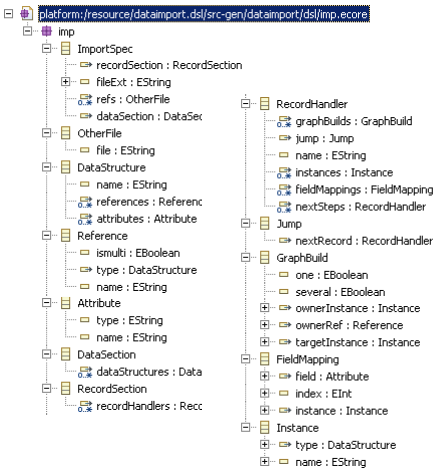
- Grammar is defined in an **EBNF-like format** in the **Xtext editor.**
- The editor provides **code completion** and **constraint checking** for the **grammars** themselves
- Grammar is a collection of **Rules**. Rules start with their **name** followed by **:** and ending with **;**



© Bernd Kolb, Markus Voelter, Peter Friese 2008

The derived EMF meta model

- The Xtext generator builds an **EMF meta model** from the grammar.
- The generated **parser** then **instantiates** the meta model from a model.
- Processing Xtext models is based on this meta model: **all existing EMF tooling** is suitable for processing.
  - We'll show a code gen template later



© Bernd Kolb, Markus Voelter, Peter Friese 2008

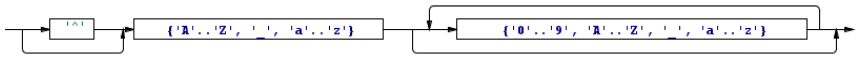
Different Kinds of Rules

- **Type Rule**
  - For each **rule** Xtext creates a **class in the metamodel**
  - Each rule **property** results in a **property of the metaclass**
- **String Rule**
  - String rules are parsed to a String.
  - They are in effect custom lexer rules (recognizing string patterns)
- **Enum Rule**
  - Limited set of alternatives; mapped to an Ecore enum
- **Native Rule**
  - A lexer rule is directly interpreted by ANTLR; mapped to a String

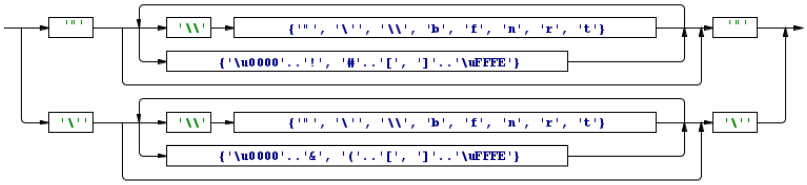
© Bernd Kolb, Markus Voelter, Peter Friese 2008

Built-in Lexer Types (I)

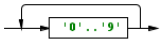
- ID



- STRING



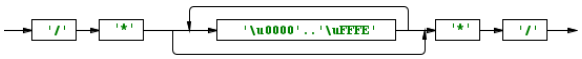
- INT



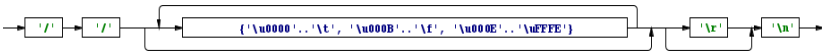
© Bernd Kolb, Markus Voelter, Peter Friese 2008

Built-in Lexer Types (II)

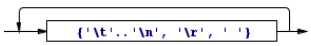
- Multiline comment



- Single line comment



- Whitespace

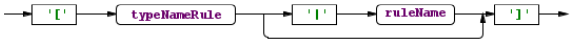


- The content of those rules is **not transformed into the meta model**

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Built-in Reference Types

- **Reference**



- **File Reference/Import:** Xtext derives the default Linking information from here (more later)

