Xtext User Guide

Heiko Behrens, Michael Clay, Sven Efftinge, Moritz Eysholdt, Peter Friese,
Jan Köhnlein, Knut Wannheden, Sebastian Zarnekow and contributors

Chapter 1. Overview

1.1. What is Xtext?

Xtext is a framework for the development of domain-specific languages and other textual programming languages. It is tightly integrated with the Eclipse Modeling Framework (EMF) and leverages the Eclipse Platform in order to provide a language-specific integrated development environment (IDE).

In contrast to common parser generators (like e.g. JavaCC or ANTLR), Xtext derives much more than just a parser and lexical analyzer (lexer) from an input grammar. The grammar language is used to describe and generate:

- an incremental, ANTLR 3 based parser and lexer to read your models from text,
- Ecore models (optional),
- a serializer to write your models back to text,
- a linker, to establish cross links between model elements,
- an implementation of the EMF Resource interface with full support for loading and saving EMF models, and
- an integration of the language into your Eclipse IDE.

Some of the IDE features, that are either derived from the grammar or easily implementable, are

- syntax coloring,
- model navigation (F3, etc.),
- code completion,
- outline view, and
- code templates.

The generated artifacts are wired up through Google Guice, a dependency injection framework which makes it easy to exchange certain functionality in a non-invasive manner.

Although Xtext aims at supporting fast iterative development of domain-specific languages, it can be used to implement IDEs for general purpose programming languages as well.

1.2. What is a domain-specific language

A domain-specific language (DSL) is a small programming language, which focuses on a particular domain. Such a domain can be more or less anything. The idea is that its concepts and notation is as close as possible to what you have in mind when you think about a solution in that domain. Of course we are talking about problems which can be solved or processed by computers somehow.

The opposite of a DSL is a so called GPL, a General Purpose Language such as Java or any other common programming language. With a GPL you can solve every computer problem, but it might not always be the best way to solve it.

Imagine you want to remove the core from an apple. You could of course use a Swiss army knife to cut it out, and this is reasonable if you have to do it just once or twice. But if you need to do that on a regular basis it might be more efficient to use an apple corer.

There are a couple of well-known examples of DSLs. For instance SQL is actually a DSL which focuses on querying relational databases. Other DSLs are regular expressions or even languages provided by tools like MathLab. Also most XML languages are actually domain-specific languages. The whole purpose of XML is to allow for easy creation of new languages. Unfortunately with XML you are not able to change the concrete syntax, which is the major problem with it. The concrete syntax of XML is way too verbose. Also a generic syntax for everything is a compromise.

Xtext is a sophisticated framework that helps to implement your very own DSL with appropriate IDE support. There is no such limitation as with XML, you are free to define your concrete syntax as you like. It may be as concise and suggestive as possible being a best match for your particular domain. The hard task of reading your model, working with it and writing it back to your syntax is greatly simplified by Xtext.

Chapter 2. Getting Started

In this mini-tutorial you will implement your first language with Xtext and create an editor from that. Later, we will create a code generator that is capable of reading the models you create with the DSL editor and process them.

2.1. Creating a DSL

Download and install the latest version of Xtext, set up a fresh workspace, take a deep breath and follow the instructions. As soon as you have finished this chapter you will have an editor that understands input files of the following format. We will refer to this snippet as an example of our “target syntax” later on:

```
type String
type Bool

entity Session {
  property Title: String
  property IsTutorial : Bool
}

entity Conference {
  property Name : String
  property Attendees : Person[]
  property Speakers : Speaker[]
}

entity Person {
  property Name : String
}

entity Speaker extends Person {
  property Sessions : Session[]
}
```

