Open Source Modeling

Textual DSLs with oAW Xtext

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

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

CONTENTS

- Intro to Textual DSLs
- Getting Started with Xtext
- Intro to Example
- Defining the Grammar
- Simple Editor Customization
- More Advanced Customization
- Processing the Models
- Summary

CONTENTS

- **Intro to Textual DSLs**
- Getting Started with Xtext
- Intro to Example
- Defining the Grammar
- Simple Editor Customization
- More Advanced Customization
- Processing the Models
- Summary

Domain Specific Languages

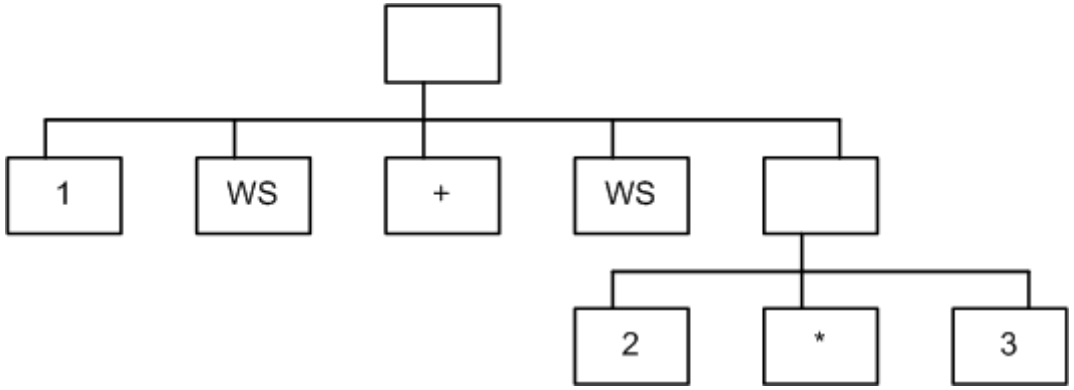
- A Domain Specific Language (DSL) is a **focused, processable language** for describing a specific **concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.

- DSLs can be **classified** in many ways:
  - Configuration vs. Customization
  - Internal vs. External
  - Graphical vs.textual

- We concentrate on **external textual customization** DSLs

How it works in general (I)

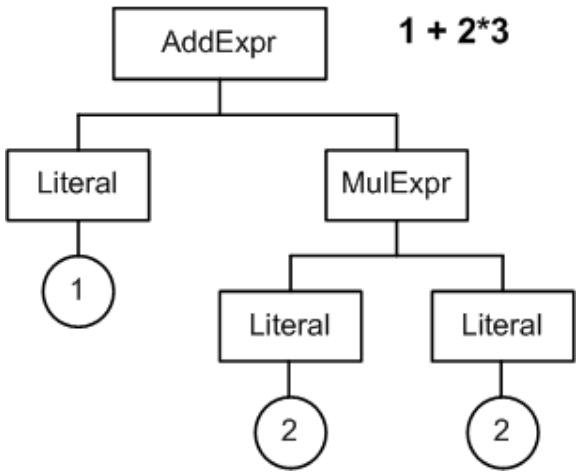
- Typically textual DSLs are build using one of the many **parser generators (**ANTLR, JavaCC, Lex/yacc), or even by a hand-writing a custom parser
- Parser: **match text** and try to create a **parse tree**