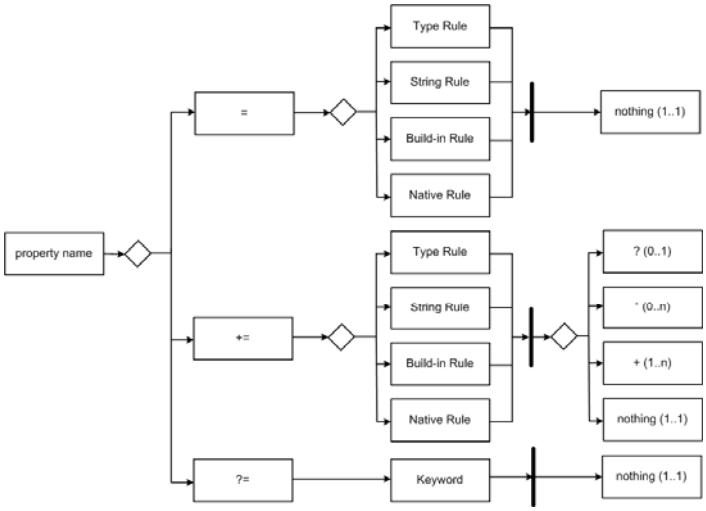
© Bernd Kolb, Markus Voelter, Peter Friese 2008

Type Rules

- A **type rule** is mapped to a metaclass in the metamodel
- It may contain **keywords** (using string literal syntax)
- It also contains **properties** which will result in properties of the respective metaclass
  - The property type is derived from the called rule
- There are different **kinds of properties**
  - = (single assign)
  - += (multiple assign/add)
  - ?= (boolean assign)
- There are different **property cardinalities**
  - ? (0..1)
  - * (0..n)
  - + (1..n)
  - nothing (1..1)

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Type Rules – Combinations



© Bernd Kolb, Markus Voelter, Peter Friese 2008

Type Rules – Examples

- ImportSpec:
  - **Keywords**: *import*, *ext*, =, { and }
  - Expects **one occurrence** of *DataSection* rule assigned to the *dataSection* property

imp.xtxt

```
ImportSpec:
    "import" "ext" "=" fileExt=STRING "("
        dataSection=DataSection
        recordSection=RecordSection
    ")";

DataStructure:
    "data" name=ID "("
        (attributes+=Attribute | references+=Reference)*
    ")";

Reference:
    "->" name=ID ":" type=[DataStructure] (ismulti?="[]")?;
```

- DataStructure:
  - Expects **any number** of *Attribute* **or** *Reference* occurences and **assigns them** to the *attributes* and *references* property, respectively

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Reference Rule

- By default**, execution of rules results in a tree** of data
- You can **reference** other elements via a **reference rule**
  - In textual languages a **reference** has to be **by name**
  - During **linking**, Xtext "dereferences" these by name-references
  - Target's *name* property is used here (customization: see later)
- Example: reference to a *DataStructure*. *ID* is is the lexer used for the referencing token

data.xtxt

```
DataStructure:
    "data" name=ID "{"
        (attributes+=Attribute | references+=Reference)*
    "}";

Reference:
    "->" name=ID ":" type=[DataStructure] (ismulti?="[]")?;

Attribute:
    name=ID ":" type=ID;
```

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Resource References

- A resource reference can be used to **import another model file** (typically of the same DSL)
- When customizing e.g. content assist**, Xtext automatically provides a global view** on the contents of all imported files

imp.xtxt

```
importGrammar "platform:/resource/dataimport.ds

ImportSpec:
 "import" "ext" "=" fileExt=STRING "{"
    (refs += OtherFile)*
    dataSection=DataSection
    recordSection=RecordSection
 "}";

OtherFile:
 "ref" file=URI;
```

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Abstract Type Rules

- A type rule can be abstract
- An abstract type rule is basically a **collection of OR-ed alternatives**: R1 | R2 | R3
- Mapped to an **abstract metaclass**
  - The or-ed alternatives become concrete subclasses
  - Common properties of the alternatives are lifted into the abstract superclass
- The same rule can be **or-ed in several abstract rules** in which case the metaclass has **several superclasses**.