2.1.1. Create an Xtext project

Use the Xtext wizard to create a new project

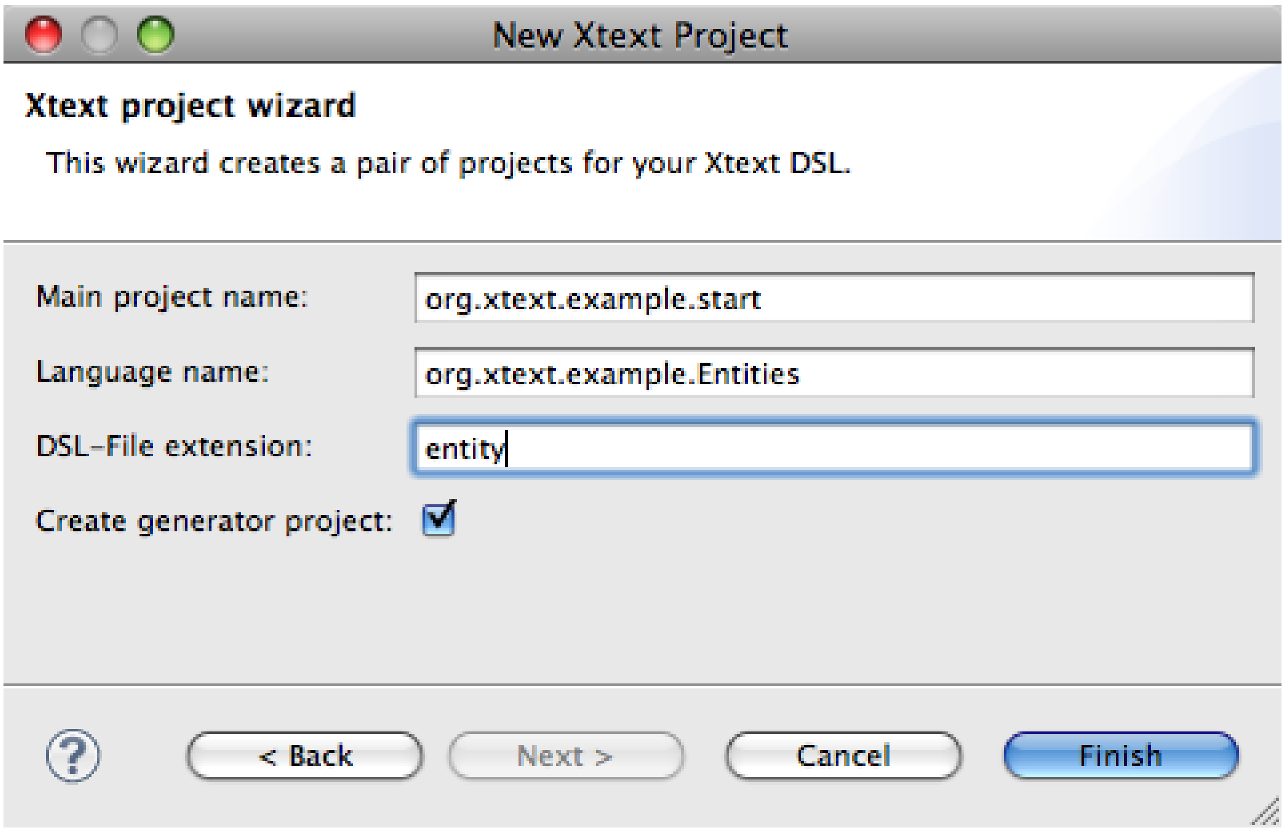
File -> *New* -> *Project...* -> *Xtext* -> *Xtext project*

Choose a meaningful project name, language name and file extension, e.g.

Main project name:	org.xtext.example.start
Language name:	org.xtext.example.Entities
DSL-File extension:	entity

Keep “Create generator project” checked, as we will also create a code generator in a second step.

Click on *Finish* to create the projects.


New Xtext Project

Xtext project wizard

This wizard creates a pair of projects for your Xtext DSL.

Main project name: org.xtext.example.start

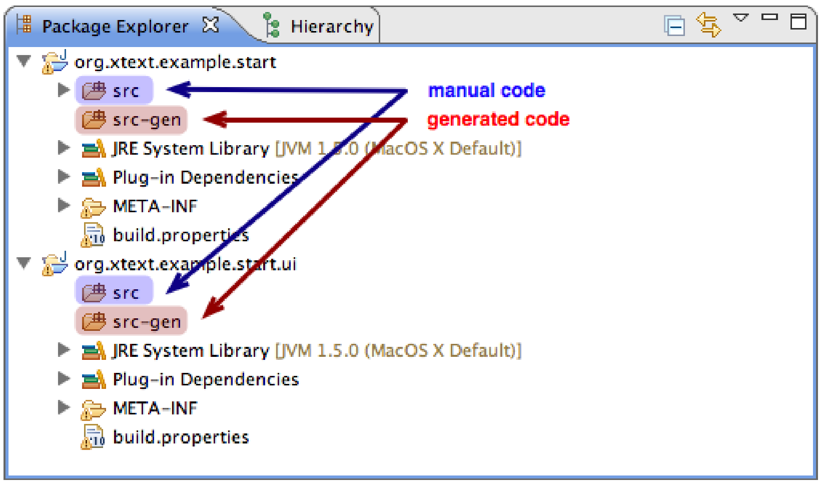
Language name: org.xtext.example.Entities

DSL-File extension: entity

Create generator project: ✓

2.1.2. Project layout

In the Package Explorer you can see three new projects. In `org.xtext.example.start` you can define the grammar and configure the runtime aspects of your language. The editor, outline view and code completion goes into `org.xtext.example.start.ui`. Both projects consist of generated classes derived from your grammar and manual code such as the grammar itself or further classes to differentiate from the default behavior.



It is good to be clear and unambiguous whether the code is generated or is to be manipulated by the developer. Thus, the generated code should be held separately from the manual code. We follow this pattern by having a folder src/ and a folder src-gen/ in each project. Keep in mind not to make changes in the src-gen/ folder. They will be overwritten by the generator.

A third project, `org.xtext.example.start.generator` will later contain a code generator that leverages the model created with the DSL editor.

2.1.3. Build your own grammar

The wizard will automatically open the example grammar file `Entities.xtext` from the first project in the editor. A grammar has two purposes. First, it is used to describe the concrete syntax of your language. Second, it contains information about how a parser shall create a model during parsing.