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

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

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 and code completion

→ A lot of work! Only few people are willing to do that

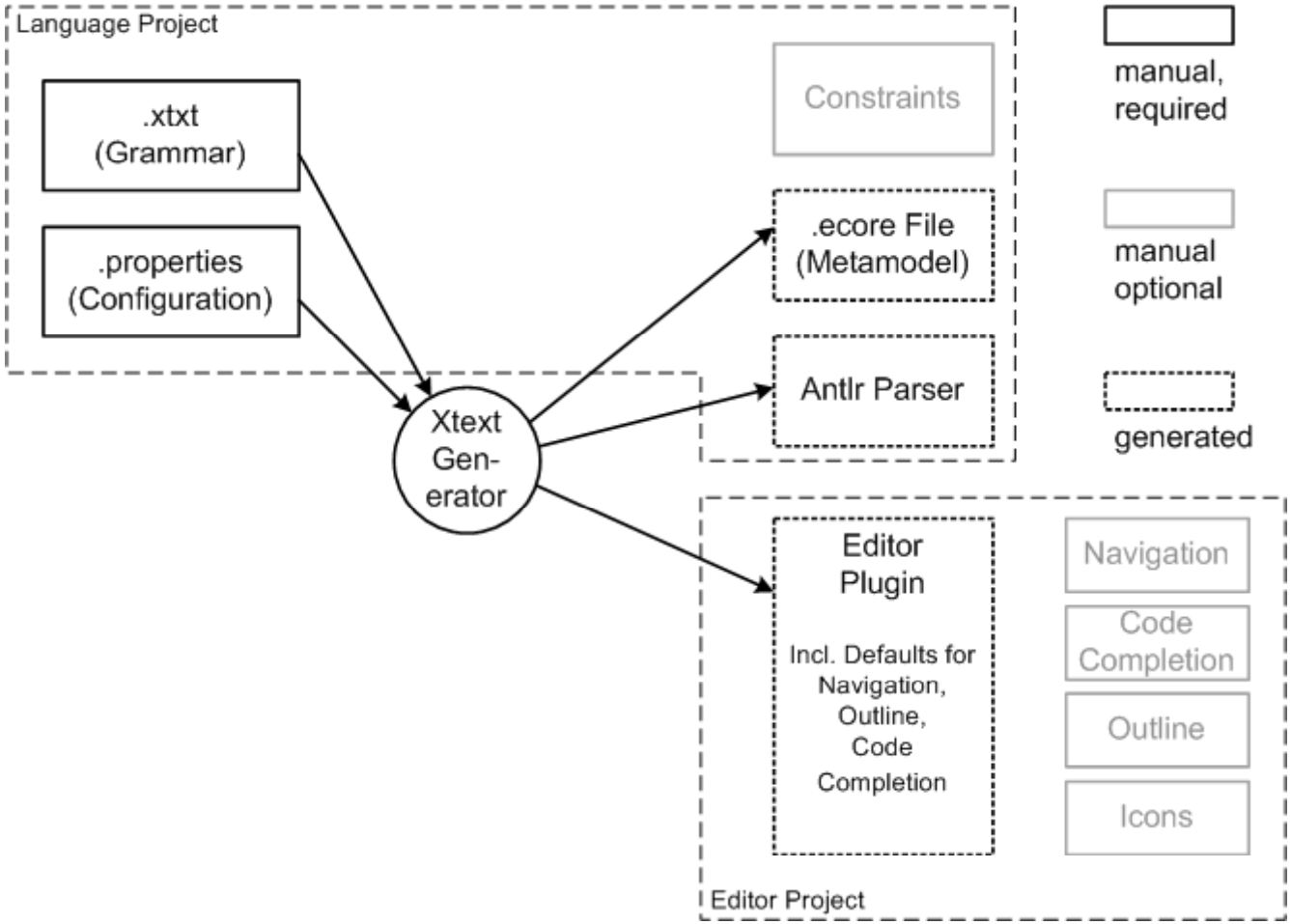
CONTENTS

- Intro to Textual DSLs
- **Getting Started with Xtext**
- Intro to Example
- Defining the Grammar
- Simple Editor Customization
- More Advanced Customization
- Processing the Models
- Summary