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Abstract Type Rules – Examples

- *Property* is the **abstract rule**.
- Can be a *Reference* **or** an *Attribute*
- The *DataStructure* has a **collection of *Properties***, referencing the abstract rule
- In the meta model, *name* has been be **lifted**, *type* has not

imp.xtxt

```
7  DataStructure:
8      "data" name=ID "{"
9          (properties+=Property)*
10     "}";
11
12 Property:
13     Reference | Attribute;
14
15 Reference:
16     "->" name=ID ":" type=[DataStructure] (ismulti?="[]")?;
17
18 Attribute:
19     name=ID ":" type=ID;
```

imp.xtxt | imp.ecore

- platform:/resource/dataimport.dsl/src-gen/dataimport/dsl/imp.ecore
  - imp
    - ImportSpec
    - DataStructure
      - properties : Property
      - name : EString
    - Property
      - name : EString
    - Reference -> Property
    - Attribute -> Property

© Bernd Kolb, Markus Voelter, Peter Friese 2008

String Rules

- String rules are **lexer rules** (i.e. used to define rules for custom character sequences)
- Declared via the *String* keyword

```
String qualifiedName:
    ID ("." ID)*;
```

- String rules can be **composed** from other lexer rules.
  - Other String rules
  - Built-in lexer rules (e.g. ID)
  - Native rules (see Xtext Reference Manual)

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Enum Rules

- A Enum Rule is used to define a **limited set of defined alternatives**
- It is **mapped to an Enum** in the metamodel
- It is declared via the *Enum* keyword and contains **Enum Literals**
  - An Enum Literal has a token name and a string representation

```
Enum RecordType:
    A001="a0001" | A002="a002" | P001="p001";
```

- It can be used just like any other rule.

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Native Rules

- A **native rule** contains a string which is **passed to ANTLR** without further processing it.
- It is typically used to **define lexer rules** that cannot be expressed using Xtext syntax
  - E.g. whitespace-aware lexer rules, such as define custom comment syntax
- It is declared via the *Native* keyword
- The example introduces a comment (# till end of line) that is preserved into the model

imp.xtxt

```
ImportSpec:
    (comment=SL_COMMENT)?
    "import" "ext" "=" fileExt=STRING "{"
        dataSection=DataSection
        recordSection=RecordSection
    "}";

Native SL_COMMENT :
  "'#' ~('\n'|'\r')* '\r'? '\n'";
```

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Parsers for existing Metamodels (I)

- As a default, Xtext is used to define grammars and **derive the meta model** (the ecore file) from this grammar.
  - You start with the concrete syntax.
- However, sometimes you want define a textual **grammar for an existing meta model**.
- To do this, you have to **import** the existing meta model:

```
importMetamodel "http://www.kolbware.de/oaw/xtext/importExample" as import;
```

- You can then **associate the rules** with an exsiting meta class:

```
RuleForExistingClass [import::AnotherClass] :
    "body" // here goes the rulebody as you know it
;
```

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Parsers for existing Metamodels (II)

- In case all of your rules refer to meta classes in existing meta models, you **do not need** any **meta model generation**: use *preventMMGeneration*
- One of the imported meta models may be **imported without an alias**.
  - if rule name and meta class name are the same, you do not have to use the explicit meta class reference ([...])

```
preventMMGeneration
importMetamodel "http://www.eclipse.org/emf/2002/Ecore";

EPackage:
    "package" name=ID "pre"":" nsPrefix=ID "uri" ":" nsURI=STRING "{"
        eSubpackages+=EPackage*
        eClassifiers+=EClassifier*
    "}"
;

EClassifier:
    EDataType | EClass;

EDataType [EDataType]:
    EEnum | EDataTypeRule
;

EDataTypeRule [EDataType]:
    serializable?="serializable" "data" name=ID
    "{"
    "}"
;

EEnum:
    serializable?="serializable" "enum" name=ID "{"
        eLiterals+=EEnumLiteral*
    "}"
;

EEnumLiteral:
    name=ID "=" value=INT;

EClass:
    (abstract?="abstract")? (interface?="interface")? "class" name=ID
        ("extends"(eSuperTypes+=[EClass] (","))*)? "{"
        (eStructuralFeatures+=EStructuralFeature)*
    "}"
;
```

© Bernd Kolb, Markus Voelter, Peter Friese 2008

CONTENTS

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Sidebar: Xtend

- 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)

Extensions.ext

```
1 import imp;
2
3 extension dataimport::dsl::GenExtensions reexport;
4
5 Collection[Instance] instancesInScope( RecordHandler this ):
6     parentHandler() != null ? instances.union( parentHandler().instancesInScope() ) : instances;
7
8 parentHandler( RecordHandler this ):
9     eContainer != null ? ( RecordHandler.isInstance(eContainer) ? ((RecordHandler)eContainer) : null ) :
10        null;
11
```

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Constraint Checks

- Uses the **oAW Check language,** based on Xtend
  - Used to validate the static semantics of the models
- Report Warnings and Errors
- Based completely on **the generated meta model,**
  no concrete syntax hassles
- Tool support (syntax highlighting, code completion, ...)