Ignore the generated sample grammar, delete everything after " `Model :`" to the end. The entry rule for the parser will be called `Model`. As a `Model` consists of one or more `Entity` entries, this rule delegates to another rule named `Entity`, which will be defined later on. As we can have one ore more entities within a model, the cardinality is “+”. Each rule is terminated with a semicolon. So our first rule reads as

```
Model :
  Entity+;
```

Please note: *If you encounter strange errors while copying and pasting these snippets to your Eclipse editor your documentation viewer most likely has inserted characters different from {space} into your clipboard. Reenter these “fillers” or type the text by hand to be sure everything works fine.*

An Xtext grammar does not only describe rules for the parser but also the structure of the resulting abstract syntax tree. Usually, each parser rule will create a new node in that tree. The type of that node can be specified after the rule name using the keyword `returns`. If the type’s name is the same as the rule name, it can be omitted as in our case.

The parser will create a new element of type `Model` when it enters the rule `Model`, and a new element of type `Entity` every time it enters the rule `Entity`. To connect these AST elements, we have to define the name of a reference. In our case, we call that reference *entities*. We specify it using the assignment operator " `+=`", which denotes a multi valued feature.

As a result, we modify the first rule to

```
Model :
  (elements += Entity)+;
```

The next rule on our list is the rule `Entity`. Looking at our target syntax, each entity begins with the keyword `entity` followed by the entity’s name and an opening curly brace (we will handle the `extends` clause in a second step). Then, an entity defines a number of properties and ends with a closing curly brace.

```
Entity returns Entity:
  'entity' name=ID '{'
    (properties+=Property)*
  '}'
;
```

Instead of creating a new AST node for the name, we rather want the name to be an attribute of the `Entity` class. Therefore we use the terminal rule `ID`, which results in a string. The assignment operator " `=`" denotes a single valued feature, and the asterisk a cardinality of 0..n. In our target syntax, some entities refer to an existing entity as their super type after the keyword `extends`. Note that this is a cross-reference, as the super type itself must be defined somewhere else. To define a cross-reference we use square brackets. Optional parts have the cardinality “?”. The complete rule now reads:

```
Entity returns Entity:
  'entity' name=ID ('extends' extends=[Entity])? '{'
    (properties+=Property)*
  '}'
;
```

We have not specified the rule `Property`, yet. In our target syntax, properties can refer to simple types such as “String” or “Bool” as well as entities. To make this easy we will first introduce a common supertype `Type` each `Property` can refer to.

Change the rule `Model` and introduce a new rule `Type` and `SimpleType`:

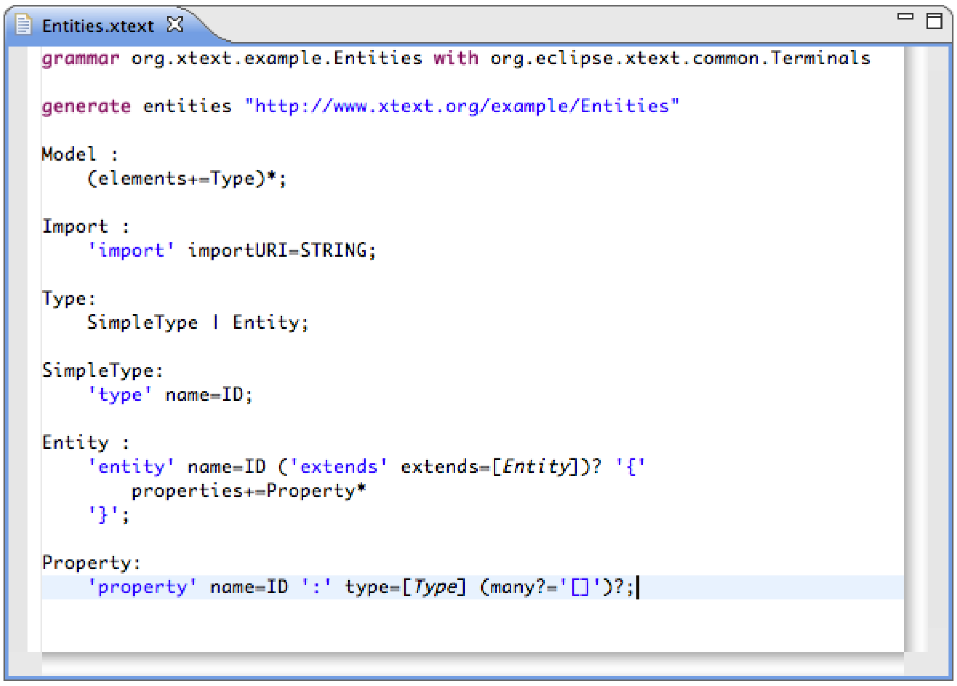
```
Model :
  (elements+=Type)*;

Type:
  SimpleType | Entity;

SimpleType:
  'type' name=ID;
```

Models new consist of types where a `Type` can either be a `SimpleType` or the `Entity` you already know. Our rule `Type` will just delegate to either of them, using the " `|`" alternatives operator. The combination of simple data types and entites this way introduces a common super type `Type` both `Entity` and `SimpleType` derive from. This allows you to refer to both types of elements with a single cross-reference.