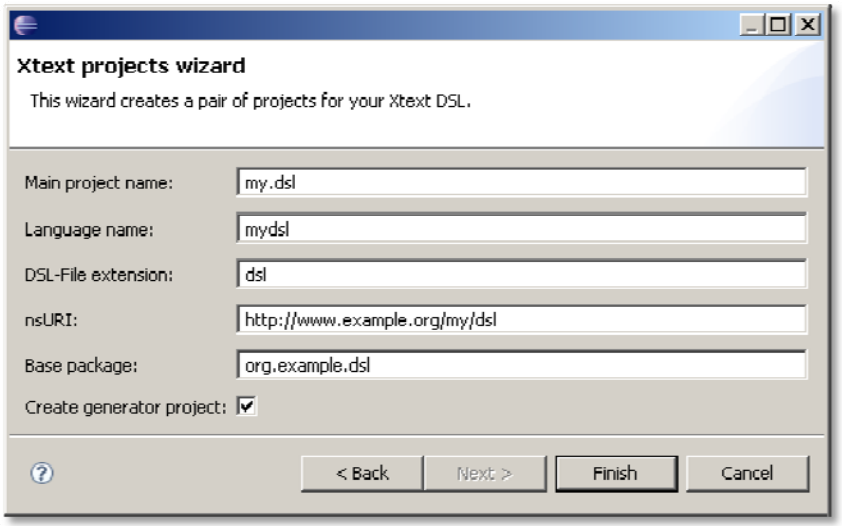
Xtext

- 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**

Xtext Overall Process

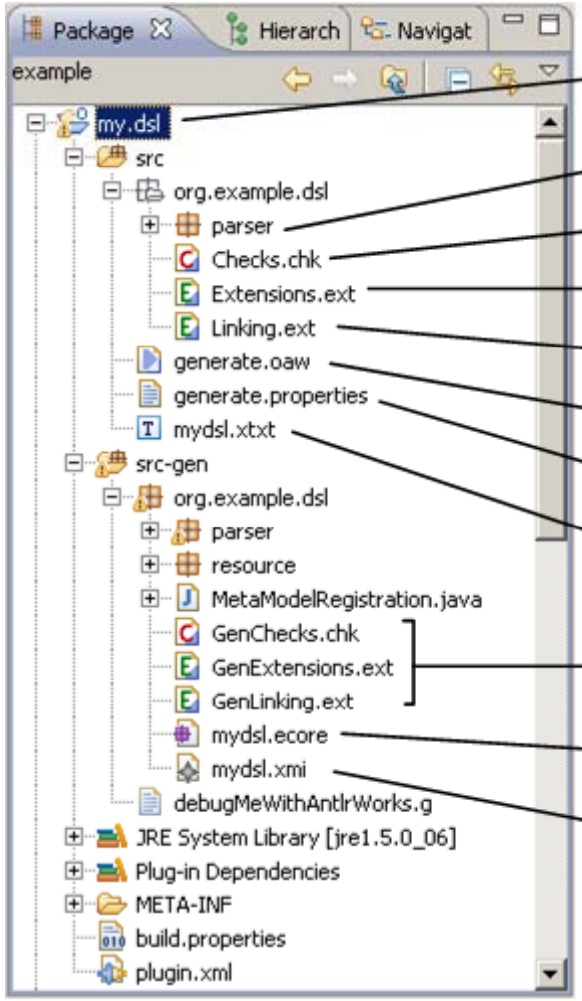


Creating a new Xtext Project I: Xtext Project Wizard



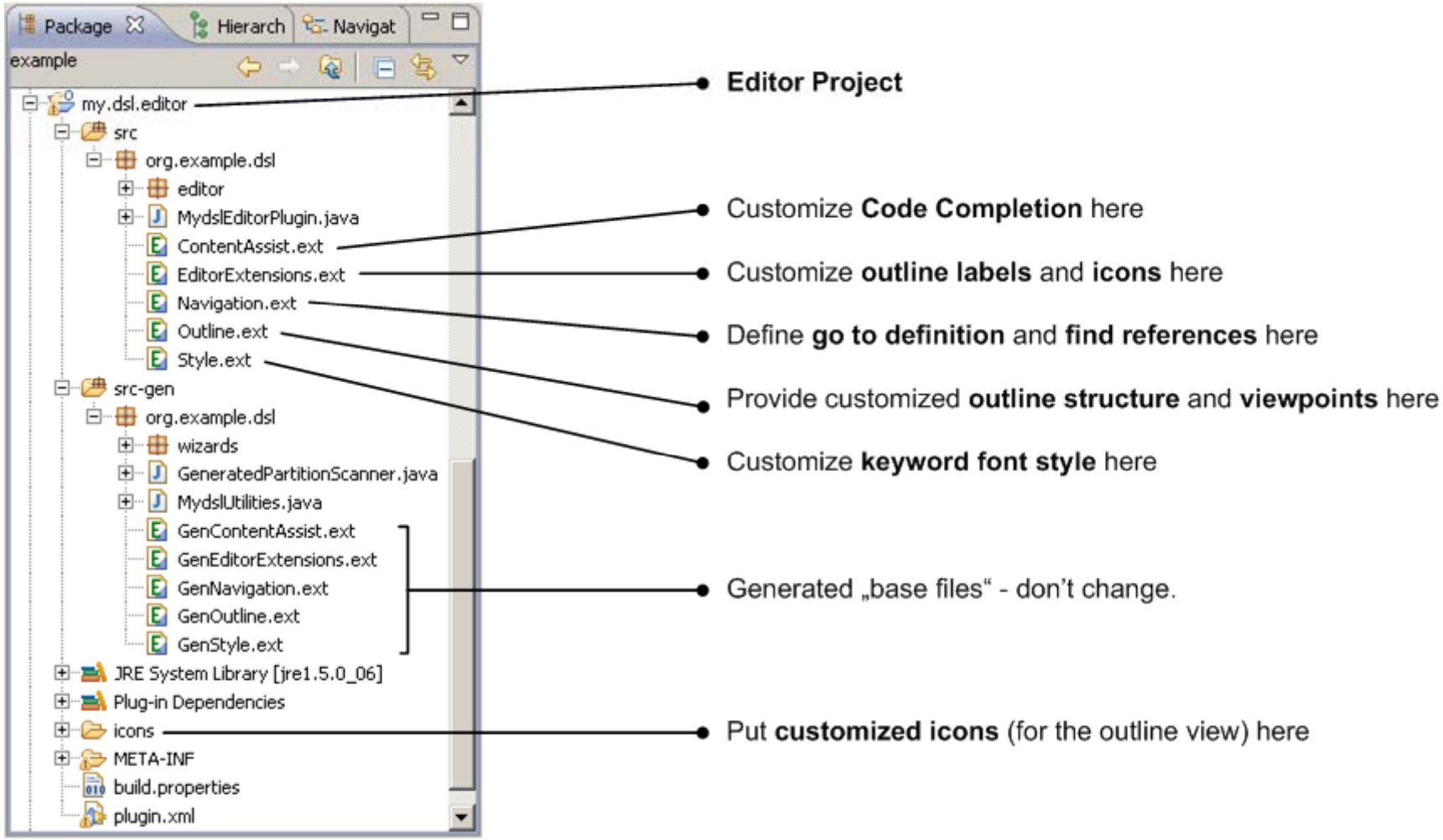
- **Main Project Name:**
  - name of language project
  - *.editor* project contains editor
- **Language Name:** package name of generated meta model
- **DSL File Extension:** file extension used to associate editor
- **nsURI:** full namespace URI of the generated meta model
- **Base Package:** all code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

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

Creating a new Xtext Project III: Created Editor Project



CONTENTS

- Intro to Textual DSLs
- Getting Started with Xtext
- **Intro to Example**
- Defining the Grammar
- Simple Editor Customization
- More Advanced Customization
- Processing the Models
- Summary

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
```

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**

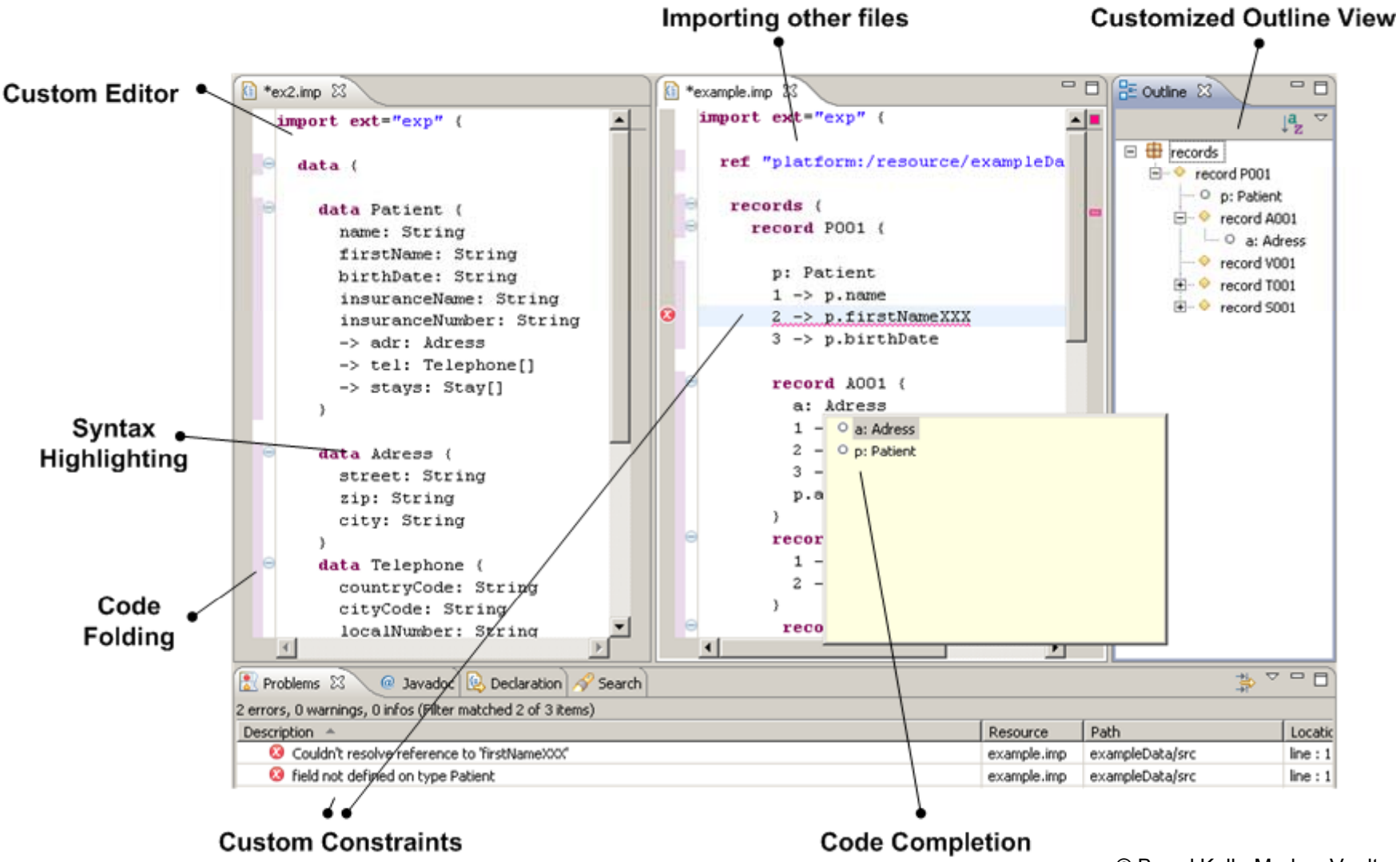
ex2.imp