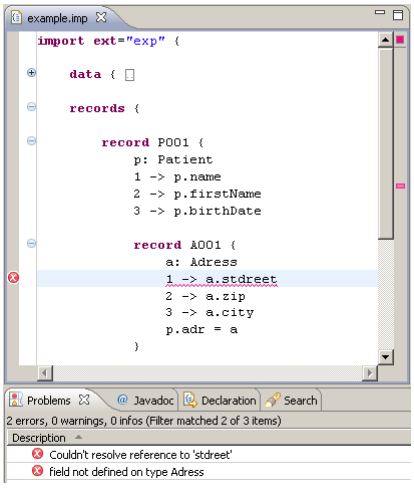
Extensions.ext | *Checks.chk

```
1 import imp;
2
3 extension dataimport::dsl::Extensions;
4
5 context Instance ERROR "type not defined":
6     allElements().typeSelect(DataStructure).contains(type);
7
8 context FieldMapping ERROR "instance not defined or not in scope":
9     ((RecordHandler)eContainer).instancesInScope().contains(instance);
10
11 context FieldMapping ERROR "field not defined on type "+instance.type.name:
12     instance.type.attributes.contains( field );
13
14 context Instance ERROR "instanc names must begin with a lowecase letter":
15     name.toFirstLower() == name;
16
```

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Real Time Constraint Validation

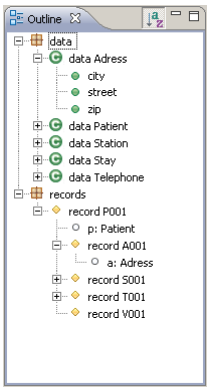
- The **constraints** defined for the language are **evaluated in the editor** on the fly.
- Constraints can also be checked as **part of a work-flow** that processes the models.

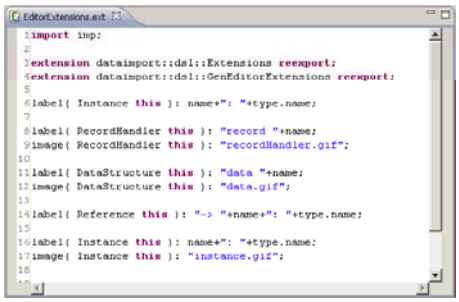


© Bernd Kolb, Markus Voelter, Peter Friese 2008

Customizing the Outline I: Labels and Icons

- Xtext supports the customization of the **Outline View** structure and icons/labels
- Labels and icons are changed by **overriding** *label(...)* and *image(...)* for your meta types.

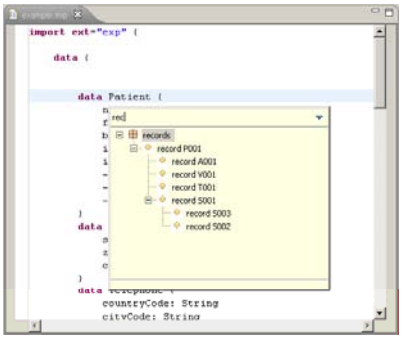




- Actual image files (.gif) go into editor project's *icons* directory

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Customizing the Outline II: Filtering the outline view



- Press Ctrl-O to show the **outline as a pop-up**
- You can then **start typing** to **filter** the visible **elements by name.**
- **Pressing enter** on any element will **navigate** to the respective element.