A `Property` consist of a keyword, a name, a colon and a cross-reference to an arbitrary `Type`. The multiplicity is either many or one. The presence of the postfix "[]" (technically a keyword) should trigger a boolean flag in the AST model. This is the purpose of the assignment operator " `?=`". Our last parser rule is:

```
Property:
  'property' name=ID ':' type=[Type] (many?='[]')?;
```

In the end your grammar editor should look like this:



```
grammar org.xtext.example.Entities with org.eclipse.xtext.common.Terminals

generate entities "http://www.xtext.org/example/Entities"

Model :
    (elements+=Type)*;

Import :
    'import' importURI=STRING;

Type:
    SimpleType | Entity;

SimpleType:
    'type' name=ID;

Entity :
    'entity' name=ID ('extends' extends=[Entity])? '{'
        properties+=Property*
    '}';

Property:
    'property' name=ID ':' type=[Type] (many?='[]')?;
```

2.1.4. Generate language artifacts

Save the grammar and make sure that no error markers appear. Then, locate the file GenerateEntities.mwe next to the grammar file in the package explorer view. From its context menu, choose

Run As -> MWE Workflow.

That will trigger the Xtext language generator. You will see its logging messages in the Console view.

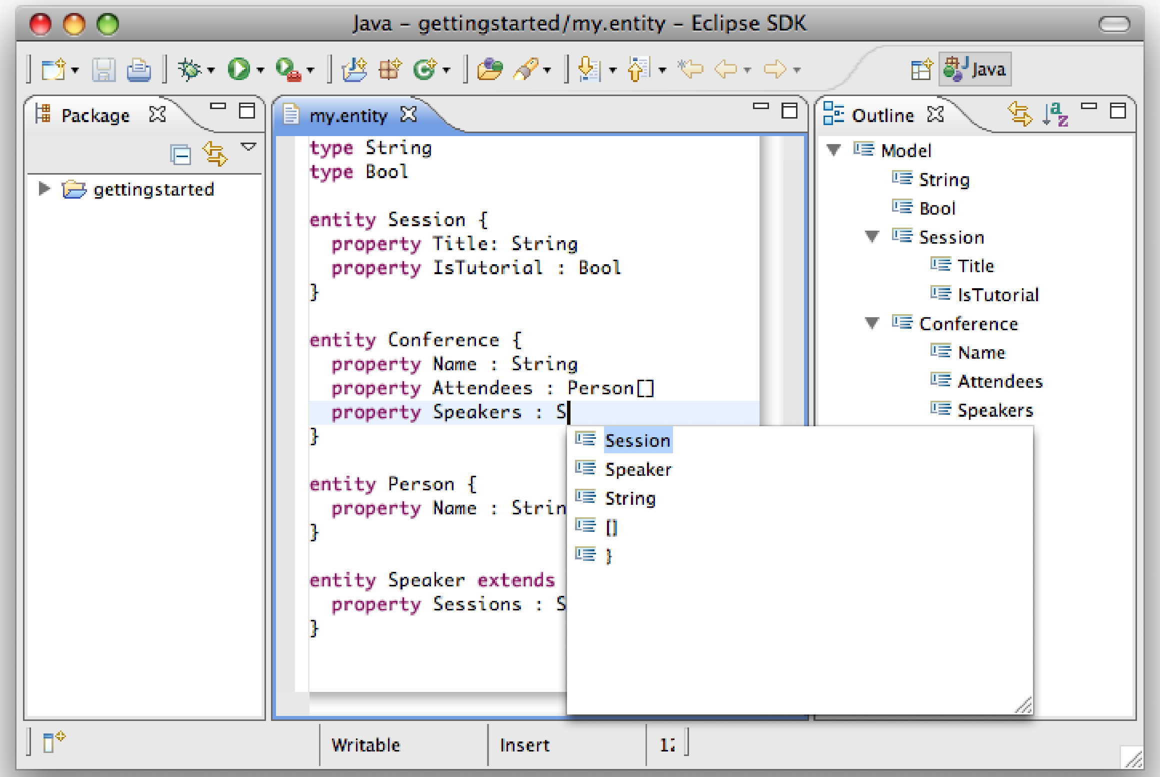
2.1.5. Run the generated editor

If code generation succeeded, right-click on the Xtext project and choose

Run As -> Eclipse Application.

This will spawn a new Eclipse workbench with your projects as plug-ins installed. In the new workbench, create a new project (*File -> New -> Project... -> General -> Project*) and therein a new file with the file extension you chose in the beginning. This will open the generated entity editor.

To get some hands-on experience with your new DSL editor, type in the following model:



```
type String
type Bool

entity Session {
  property Title: String
  property IsTutorial : Bool
}

entity Conference {
  property Name : String
  property Attendees : Person[]
  property Speakers : S
}

entity Person {
  property Name : Strin
}

entity Speaker extends
  property Sessions : S
}
```

2.2. Writing a code generator

In this part of the tutorial, we will write a code generator that is capable of processing the models created with the DSL editor you developed in the previous section.