```
import ext="exp" {
    data {
        data Patient {
            name: String
            firstName: String
            birthDate: String
            insuranceName: String
            insuranceNumber: String
            -> adr: Adress
            -> tel: Telephone[]
            -> stays: Stay[]
        }
        data Adress {
        data Telephone {
        data Stay {
        data Station {
    }
}
```

*example.imp

```
import ext="exp" {
    ref "platform:/resource/exampleData/src/ex2.
    records {
        record P001 {
            p: Patient
            1 -> p.name
            2 -> p.firstName
            3 -> p.birthDate

            record A001 {
            record V001 {
            record T001 {
            record S001 {
        }
    }
}
```

Intro to Example III: Solution Tooling



CONTENTS

- Intro to Textual DSLs
- Getting Started with Xtext
- Intro to Example
- **Defining the Grammar**
- Simple Editor Customization
- More Advanced Customization
- Processing the Models
- Summary

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 **;**

imp.xtxt

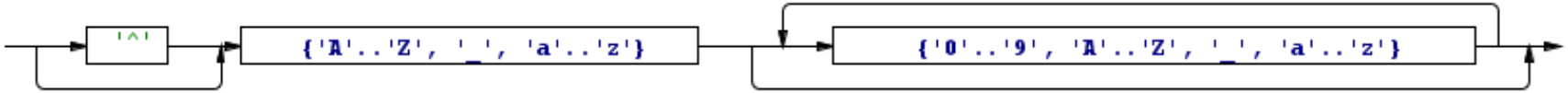
```
ImportSpec:
    "import" "ext" "=" fileExt=STRING "{"
        dataSection=DataSection
        recordSection=RecordSection
    "}";

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

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

Attribute:...

DataSection:...

RecordSection:...

RecordHandler:...

Jump:...

GraphBuild:...

FieldMapping:...