© Bernd Kolb, Markus Voelter, Peter Friese 2008

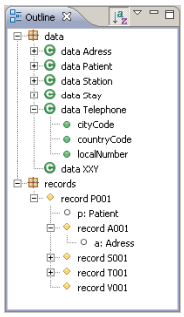
CONTENTS

© Bernd Kolb, Markus Voelter, Peter Friese 2008

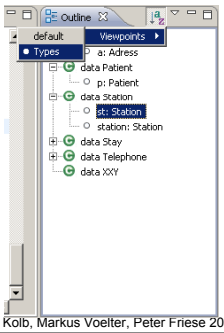
Customizing the Outline II: Structural Customization

- By default, the **structure of the outline** view corresponds to the structure of the document.
  - This default structure can be **changed**
  - **Additional** structures ("viewpoints") can be defined

- **Default** Viewpoint

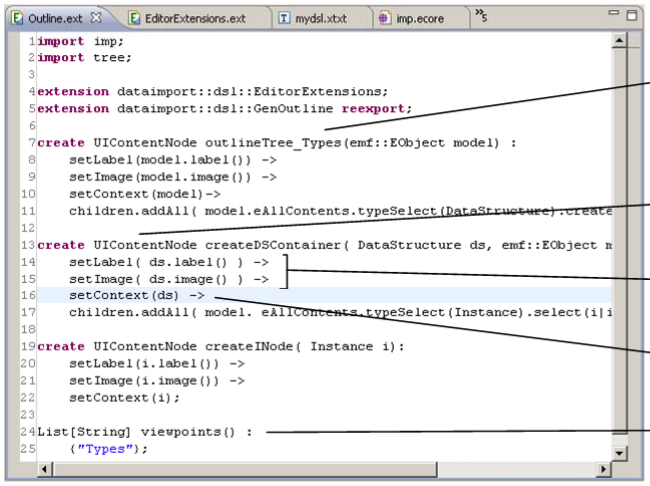


- **Types** Viewpoint (shows data types and instances)



© Bernd Kolb, Markus Voelter, Peter Friese 2008

Customizing the Outline II: Structural Customization II

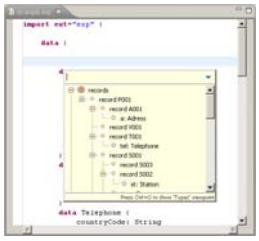


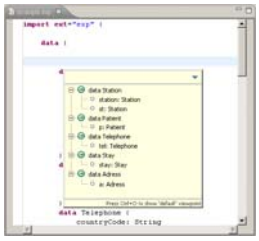
- Use **naming convention** to define a *outlineTree* function for a given viewpoint
- Use *create* extensions creating *UIContentNodes* to **construct the tree**
- Use these functions to **parameterize** the created *UIContentNodes*
- Connects to the original file for **selection highlight**
- Return list of **viewpoint IDs**

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Customizing the Outline III: Popups and Filters revisited

- **Repeatedly** pressing Ctrl-O shows the **sequence of the various viewpoints** in the popup.
  - Ctrl-O once:
  - Ctrl-O again:





© Bernd Kolb, Markus Voelter, Peter Friese 2008

Customizing Content Assist

- Content Assist / Code Completion is customized via **overwriting certain extension functions**:

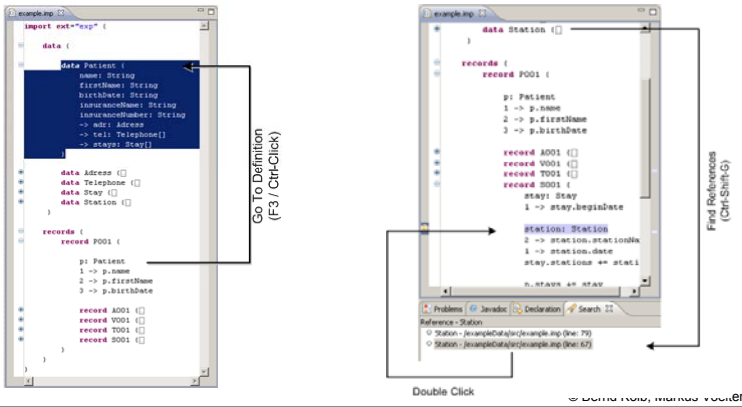
  List[Proposal] complete<MyType>_<propName>
  (emf::Eobject ctx, String prefix)