2.2.1. Creating the main template

One of the major goals of model driven software development is to raise the level of abstraction. The concepts in your meta model usually map to several artifacts in your source code. In our sample, we will generate the following things for each entity:

- a data access object (DAO), capable of loading and storing the entities
- a class holding all the attributes of the entity, annotated with JPA annotations

To provide for a better overview and to easier manage our code templates, we will choose the following template structure:

- Main.xpt – the main entry point, will dispatch to all other templates
- DAO.xpt – will generate the *DAO* class
- Entity.xpt – will generate the *Entity* class

In the project navigator, right click on *org.xtext.example.start.generator/templates* and select *New -> Other... -> Xpand Template*. Name the new template `Main.xpt` and click on *Finish*.

First, we need to import the meta model, as we're going to be working with the concepts from the meta model:

```
´IMPORT entitiesª
```

Next, we need to define a main template which will be invoked by the code generator. This template will then dispatch to two sub templates, `DAO::dao` and `Entity::entity`:

```
´DEFINE main FOR Modelª
  ´EXPAND DAO::dao FOREACH this.types.typeSelect(Entity)ª
  ´EXPAND Entity::entity FOREACH this.types.typeSelect(Entity)ª
´ENDDEFINEª
```

By using the expression `this.types.typeSelect(Entity)`, we make sure we only iterate over `Entity` elements, and skip `SimpleType` elements.

Your template `Main.xpt` should now look like this:

```
´IMPORT entitiesª
´DEFINE main FOR Modelª
  ´EXPAND DAO::dao FOREACH this.types.typeSelect(Entity)ª
  ´EXPAND Entity::entity FOREACH this.types.typeSelect(Entity)ª
´ENDDEFINEª
```

2.2.2. Creating the template for the entity

Every data-oriented application needs a bunch of classes to hold the data. Usually referred to as entities, these classes are POJOs in our case. So, let’s now create a template which helps us to create POJOs from our model.

Please add another Xpand template to your project by selecting *File* -> *New* -> *Other ...* -> *Xpand Template*. Name the new file `Entity.xpt`, making sure to save it to the same folder as `Main.xpt`.

Add the following code to `Entity.xpt`:

```
´IMPORT entitiesª

´DEFINE entity FOR Entityª
  ´FILE this.name + ".java"ª
    public class ´this.nameª {
      ´EXPAND property FOREACH this.propertiesª
    }
  ´ENDFILEª
´ENDDEFINEª

´DEFINE property FOR Propertyª
  private ´this.type.nameª ´this.nameª;

  public void set´this.name.toFirstUpper()ª(´this.type.nameª ´this.nameª) {
    this.´this.nameª = ´this.nameª;
  }

  public ´this.type.nameª get´this.name.toFirstUpper()ª() {
    return this.´this.nameª;
  }
´ENDDEFINEª
```

2.2.3. Creating the template for the DAO

To load entities from a database and save them back, we will need to write some entities, but again, this is something the generator can do for us.

Create a template `DAO.xpt` and insert this text:

```
´IMPORT entitiesª

´DEFINE dao FOR Entityª
  ´FILE this.name + "DAO.java"ª
    import java.util.Collection;
    import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
    public class ´this.nameªDAO
      extends HibernateDaoSupport {
      ´EXPAND crud FOR thisª
    }
  ´ENDFILEª
´ENDDEFINEª

´DEFINE crud FOR Entityª
  public ´this.nameª load(Long id) {
    return (´this.nameª)getHibernateTemplate().get(´this.nameª.class, id);
  }

  @SuppressWarnings("unchecked")
  public Collection<´this.nameª> loadAll() {
    return getHibernateTemplate().loadAll(´this.nameª.class);
  }

  public ´this.nameª create(´this.nameª entity) {
    return (´this.nameª) getHibernateTemplate().save(entity);
  }

  public void update(´this.nameª entity) {
    getHibernateTemplate().update(entity);
  }

  public void remove(´this.nameª entity) {
    getHibernateTemplate().delete(entity);
  }
´ENDDEFINEª
```

2.2.4. Adjusting the generator workflow

You might have noticed the generator workflow `org.xtext.example.start.generator/src/workflow/EntitiesGenerator.mwe` will invoke a template named `Template.xpt`, so we need to change this to make sure the right templates get invoked. Open `/org.xtext.example.start.generator/src/workflow/EntitiesGenerator.mwe` and make sure it invokes `Main.xpt`:

```
<component class="org.eclipse.xpand2.Generator">
 <metaModel class="org.eclipse.xtend.typesystem.emf.EmfRegistryMetaModel"/>
 <fileEncoding value="ISO-8859-1"/>
 <expand value="templates::Main::main FOR model"/>
 <genPath value="${targetDir}"/>
</component>
```

2.2.5. Running the generator

In order to invoke the generator, select `EntitiesGenerator.mwe` (in `org.xtext.example.start.generator/src/workflow/`) and choose *Run As -> MWE Workflow* from the context menu. You'll see a bunch of log messages in the console view, and after a few seconds, you'll find a number of just generated files in `org.xtext.example.entity.generator/src-gen`.

