

Xtext Reference Documentation

Sven Efftinge, sven@efftinge.de, www.efftinge.de

PRIMARY SPONSORS



SECONDARY SPONSOR



Table of Contents

INTRODUCTION.....	3
INSTALLATION.....	3
GETTING STARTED.....	3
THE GRAMMAR LANGUAGE.....	4
EXAMPLE.....	4
RULES.....	5
<i>Simple Rule</i>	5
<i>Abstract Rule</i>	5
<i>Enum Rule</i>	6
<i>String rule</i>	6
TOKENS.....	7
<i>Keyword Token</i>	7
<i>Identifier Token</i>	8
<i>String Token</i>	8
Specifying the string literal terminals.....	8
<i>Single Line Token</i>	9
COMMENTS.....	9
THE GENERATOR.....	9
GENERATED AND MANUAL CODE.....	10
THE DIFFERENT PROJECTS.....	10
<i>The main project</i>	10
<i>The editor project</i>	10
<i>The generator project</i>	11
FUTURE ENHANCEMENTS.....	11

Introduction

Does this sound familiar to you?

Oh, I need to rename this misspelled property within our domainmodel. Ok, so let's startup this big UML monster... and by the way let's get a new cup of coffee. Cool, it has been started up already. Grabbing the mouse, clicking through the different diagrams and graph visualizations... Ahhh, there is the name of the property right down there in the properties view. Let's change it, export it to XMI (...drinking coffee), starting the oAW generator (in a jiffy ;-)). Oh, it's not allowed for the property to be named that way, a constraint says that properties names should start with a lower case letter. Ok, let's change that and again reexport...

Some moments later, everything seems to work (tests are green). Ok let's check it in!

Oh someone else has also modified the model! Aaarrgggh....

Think of this:

Want to change a properties name? Ok, open the respective text file, rename the properties name and save it. The editor complains about a violated constraint. Ok fix the issue, save again and generate. Check the changes into SVN (CVS). Oh there is a conflict, ok, let's simply merge it using Diff.

And now? Let's have a cup of coffee :-)

Xtext is a textual DSL development framework. Providing the ability to describe your DSL using a simple EBNF notation. Xtext will create a parser, a metamodel and a specific Eclipse texteditor for you!

Installation

Xtext is shipped as a set of Eclipse Plugins. Just use the Eclipse update manager and get it from the updatesite :

<http://www.openarchitectureware.org/updatesite/milestone/site.xml>

Xtext is based on oAW 4.1 and EMF 2.2, therefore those features are also required

Getting started

You should have a look at the Xtext tutorial screencast located at <http://www.openarchitectureware.org/>. This will give you a good overview of how Xtext basically works. Come back to this documentation to find out about additional details.

The Grammar Language

At the heart of Xtext lays its grammar language. It's a lot like an extended Backus-Naur-Form but it not only describes the concrete syntax, but also the abstract syntax (metamodel). The core abstraction of the grammar language is called a *Rule*.

Example

This is an example for a *Rule* describing something called an entity:

```
Entity :
    "entity" name=ID "{"
        (features+=Feature)+
    "}"
```

`Entity` is both the name of the rule and the name of the metatype corresponding to this rule. After the colon the description of the rule is following. A description is made up of *tokens*. The first token is a *KeywordToken* which says that a description of an entity starts with the keyword `entity`. A so called *IdentifierToken* follows (`name=ID`) where the value of the identifier is assigned to the Entity's property called `name`. Then (enclosed in curly brackets ("`{`" and "`}`")) one or more features should be declared (`(features+=Feature)+`).

This time the token points to another rule (called `Feature`) and each feature is added (note `+=` operator) to the Entity's reference called `features`.

The `Feature` rule itself could be described like this:

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

so that the following description of an entity would be valid according to the grammar:

```
entity Customer {
    String name;
    String street;
    Integer age;
    Boolean isPremiumCustomer;
}
```

Note, that the types (`String`, `Integer`, `Boolean`) used in this description of a customer, are simple identifiers, they don't have been mapped to e.g. Java types or something else. So according to the grammar this would also be valid, so far:

```
entity X {
    X X;
    X X;
    X X;
    cjbdlfjerifuerfijerf dkjdhferifheirhf;
}
```

Rules

There are different types of rules.

Simple Rule

We've already seen how the Simple Rule works. The rule's name is the name of a concrete meta type (generated by Xtext). Therein tokens are specified and the values are assigned to features of the actual meta type.

Abstract Rule

Sometimes you need abstract rules, in order to let a feature contain elements of different types. We have seen the Feature rule in the example. If you would like to have two different kinds of Feature (e.g. Attribute and Reference) you could create an abstract rule and two simple rules like this:

```
Abstract Feature :  
    Attribute | Reference;  
  
Attribute :  
    type=ID name=ID ";" ;  
  
Reference :  
    "ref" (containment?"+" )? type=ID name=ID  
        ("<->" oppositeName=ID) ? ";" ;
```

With this you could describe the bi-directional relations between two entities:

```
entity Customer {  
    String name;  
    ref +Order orders <-> customer  
}  
  
entity Order {  
    String productId;  
    Amount price;  
    ref Customer customer  
}
```

Note that you have to link the references using appropriate extensions (we will see later).