- A Proposal consists of a **label**, an **icon** and the **text to insert**

```
import imp;
import codeassist;

extension dataimport::dsl::Extensions;
extension dataimport::dsl::GenContentAssist reexport;
extension dataimport::dsl::EditorExtensions;
extension org::openarchitectureware::util::stdlib::io;

List[Proposal] completeRecordHandler(emf::EObject grammarElement, emf::EObject ct
    ( newProposal( "new record...", "record <unnamed> {  }", null ) );

List[Proposal] completeFieldMapping_field(emf::EObject ctx, String prefix) :
    ((FieldMapping)ctx).instance.type.attributes.
        collect(x|newProposal(x.label(),x.id(),x.image()));

List[Proposal] completeFieldMapping_instance(emf::EObject ctx, String prefix) :
    ((RecordHandler)ctx.eContainer).instancesInScope().
        collect(x|newProposal(x.label(),x.id(),x.image()));
```

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Navigation

- By default, if you use references in your DSL, you can use ***Go To Definition*** and ***Find References*** out of the box (it works cross-resource, see next section)
- 

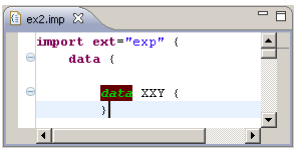
© Bernd Kolb, Markus Voelter, Peter Friese 2008

Navigation II: Customization

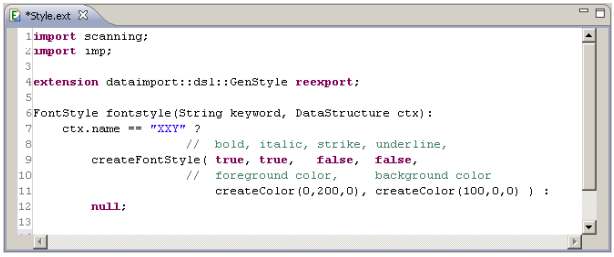
- To customize navigation, **overwrite the following extension** in the *Navigation.ext* file.
  (you do NOT have to do this for DSL references!)
- **Find the Definition** of a "string" in a context:
  emf::EObject findDeclaration(String s, emf::EObject ctx)
- **Find the References** of a "string" in a context:
  List[UIContentNode] findReferences(String s, emf::EObject ctx)
- **Create the node** for the list of results:
  create UIContentNode createNode(RuleName tr)

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Customizing Keyword Format



- It is possible to **customize the font style** for keywords (and only keywords)
- You can **customize** bold, italic, underline, strikethrough as well as foreground and background colors
- **Override the *fontstyle()* extension** for a specific meta class:



Also override the *delayContextSensitiveStyling()* extension and return the number of milliseconds of delay before the styling is done. Performance! A good number is 500.

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Integrating Non-Xtext Models (I)

- Often you want to **integrate** an Xtext textual model with other, (possibly non-Xtext) models
  - For example, you want to **"annotate" an existing UML or GMF model**
  - Or you want to define a **number of Xtext languages** separately and integrate the models via referenceces
- To do this, you first have to make the **referencing Xtext grammar** aware of the referenced meta model:

```
importMetamodel "http://www.kolbware.de/oaw/xtext/importExample" as import;
```

At the moment the metamodel has to be deployed into the "surrounding" Eclipse to make the editor work correctly, but we are working on a solution to that.

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Integrating Non-Xtext Models (II)

- You can then reference "external" meta classes **using the reference syntax** introduced before:

```
importMetamodel "http://www.kolbware.de/oaw/xtext/importExample" as import;

MyReference:
        "ref" eSuperTypes=[import::TheClass]
;
```

- To import the actual data, use a **Resource Reference** (the URI grammar keyword).