Please note that you'll get a number of compile-time errors now, as the DAO classes depend on Hibernate. To get rid of them, just download Hibernate from http://www.hibernate.org and add it to the build class path of the project.

2.3. Where next?

This is the end of the “Getting Started” chapter of this documentation. The next part of this document is more detailed and compact at he same time. It discusses technical topics not always in an introductory way but acts as a comprehensive reference. You will find more material that introduces Xtext at the Xtext website .

Chapter 3. The Grammar Language

The grammar language is the corner stone of Xtext and is defined in itself. Actually it is a DSL to create DSLs, so what would be more helpful than developing the Xtext language with Xtext?

The grammar language is a domain-specific language, carefully designed for the description of textual languages based on LL(*)-Parsing that is like ANTLR3's parsing strategy. The main idea is to describe the concrete syntax and how an EMF-based in-memory model is created during parsing.

3.1. A first example

To get an idea of how it works we'll start by implementing a simple example introduced by Martin Fowler. It's mainly about describing state machines used as the (un)lock mechanism of secret compartments. People who have secret compartments control their access in a very old-school way, e.g. by opening the door first and turning on the light afterwards. Then the secret compartment, for instance a panel, opens up. One of those state machines could look like this:

```
events
  doorClosed  D1CL
  drawOpened  D2OP
  lightOn     L1ON
  doorOpened  D1OP
  panelClosed PNCL
end

resetEvents
  doorOpened
end

commands
  unlockPanel PNUL
  lockPanel   PNLK
  lockDoor    D1LK
  unlockDoor  D1UL
end

state idle
  actions {unlockDoor lockPanel}
  doorClosed => active
end

state active
  drawOpened => waitingForLight
  lightOn    => waitingForDraw
end

state waitingForLight
  lightOn => unlockedPanel
end

state waitingForDraw
  drawOpened => unlockedPanel
end

state unlockedPanel
  actions {unlockPanel lockDoor}
  panelClosed => idle
end
```

So, we have a bunch of declared events, commands and states. Within states there are references to declared actions, which should be executed when entering such a state. Also there are transitions consisting of a reference to an event and a state. Please read Martin's description if it is not clear enough.

In order to get a complete IDE for this little language from Xtext, you need to write the following grammar:

```
grammar my.pack.SecretCompartments
   with org.eclipse.xtext.common.Terminals

generate secretcompartment "http://www.eclipse.org/secretcompartment"

Statemachine :
  'events'
     (events+=Event)+
  'end'
  ('resetEvents'
     (resetEvents+=[Event])+
  'end')?
  'commands'
     (commands+=Command)+
  'end'
  (states+=State)+;

Event :
  name=ID code=ID;

Command :
  name=ID code=ID;

State :
  'state' name=ID
     ('actions' '{' (actions+=[Command])+ '}')?
     (transitions+=Transition)*
  'end';

Transition :
  event=[Event] '=>' state=[State];
```

3.2. The Syntax

In the following the different concepts of the grammar language are explained.

3.2.1. Language Declaration

The first line

```
grammar my.pack.SecretCompartments with org.eclipse.xtext.common.Terminals
```

declares the name of the grammar. Xtext leverages Java's class path mechanism. This means that the name can be any valid Java qualifier. The file name needs to correspond to the grammar name and have the file extension 'xtext'. This means that the name has to be `SecretCompartments.xtext` and must be placed in a package `my.pack` somewhere on your project's class path.

The first line is also used to declare any used language (for mechanism details see Grammar Mixins).

3.2.2. EPackage declarations

Xtext parsers instantiate Ecore models (aka meta model). An Ecore model basically consists of an EPackage containing EClasses, EDatatypes and EEnums. Xtext can infer Ecore models from a grammar (see Ecore model inference) but it is also possible to instantiate existing Ecore models. You can even mix this and use multiple existing Ecore models and infer some others from one grammar.

3.2.2.1. EPackage generation

The easiest way to get started is to let Xtext infer the Ecore model from your grammar. This is what is done in the secret compartment example. To do so just state:

```
generate secretcompartment 'http://www.eclipse.org/secretcompartment'
```

That statement means: generate an EPackage with the name `secretcompartment` and the nsURI `http://www.eclipse.org/secretcompartment`. Actually these are the properties that are required to create an EPackage. The whole algorithm used to derive complete Ecore models from Xtext grammars is described in the section Ecore model inference.

3.2.2.2. EPackage import

If you already have an existing EPackage, you can import it using either a namespace URI or a resource URI. An URI (Uniform Resource Identifier) provides a simple and extensible means for identifying an abstract or physical resource. For details about the EMF implementation of this concept, please see its documentation.

Using namespace URIs to import existing EPackages

You can import an existing EPackage with the following statement:

```
import 'http://www.eclipse.org/secretcompartment'
```

Note that if you use a namespace URI, the corresponding EPackage needs to be installed into the workbench. Otherwise the editor cannot find it. At runtime (i.e. when starting the generator) you need to make sure that the corresponding EPackage is registered in the `EPackage.Registry.INSTANCE`. If you use MWE to drive your code generator, you need to add the following lines to your workflow file:

```
<bean class="org.eclipse.emf.mwe.utils.StandaloneSetup"
  platformUri="${runtimeProject}/..">
  <registerGeneratedEPackage value="my.pack.SecretcompartmentPackage"/>
</bean>
```

Using namespace URIs is typically only interesting when common Ecore models are used, such as Ecore itself or the UML meta model. If you're developing the EPackage together with the DSL but don't want to have it derived from the grammar for some reason, we suggest to use a resource URI.