The metamodel derived from the grammar described abstract syntax trees. Crossreferences between tree nodes (such as from reference to entity) have to be made in a later step. But it is fairly easy, so don't bother right now.

Enum Rule

The enum rule is used to have Enumerations inside the metamodel. For example if you would like to hardwire the possible datatypes for attributes into the language you could just write:

```
Attribute :  
    type=DataType name=ID ";;"  
  
Enum DataType :  
    String="string" | Integer="int" | Boolean="bool";
```

So that this would be valid:

```
entity Customer {  
    string name;  
    string street;  
    int age;  
    bool isPremiumCustomer;  
}
```

but this would not

```
entity Customer {  
    X name; // type X is not known  
    String street; // type String is not known (case sensitivity!)  
}
```

String rule

Xtend provides built-in Tokens (we have seen the `IdentifierToken` and the `KeywordToken` already). Sometimes this is not sufficient, so we might want to create our own Tokens. Therefore we have the so called `String Rule`.

Example:

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

The contents of the `String Rule` is simply concatenated and returned as a string. One can refer to a `String Rule` in the same manner we refer to any other rule.

So just for the case you want to declare datatypes using your DSL and therein specify how it is mapped to Java (not Platform independent, i know, but expressive and comfortable), you could do so using the following rules.

```
Attribute :  
    type=DataType name=ID ";;"  
  
DataType :  
    "datatype" name=ID "mappedto" javaType=JavaIdentifier;
```

And a respective model could look like this:

```
entity Customer {
    string name;
    string street;
    int age;
    bool isPremiumCustomer;
}

datatype string mappedto java.util.String
datatype int mappedto int
datatype bool mappedto boolean
```

You could of course point to a custom typemapping implementation, if you need to support multiple platforms (like e.g. SQL, WSDL, Java,...). Additionally you should consider to define the datatypes in a separate file, so the users of your DSL can import and use them. It's like any other language providing not only the language but also libraries.

For now we just have those four different types of rules. Let's have a look at the different Token types.

Tokens

Within rules there are different Token types which can be used. Let's start with the two simplest.

Keyword Token

All static characters or words (keywords) can be specified directly into the grammar rules using the usual string literal syntax. We never need the value of keyword because it's static therefore we know the value implicitly. But sometimes there are optional keywords like e.g. the modifiers in Java. The existence of a keyword can be assigned using the boolean assignment operator „?“.

Example:

```
Entity :
    (abstract?"abstract")? "entity" name=ID ("<" extends=ID)?
    "{"
    (features+=Feature)*
    "}";
```

With this the type `Entity` will have the boolean property `abstract`, which is set to true if the respective keyword has been specified for an entity. (I've added the `extends` part, because an abstract entity wouldn't make sense without inheritance)

Identifier Token

We also have seen the identifier token (ID). This is the token rule expressed in Antlr grammar syntax:

```
('a'..'z'|'A'..'Z'|'_' ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*)
```

So an identifier is a word starting with a character or underscore followed by optionally additional characters, underscores or digits. The return value of the Identifier token is a String. So if you use the usual assignment operator „=“, the feature the value is assigned to is of type String. You could also use the boolean operator and the type will be Boolean.

String Token

There is also a built-in String Token. Here is an example:

```
Attribute :
    type=DataType name=ID (description=STRING)? ";"
```

With this one can optionally specify a description for an entity such in this example:

```
entity Customer {
    string name ;
    string street "should include the street number, too.";
    int age;
    bool isPremiumCustomer;
}
```

By default the two string literal syntaxes "my text" and 'my text' are supported. Note that unlike in Java also multiline strings are supported:

```
entity Customer {
    string name ;
    string street "should include the street number, too.
                  And if you don't want to specify it, you
                  should definitely consider to specify it
                  somewhere else (in another company)";
    int age;
    bool isPremiumCustomer;
}
```

Specifying the string literal terminals.

If you, for some reason, need other string literal terminal sequences, you can make use of the parameterized String Token. Here's an Example:

```
Attribute :
    type=DataType name=ID (description=STRING("<!--", "-->"))?
    ";"
```

With this the customer example should look like this:

```
entity Customer {  
    string name ;  
    string street <!--should include the street number, too.-->;  
    int age;  
    bool isPremiumCustomer;  
}
```

Single Line Token

Sometimes you don't want to enclose text within terminals, but instead want to get everything between a character sequence and the end of line.

```
Attribute :  
    (comment=LINE ("///  
    type=DataType name=ID;
```

So we can add comments to our attributes like this:

```
entity Customer {  
    string name;  
    ///  
    string street;  
    int age;  
    bool isPremiumCustomer;  
}
```

Comments

There are two different kinds of comments automatically available in any Xtext language.

```
// single-line comments  
  
/*  
    mutli-line comments  
*/
```

Note that those comments are ignored by the language's parser (i.e. they are not contained in the AST returned from the parser).