Instance:...
```

Different Kinds of Rules

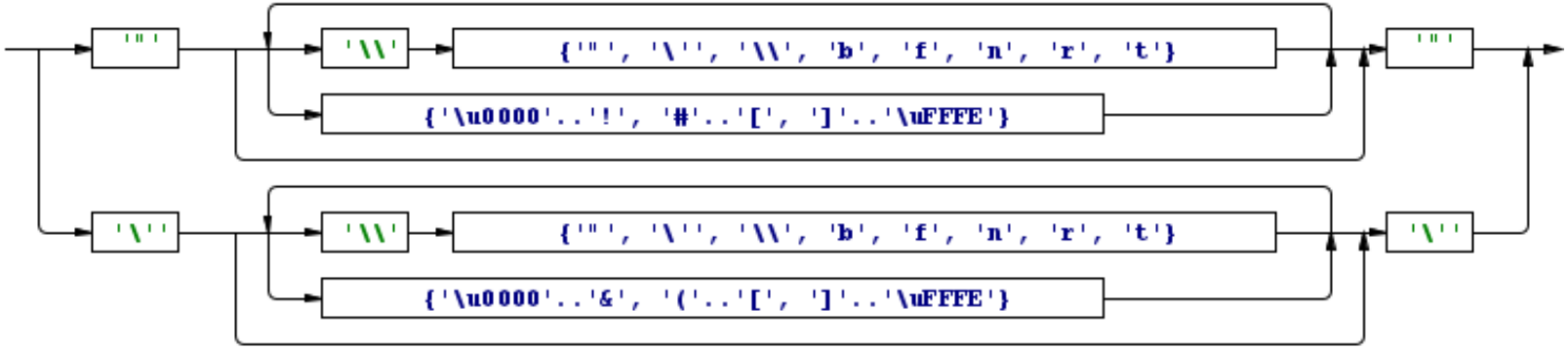
- **Type Rule**
  - For each **rule** Xtext creates a **class in the metamodel**
  - Each rule **property** results in a **property of the metaclass**
- **String Rule**
  - String rules are parsed to a String.
  - They are in effect custom lexer rules (recognizing string patterns)
- **Enum Rule**
  -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

Built-in Lexer Types (I)

- ID



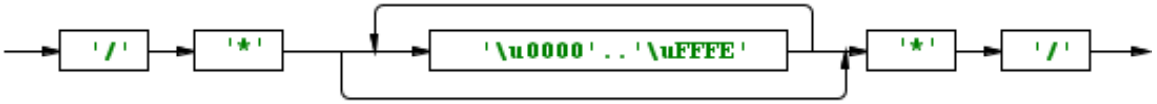
- STRING



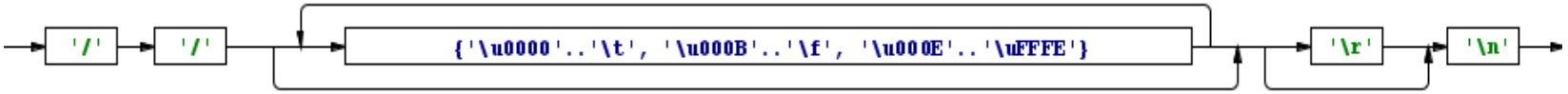
- INT

Built-in Lexer Types (II)

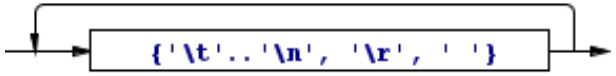
- **Multiline comment**



- **Single line comment**



- **Whitespace**



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

Built-in Reference Types

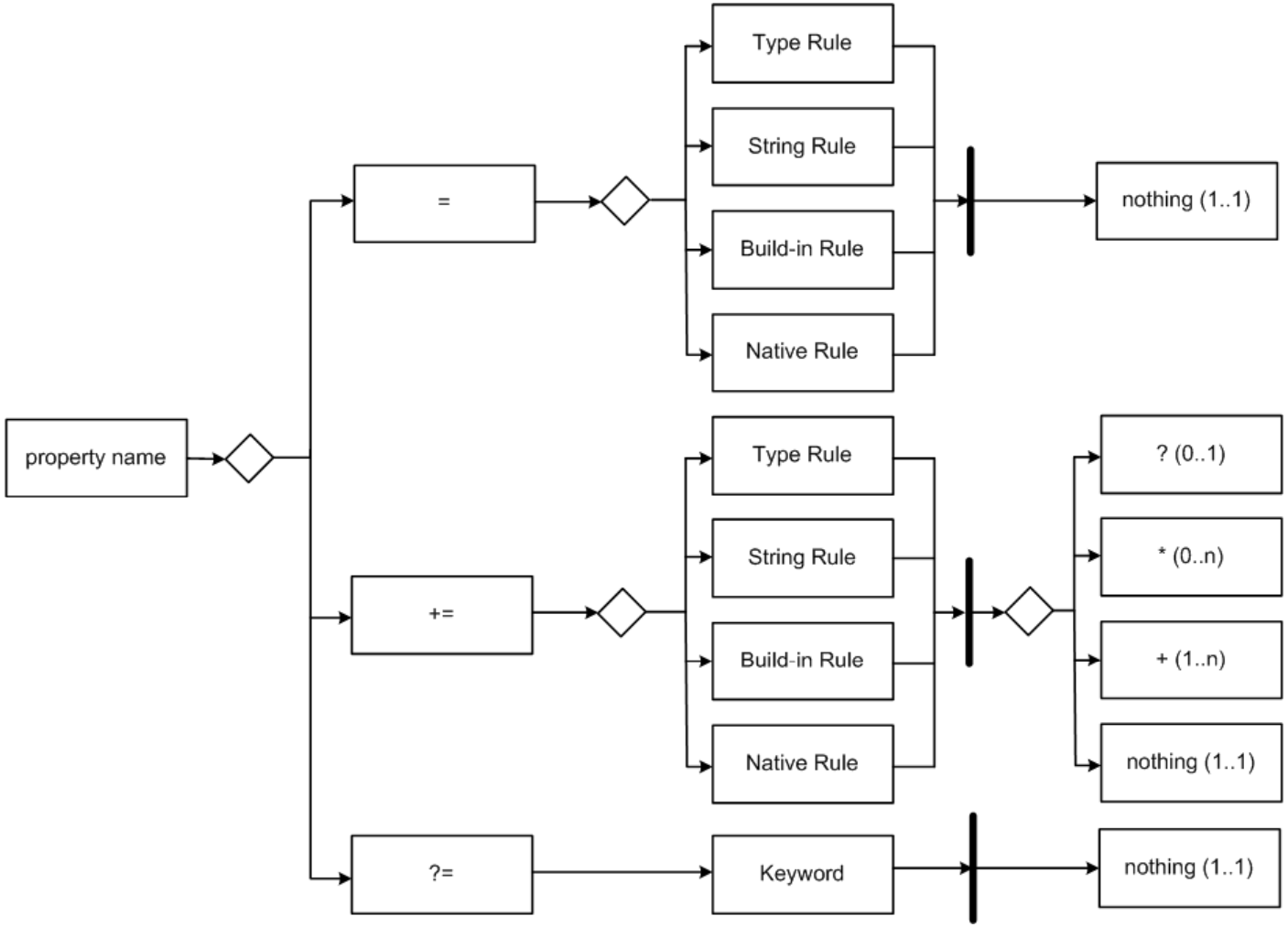
- **Reference**

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

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)

Type Rules – Combinations



Type Rules – Examples

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

imp.xtxt

```
1  ImportSpec:
2      "import" "ext" "=" fileExt=STRING "{"
3          dataSection=DataSection
4          recordSection=RecordSection
5      "}";
6
7  DataStructure:
8      "data" name=ID "{"
9          (attributes+=Attribute | references+=Reference)*
10     "}";
11
12 Reference:
13     "->" name=ID ":" type=[DataStructure] (ismulti?="[]")?;
```

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

Reference Rule

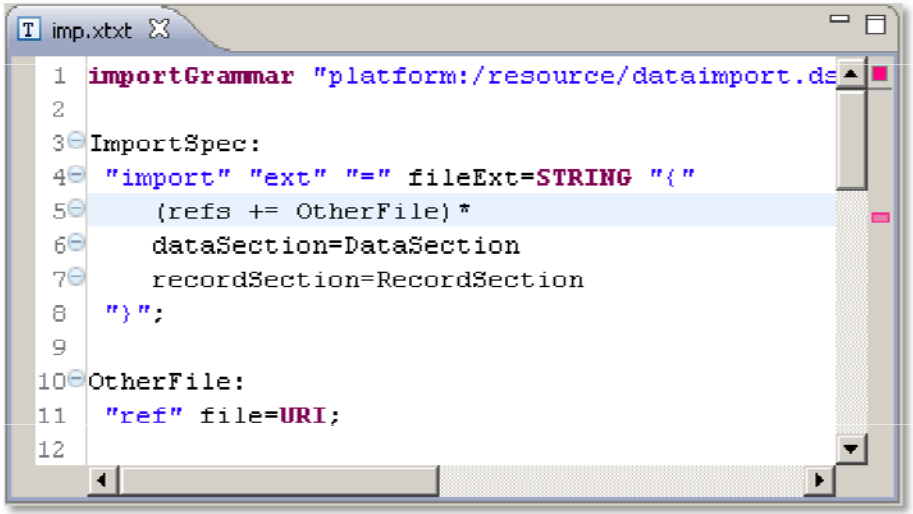
- By default**, execution of rules results in a tree** of data
- You can **reference** other elements via a **reference rule**
  - In textual languages a **reference** has to be **by name**
  - During **linking**, Xtext "dereferences" these by name-references
  - Target's *name* property is used here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

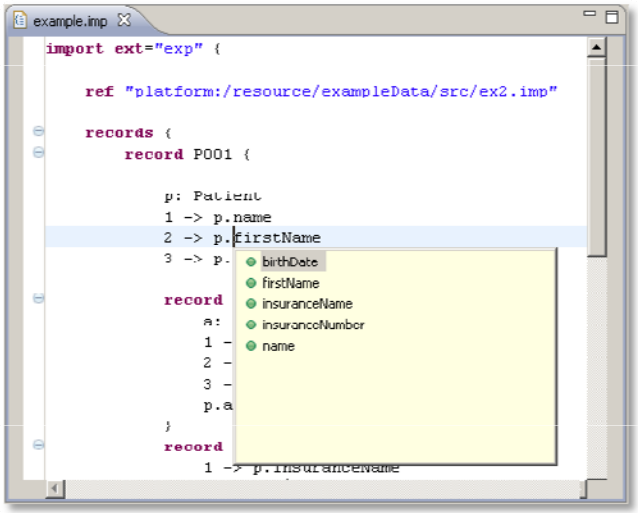
data.xtxt

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

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

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

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





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**.

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

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)

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.

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'";
```