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Customizing the Linker

- The Linker is used to **wire different models**
- In most cases the **automatically created** grammar-derived Linker is sufficient
- However it is possible to **customize the behavior** further:
  Override *link(emf::EObject this)*

```
extension dataimport::dsl::Extensions;

link(emf::EObject this) :
    let all = allLocalElements() :
        all.typeSelect(imp::Jump).collect(e|e.link_nextRecord()) ->
        all.typeSelect(imp::GraphBuild).collect(e|e.link_ownerInstance()) ->
        all.typeSelect(imp::GraphBuild).collect(e|e.link_ownerRef()) ->
        all.typeSelect(imp::GraphBuild).collect(e|e.link_targetInstance()) ->
        all.typeSelect(imp::FieldMapping).collect(e|e.link_instance()) ->
        all.typeSelect(imp::FieldMapping).collect(e|e.link_field()) ->
        null;
```

```
Void link_nextRecord(imp::Jump this) :
    (let ents = this.allElements().typeSelect(imp::RecordHandler) :

        this.setNextRecord(ents.select(e|e.id() ==
            this.parsed_nextRecord).first())
);
```

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Identifying Model Elements

- It is necessary in several cases to uniquely **identify model elements** by one of their properties. Used by
  - Linker
  - findReference/ findDeclaration
  - …
- Default property is *name*
- To **customize** this, you can override the *id()* extension in the *Extensions.ext* file.

GenExtensions.ext

```
String id(emf::EObject x) :
    let p = x.metaType.getProperty("name") :
        p==null?
            null :
            p.get(x);
```

© Bernd Kolb, Markus Voelter, Peter Friese 2008

CONTENTS

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Processing Xtext Files I: Overview

- Since Xtext instantiates Ecore metamodels, the models can **be processed with any EMF tool** – e.g. oAW.
  - We will not explain those here.
- You can process them with **oAW Workflows,** use them like any other **EMF resource** and parse them into **your own Java programs**.
- You can check the *Create Generator Project* checkbox in the projection creation wizard to get a sample generator as a starting point.

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Processing Xtext Files II: From within an oAW Workflow

- The only Xtext-specific aspect is **using the generated parser.**

*generator.oaw

```
1  <workflow>
2
3     <property name='modelFile' />
4
5     <property name='targetDir' value='src-gen/'/>
6
7     <component file='org/example/dsl/parser/Parser.oaw'>
8        <modelFile value='${modelFile}'/>
9        <outputSlot value='theModel'/>
10    </component>
11
12    <component class='oaw.workflow.common.DirectoryCleaner' directories='${targetDir}'/>
13
14    <component class='oaw.xpand2.Generator'>
15       <metaModel id='mm' class='org.eclipse.m2t.type.emf.EmfRegistryMetaModel'/>
16       <expand value='org::example::dsl::Main::main FOR theModel'/>
17       <genPath value='${targetDir}'/>
18    </component>
19
20 </workflow>
```

This workflow cartridge is **generated automatically** by Xtext

Specify the **model file** that should be processed

Specify the **workflow slot** in which the created model should be stored

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Processing Xtext Files II: Code Generation Templates

- oAW's **Xpand** template language is a powerful and well established code generation facility with nice tooling.
- You can easily **traverse the model**/meta model using the Xtend language (à la OCL)

data.xpt

```
1 «IMPORT imp»
2
3 «EXTENSION dataimport::dsl::impUtil»
4
5 «DEFINE root FOR DataStructure»
6 «FILE filename()»
7
8 import java.util.List;
9 import java.util.ArrayList;
10 import dataimport.platform.DataBase;
11
12
13 public class «classname()» extends dataimport.platform.DataBase
14
15     «FOREACH attributes AS a»
16         private «a.type» «a.name»;
17     «ENDFOREACH»
18
19     «FOREACH references AS r»
20         «IF r.ismulti»
21             private List<«r.type.fqClassname()»> «r.name»List =
22         «ELSE»
23             private «r.type.fqClassname()» «r.name»;
24         «ENDIF»
25     «ENDFOREACH»
26
27     «FOREACH attributes AS a»
28         public void set«a.name.toFirstUpper()»( «a.type» value )
```

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Processing Xtext Files III: The EMF way

- Use EMF's **native resource mechanism**
  - Xtext **registers a *ResourceFactory*** for the DSL's file extension
  - Simply **pass the uri** of the file you want to read to *getResource()*
  - Note that you **cannot write** the model back!