The Generator

It's assumed that you've used the Xtext Projects Wizard and that you have successfully written an Xtext grammar file describing your little language. Next up you need to start Xtext's generator in order to get a respective parser, metamodel and editor. To do so just right-click the workflow file (*.oaw) located next to the grammar file and choose „Run As“-> „Run Workflow“. The generator will read the grammar file in and create a bunch of files. Some of them located in the 'src-gen' directory others located in the 'src' directory.

Generated and manual code

Any textual artifacts located in the 'src' dir (of any project) will always stay untouched. The generator just creates them the first time when they don't exist.

Files generated to the 'src-gen' directory should never be touched! The whole directory will be wiped out the next time one starts the generator.

The different projects

The wizard creates 2 (optionally 3) different projects.

The main project

The name of the main project can be specified in the wizard. This project contains the main language artifacts and is 100% eclipse independent. The default locations of the most important resources are:

<i>Location</i>	<i>Description</i>
src/[dslname].txt	The grammar file, containing the grammar rules describing your DSL
src/generate.oaw	The workflow file for the Xtext generator.
src/generator.properties	Properties passed to the Xtext generator
src/[base.package.name]/[dslname]Checks.chk	The Check-file used by the parser and within the editor. Used to add semantic constraints to your language.
src-gen/[base.package.name]/[dslname].ecore	Metamodel derived from the grammar
src-gen/[base.package.name]/parser/*	Generated Antlr parser artifacts

The editor project

The name of the editor project is derived from the main project's name by appending the suffix `.editor` to it. The editor project contains the Eclipse Texteditor specific informations. Note that it uses a generic `xtext.editor` plugin, which does most of the job. These are the most important resources:

<i>Location</i>	<i>Description</i>
src/[base.package.name]/[dslname]EditorExtensions.ext	The Xtend-file is used by the outline view. If you want to customize the labels of the outline view, you can do that here.
src-gen/[base.package.name]/[dslname]Utilities.java	Contains all the important DSL-specific information. You should subclass it in order to overwrite the default behaviours.
src/[base.package.name]/[dslname]EditorPlugin.java	If you have subclassed the *Utilities class, make sure to change the respective instantiation here.

The generator project

The name of the generator project is derived from the main project's name by appending the suffix `.generator` to it. The generator project is intended to contain all needed generator resources such as Xpand templates, platform-specific Xtend files etc..

These are the most important resources:

<i>Location</i>	<i>Description</i>
src/[base.package.name]/generator.oaw	The generators workflow preconfigured with the generated DSL parser and the Xpand component. As this is just a proposal, feel free to change/add the workflow as you see fit.
src-gen/[base.package.name]/Main.xpt	The proposed entry Xpand template file.

Future enhancements

As Xtext is in an early development phase, we would be happy to get some feedback about missing or superfluous features.

We have plans to

- add a global model repository into eclipse so one can access all the available models using extensions. This allows to validate over multiple (different) resources and models.
- add the possibility to describe Expressions
- add an inclusion feature so you can reuse sublanguages (such as OCL expressions) within your language
- migrate to Antlr 3.0

About our Sponsors

itemis GmbH & Co. KG is an independent IT service company with an emphasis on consulting, coaching, and software development. Every single itemis expert provides many years of project experience and widespread knowledge about all object oriented and component based software development issues - especially in the field of model driven software development.

b+m is the founder of the openArchitectureWare project. The software was originally developed within the scope of many successful projects. b+m opened the software to the community in late 2003. All of the paradigms of Model-Driven Software Development including Product Line Engineering and not only the generator framework have become a key concept for product and customer specific development at b+m. b+m customers can make use of long time experience and substantial know-how in that field. Located at the company headquarters in Melsdorf/Kiel and at its subsidiaries in Berlin, Cottbus, Hamburg, Hanover and Kiel the b+m staff of 205 provides practical solutions for customized business applications, business process optimization and comprehensive architecture, project and quality management.

oose Innovative Informatik GmbH offers coaching, consulting and training in all themes about software engineering. The main focus of their activities are software architecture, requirements engineering and project-management. oose have first-hand information and experience, because our staff take actively part with others in actual trends, standards and innovations. Our staff support this and pass their know-how regularly on by writing and publishing books or being speaker at conferences, etc. Within the OMG oose collaborate actively on the specifications of the UML and also the SysML.

MID Enterprise Software Solutions GmbH is a leading supplier of optimized tool environments for standardsbased and model-centric software development as well as business process modeling. This includes professional tool consulting and tool components to build a complete tool environment using the best techniques and tool modules available - Architectural and Operational Excellence. With innovatorAOX, MID provides a holistic standard tool environment for object- and function-oriented software development as well as business process and data modeling to help its customers establish highly efficient processes and tool environments for software production, The unique and seamless integration of business process modeling into the development process ensures an unprecedented level of convergence of business requirements and implemented IT systems. Project members from all departments speak the same language and all requirements are clearly described.