CONTENTS

- Intro to Textual DSLs
- Getting Started with Xtext
- Intro to Example
- Defining the Grammar
- **Simple Editor Customization**
- More Advanced Customization
- Processing the Models
- Summary

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
```

Constraint Checks

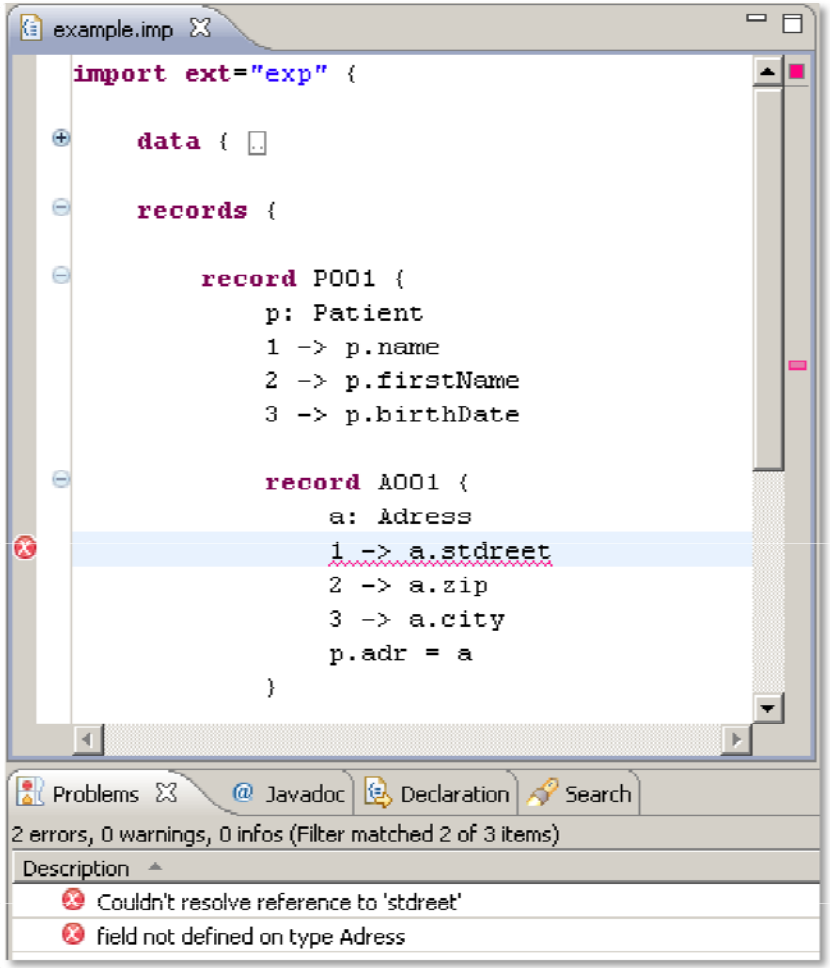
- Uses the **oAW Check language**, based on Xtend
  - Used to validate the static semantics of the models
- Report Warnings and Errors
- Based completely on **the generated meta model**, no concrete syntax hassles
- Tool support (syntax highlighting, code completion, …)

Extensions.ext | *Checks.chk

```
import imp;

extension dataimport::dsl::Extensions;

context Instance ERROR "type not defined":
    allElements().typeSelect(DataStructure).contains(type);

context FieldMapping ERROR "instance not defined or not in scope":
    ((RecordHandler)eContainer).instancesInScope().contains(instance);

context FieldMapping ERROR "field not defined on type "+instance.type.name:
    instance.type.attributes.contains( field );

context Instance ERROR "instanc names must begin with a lowecase letter":
    name.toFirstLower() == name;
```

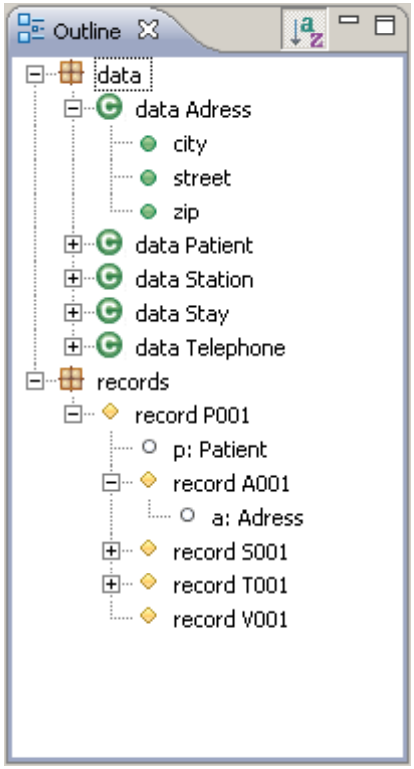
Real Time Constraint Validation

- 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.



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.



EditorExtensions.ext