Test.java

```
22     private void load( String fileURI ) throws IOException {
23         new StandaloneSetup().setPlatformUri("..");
24         impResourceFactory.register();
25         ResourceSet resourceSet = new ResourceSetImpl();
26         Resource res = resourceSet
27             .getResource( URI.createURI( fileURI ), true );
28         if ( !res.isLoaded() ) {
29             res.load( Collections.EMPTY_MAP );
30         }
31     }
```

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Processing Xtext Files III: From you own custom code

- Use the **generated parser**
  - This is what the *ResourceFactory* does internally with some default behavior for error handling
  - Note **how parse errors** and **constraint violations** are handled.

```
Reader2.java
1  package dataimport.dsl.parser;
2
3  import java.io.FileInputStream;
14
15 public class Reader2 {
16
17     protected ResourceSet resourceSet = new ResourceSetImpl();
18
19     public EObject read(String fileURI) throws IOException {
20         XtextParser p = new XtextParser(new FileInputStream(fileURI));
21         // The parseTree Root Node
22         Node rootNode = p.getRootNode();
23         p.preLinking();
24         p.doLinking();
25         p.postLinking();
26         List<ErrorMsg> parseErrors = p.getParseErrors();
27         if (p.getParseErrors().size()== 0) {
28             Issues listOfViolatedConstraints = p.doCheck();
29         }
30         return NodeUtil.getModelElement(rootNode);
31     }
32
33 }
```

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Details about nodes: NodeUtil

- Typically you you are **only working on the AST** (ecore file)
- However, sometimes you need data from the **parse tree**
  - **Location** of an element in the source file
  - The **text** for a certain element
  - The parse tree **node** at a certain **offset**

```
NodeUtil 1.7.2.4 (ASCII -kkv)
  getContainingAssignment(Element)
  getContainingElement(Element)
  getContainingRule(Element)
  getModelElement(Node)
  getNode(EObject)
  getNode(Node, EObject)
  getNodeBeforeOffset(Node, int)
  getNodeUnderOffset(Node, int)
  getPossibleFollows(XtextFile, Node)
  getRoot(Node)
  getText(String, Node)
  isIdentifier(Token)
```

- Here is an example of how to get the **location of an element** in the source file:

```
NodeUtilSample.java
8
9   public String getLocInfo(EObject modelElement) {
10      Node node = NodeUtil.getNode(modelElement);
11      return "Line " + node.getLine() +
12          " [" + node.getStart() +":" +node.getEnd()+"]" ;
13  }
```

© Bernd Kolb, Markus Voelter, Peter Friese 2008

CONTENTS

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Language Tooling Made Easy!

- **Designing good languages** is not easy – and Xtext cannot simplify that process.
- However, **building good tools for good languages** is also not easy, and often a reason for not trying to build a language.
- Xtext makes that second challenge – the tooling – much much easier, thereby **making language development a real practical option.**
- **Go ahead, give it a try!**

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Eclipse TMF

- oAW Xtext is now part of the **Eclipse Textual Modeling Framework** project.
- Together with INRIA's TCS, it **serves as the basis** for TMF.
- See also
  - http://www.eclipse.org/proposals/tmf/
  - http://wiki.eclipse.org/TMF
  - http://wiki.eclipse.org/Xtext

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Limitations

- It is not yet possible to specify **custom action code in the parser**
  - This sometimes results in "ugly" meta models
  - Becomes apparent when building expression languages
  - [This is a plan item for TMF]

© Bernd Kolb, Markus Voelter, Peter Friese 2008

More...

- www.openarchitectureware.org
  - Specifically the reference docs contain more info about Xtext
- www.eclipse.org/modeling/
  - EMF and all the other related tooling
- www.eclipse.org/modeling/tmf
  - Home of TMF / Xtext
- Podcast: Software Engineering Radio
  - www.se-radio.net
  - Several Episodes on MDSD

© Bernd Kolb, Markus Voelter, Peter Friese 2008

Thank you

Questions?

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

völter
ingenieurbüro für softwaretechnologie
Markus Voelter
voelter@acm.org
http://www.voelter.de

itemis
Software | Consulting | Coaching
Peter Friese
peter.friese@itemis.de
http://oaw.itemis.eu

© Bernd Kolb, Markus Voelter, Peter Friese 2008