```
import imp;

extension dataimport::dsl::Extensions reexport;
extension dataimport::dsl::GenEditorExtensions reexport;

label( Instance this ): name+": "+type.name;

label( RecordHandler this ): "record "+name;
image( RecordHandler this ): "recordHandler.gif";

label( DataStructure this ): "data "+name;
image( DataStructure this ): "data.gif";

label( Reference this ): "-> "+name+": "+type.name;

label( Instance this ): name+": "+type.name;
image( Instance this ): "instance.gif";
```

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

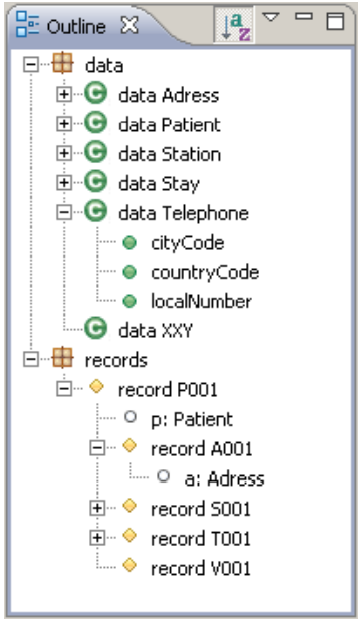
CONTENTS

- Intro to Textual DSLs
- Getting Started with Xtext
- Intro to Example
- Defining the Grammar
- Simple Editor Customization
- **More Advanced Customization**
- Processing the Models
- Summary

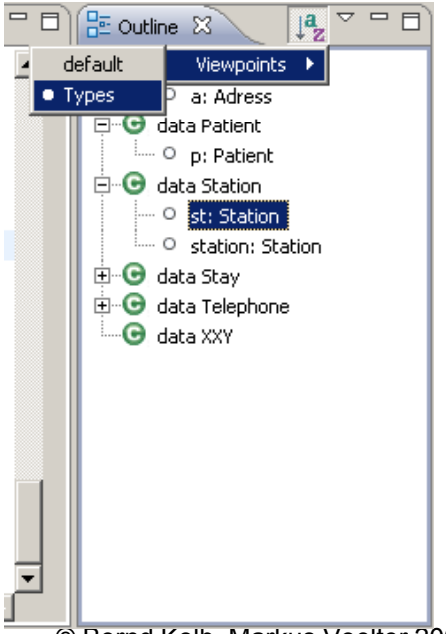
Customizing the Outline II: Structural Customization

- 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)



Customizing the Outline II: Structural Customization II

Outline.ext | EditorExtensions.ext | mydsl.xtxt | imp.ecore

```
1 import imp;
2 import tree;
3
4 extension dataimport::dsl::EditorExtensions;
5 extension dataimport::dsl::GenOutline reexport;
6
7 create UIContentNode outlineTree_Types(emf::EObject model) :
8     setLabel(model.label()) ->
9     setImage(model.image()) ->
10    setContext(model)->
11    children.addAll( model.eAllContents.typeSelect(DataStructure).create
12
13 create UIContentNode createDSContainer( DataStructure ds, emf::EObject m
14    setLabel( ds.label() ) ->
15    setImage( ds.image() ) ->
16    setContext(ds) ->
17    children.addAll( model. eAllContents.typeSelect(Instance).select(i|i
18
19 create UIContentNode createINode( Instance i):
20    setLabel(i.label()) ->
21    setImage(i.image()) ->
22    setContext(i);
23
24 List[String] viewpoints() :
25    {"Types"};
```

- 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**

Customizing Content Assist

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

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

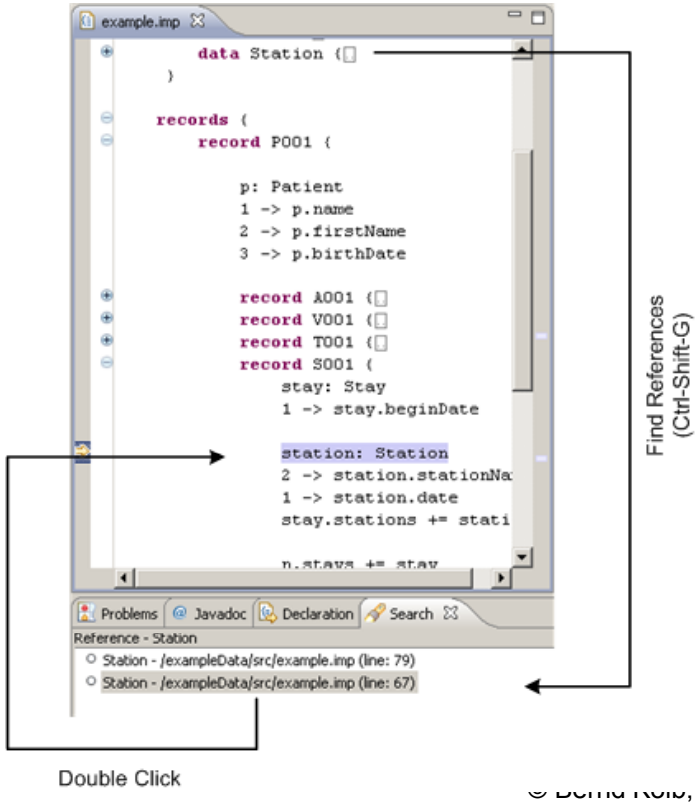
- A Proposal consists of a **label**, an **icon** and the **text to insert**

*ContentAssist.ext

```
1 import imp;
2 import codeassist;
3
4 extension dataimport::dsl::Extensions;
5 extension dataimport::dsl::GenContentAssist reexport;
6 extension dataimport::dsl::EditorExtensions;
7 extension org::openarchitectureware::util::stdlib::io;
8
9 List[Proposal] completeRecordHandler(emf::EObject grammarElement, emf::EObject ct
10     { newProposal( "new record...", "record <unnamed> {  }", null ) };
11
12 List[Proposal] completeFieldMapping_field(emf::EObject ctx, String prefix) :
13     ((FieldMapping)ctx).instance.type.attributes.
14         collect(x|newProposal(x.label(),x.id(),x.image()));
15
16 List[Proposal] completeFieldMapping_instance(emf::EObject ctx, String prefix) :
17     ((RecordHandler)ctx.eContainer).instancesInScope().
18         collect(x|newProposal(x.label(),x.id(),x.image()));
19
```

Navigation

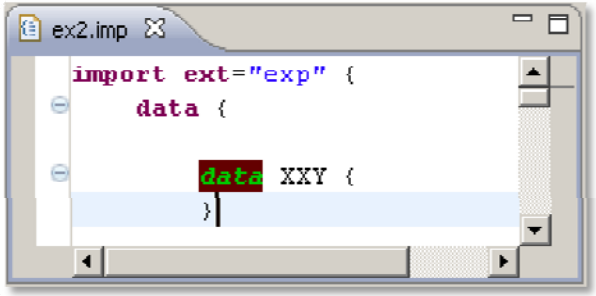
- By default, if you use references in your DSL, you can use ***Go To Definition*** and ***Find References*** out of the box (it works cross-resource, see next section)



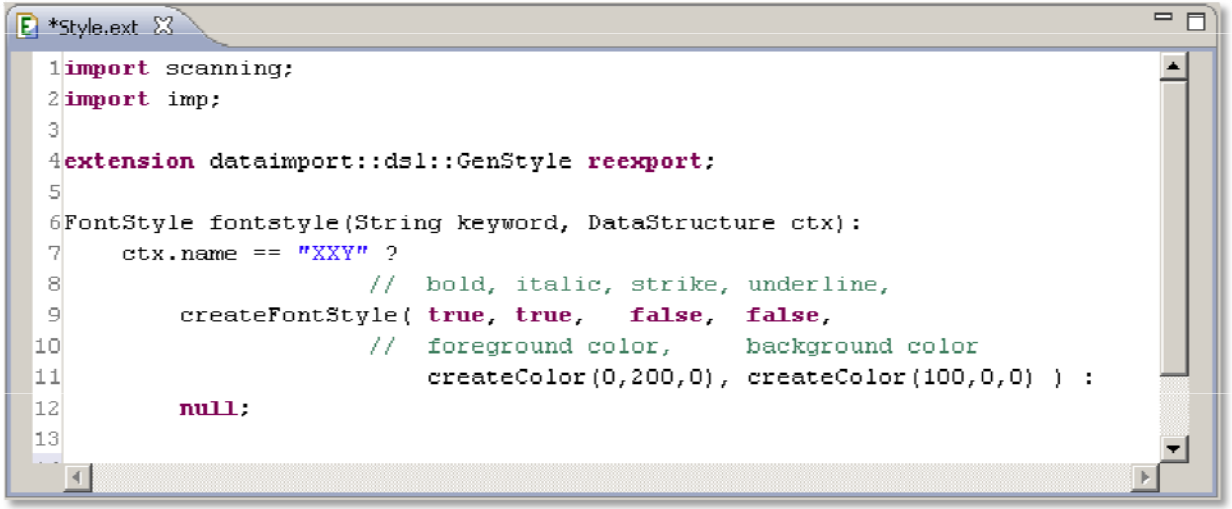
Navigation II: Customization

- 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)

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:



There is some **performance impact** when using this feature.

Integrating other EMF Files

- Tbd.
- *Explain how to code-complete and constraint check into other EMF files (non-Xtext Stuff)*

Todo.

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)*

GenLinking.ext

```
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()) ->
        all.typeSelect(imp::Instance).collect(e|e.link_type()) ->
        all.type
        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())
);
```

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.

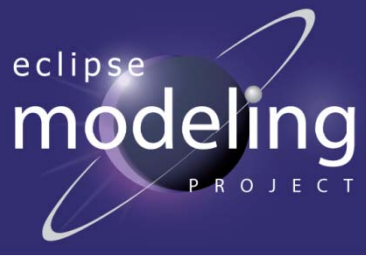
GenExtensions.ext

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

CONTENTS

- Intro to Textual DSLs
- Getting Started with Xtext
- Intro to Example
- Defining the Grammar
- Simple Editor Customization
- More Advanced Customization
- **Processing the Models**
- Summary

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.

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

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!

*Reader.java

```
1  package dataimport.dsl.parser;
2
3  import java.io.IOException;
10
11 public class Reader {
12
13     protected ResourceSet resourceSet = new ResourceSetImpl();
14
15     public Resource read(String fileURI) throws IOException {
16         Resource res = resourceSet.getResource(URI.createURI(fileURI), true);
17         if (!res.isLoaded()) {
18             res.load(Collections.EMPTY_MAP);
19         }
20         return res;
21     }
22 }
```

Processing Xtext Files III: From you own custom code

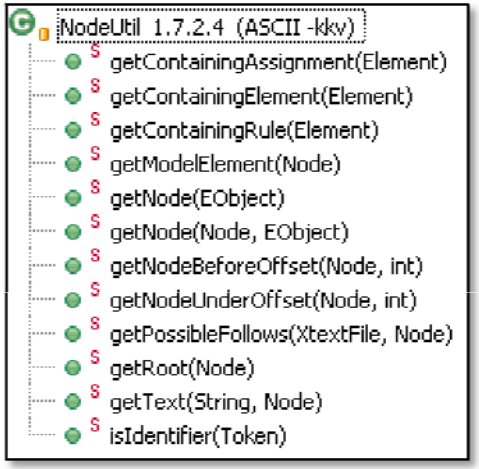
- Use the **generated parser**
  - This is what the *ResourceFactory* does internally with some default behavior for error handling
  - Note **how parse errors** and **constraint violations** are handled.

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 }
```

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**



- 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  }
```

CONTENTS

- Intro to Textual DSLs
- Getting Started with Xtext
- Intro to Example
- Defining the Grammar
- Simple Editor Customization
- More Advanced Customization
- Processing the Models
- **Summary**

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!**

Eclipse TMF

- oAW Xtext is currently in the process of transitioning into the (relatively) new **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

Limitations

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

- It is not yet possible to **start from a metamodel** and annotate the notation
  - The current approach is suitable if you only use a textual syntax
  - If you want to mix syntaxes or want to define a textual syntax for an existing metamodel, the current approach is not useful
  - [This is a plan item for TMF]

More...

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

Thank you

- Thanks for your attention!

Questions?

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

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