Using resource URIs to import existing EPackages

If the EPackage you want to use is somewhere in your workspace you should refer to it by a `platform:/resource/`-URI. Platform URIs are a special EMF concept, which allow for some kind of transparency between workspace projects and installed bundles. Consult the EMF documentation (we recommend the book) for details.

An import statement referring to an Ecore file by a platform:/resource/ URI looks like this:

```
import 'platform:/resource/project/src/my/pack/SecretCompartments.ecore'
```

If you want to mix generated and imported Ecore models you'll have to configure the generator fragment in your MWE file responsible for generating the Ecore classes (EcoreGeneratorFragment) with resource URIs that point to the genmodels for the referenced Ecore models. Example:

```
<fragment class="org.eclipse.xtext.generator.ecore.EcoreGeneratorFragment"
  genModels=
    "platform:/resource/project/src/my/pack/SecretCompartments.genmodel"/>
```

3.2.2.3. Ecore model aliases for packages

If you want to use multiple EPackages you need to specify aliases in the following way:

```
generate secretcompartment 'http://www.eclipse.org/secretcompartment'
import 'http://www.eclipse.org/anotherPackage' as another
```

When referring to a type somewhere in the grammar you need to qualify the reference using that alias (example `another::CoolType`). We'll see later where such type references occur.

It is also supported to put multiple EPackage imports into one alias. This is no problem as long as there are not any two EClassifiers with the same name. In such cases none of them can be referenced. It is even possible to 'import' multiple and 'generate' one Ecore model and all of them are declared for the same alias. If you do so, for a reference to an EClassifier first the imported EPackages are scanned before it is assumed that a type needs to by generated into the to-be-generated package.

Example:

```
generate toBeGenerated 'http://www.eclipse.org/toBeGenerated'
import 'http://www.eclipse.org/packContainingClassA'
import 'http://www.eclipse.org/packContainingClassB'
```

With the declaration above

1. a reference to type `ClassA` would be linked to the EClass contained in `http://www.eclipse.org/packContainingClassA`,
2. a reference to type `ClassB` would be linked to the EClass contained in `http://www.eclipse.org/packContainingClassB`,
3. a reference to type `NotYetDefined` would be linked to a newly created EClass in `http://www.eclipse.org/toBeGenerated`.

Note, that using this feature is not recommended, because it might cause problems, which are hard to track down. For instance, a reference to `classA` would as well be linked to a newly created EClass, because the corresponding type in `http://www.eclipse.org/packContainingClassA` is spelled with a capital letter.

3.2.3. Rules

The default parsing is based on a home-grown packrat parser. It is advised to substitute it by an ANTLR parser through the Xtext service mechanism. ANTLR is a sophisticated parser generator framework based on an LL(*) parsing algorithm, that works quite well for Xtext. Please download the plugin de.itemis.xtext.antlr (from update site `http://download.itemis.com/updates/`) and use the ANTLR Parser instead of the packrat parser (cf. Xtext Workspace Setup).

Basically parsing can be separated in the following phases.

1. lexing
2. parsing
3. linking
4. validation

3.2.3.1. Terminal Rules

In the first stage, i.e. lexing, a sequence of characters (the text input) is transformed into a sequence of so called tokens. In this context, a token is sort of a strongly typed part of the input sequence. It consists of one or more characters and was matched by a particular terminal rule or keyword and therefore represents an atomic symbol. Terminal rules are also referred to as token rules or lexer rules. There is an informal naming convention that names of terminal rules are all upper-case.

In the secret compartments example there are no explicitly defined terminal rules, since it only uses the ID rule which is inherited from the grammar `org.eclipse.xtext.common.Terminals` (cf. Grammar Mixins). Therein the ID rule is defined as follows:

```
terminal ID :
  ('^')?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

It says that a Token ID starts with an optional '^' character (caret), followed by a letter ('a'..'z'|'A'..'Z') or underscore ('_') followed by any number of letters, underscores and numbers ('0'..'9').

The caret is used to escape an identifier for cases where there are conflicts with keywords. It is removed by the ID rule's ValueConverter.

This is the formal definition of terminal rules:

```
TerminalRule :
  'terminal' name=ID ('returns' type=TypeRef)? ':'
    alternatives=TerminalAlternatives ';'
;
```

Note, that *the order of terminal rules is crucial for your grammar*, as they may hide each other. This is especially important for newly introduced rules in connection with mixed rules from used grammars.

If you for instance want to add a rule to allow fully qualified names in addition to simple IDs, you should implement it as a data type rule, instead of adding another terminal rule.

Return types

A terminal rule returns a value, which is a string (type `ecore::EString`) by default. However, if you want to have a different type you can specify it. For instance, the rule 'INT' is defined as:

```
terminal INT returns ecore::EInt :
  ('0'..'9')+;
```

This means that the terminal rule 'INT' returns instances of `ecore::EInt`. It is possible to define any kind of data type here, which just needs to be an instance of `ecore::EDataType`. In order to tell the parser how to convert the parsed string to a value of the declared data type, you need to provide your own implementation of IValueConverterService (cf. value converters). The value converter is also the point where you can remove things like quotes from string literals or the caret ('^') from identifiers. Its implementation needs to be registered as a service (cf. Service Framework).

3.2.3.2. Extended Backus-Naur form expressions

Token rules are described using "Extended Backus-Naur Form"-like (EBNF) expressions. The different expressions are described in the following. The one thing all of these expressions have in common is the cardinality operator. There are four different possible cardinalities

1. exactly one (the default, no operator)
2. one or none (operator "?")
3. any (zero or more, operator "*")
4. one or more (operator "+")

Keywords / Characters

Keywords are a kind of token rule literals. The 'ID' rule in `org.eclipse.xtext.common.Terminals` for instance starts with a keyword:

```
terminal ID : '^'? .... ;
```

The question mark sets the cardinality to "none or one" (i.e. optional) like explained above.

Note that a keyword can have any length and contain arbitrary characters.

Character Ranges

A character range can be declared using the '..' operator.

Example:

```
terminal INT returns ecore::EInt: ('0'..'9')+;
```

In this case an INT is comprised of one or more (note the '+' operator) characters between (and including) '0' and '9'.

Wildcard

If you want to allow any character you can simple write the wildcard operator '.' (dot): Example:

```
FOO : 'f' . 'o';
```

The rule above would allow expressions like 'foo', 'f0o' or even 'f\no'.

Until Token

With the until token it is possible to state that everything should be consumed until a certain token occurs. The multiline comment is implemented this way:

```
terminal ML_COMMENT : '/*' -> '*/';
```

This is the rule for Java-style comments that begin with '/*' and end with '*/'.

Negated Token

All the tokens explained above can be inverted using a preceding exclamation mark:

```
terminal ML_COMMENT : '/*' (!'*/')+;
```

Rule Calls

Rules can refer to other rules. This is done by writing the name of the rule to be called. We refer to this as rule calls. Rule calls in terminal rules can only point to terminal rules.

Example:

```
terminal QUALIFIED_NAME : ID ('.' ID)*;
```

Alternatives

Using alternatives one can state multiple different alternatives. For instance, the whitespace rule uses alternatives like this:

```
terminal WS : (' '|'\t'|'\r'|'\n')+;
```

That is a WS can be made of one or more whitespace characters (including ‘ ’,‘\t’,‘\r’,‘\n’).

Groups

Finally, if you put tokens one after another, the whole sequence is referred to as a group. Example:

```
terminal ASCII : '0x' ('0'..'7') ('0'..'9'|'A'..'F');
```

That is the 2-digit hexadecimal code of ascii characters.

3.2.4. Parser Rules

The parser reads a sequence of terminals and walks through the parser rules. Hence a parser rule – contrary to a terminal rule – does not produce a single terminal token but a tree of non-terminal and terminal tokens. They lead to a so called parse tree (in Xtext it is also referred as node model). Furthermore, parser rules are handled as kind of a building plan for the creation of the EObjects that form the semantic model (the linked abstract syntax graph or AST). Due to this fact, parser rules are even called production rules. The different constructs like actions and assignments are used to derive types and initialize the semantic objects accordingly.

3.2.4.1. Extended Backus-Naur Form expressions

Not all the expressions that are available in terminal rules can be used in parser rules. Character ranges, wildcards, the until token and the negation are only available for terminal rules.

The elements that are available in parser rules as well as in terminal rules are

1. groups,
2. alternatives,
3. keywords and
4. rule calls.

In addition to these elements, there are some expressions used to direct how the AST is constructed, which are listed and explained in the following.

Assignments

Assignments are used to assign the parsed information to a feature of the current object. The type of the current object, its EClass, is specified by the return type of the parser rule. If it is not explicitly stated it is implied that the type’s name equals the rule’s name. The type of the feature is infered from the right hand side of the assignment.

Example:

```
State :
  'state' name=ID
    ('actions' '{' (actions+=[Command])+ '}')?
    (transitions+=Transition)*
  'end'
;
```

The syntactic declaration for states in the state machine example starts with a keyword ‘state’ followed by an assignment:

```
name=ID
```

The left hand side refers to a feature ' `name`' of the current object (which has the EClass 'State' in this case). The right hand side can be a rule call, a keyword, a cross reference (explained later) or even an alternative comprised by the former. The type of the feature needs to be compatible with the type of the expression on the right. As ID returns an EString in this case, the feature ' `name`' needs to be of type EString as well.

Assignment Operators

There are three different assignment operators, each with different semantics.

1. The simple equal sign "=" is the straight forward assignment, and used for features which take only one element.
2. The "+=" sign (the add operator) expects a multi valued feature and adds the value on the right hand to that feature, which is a list feature.
3. The "?=" sign (boolean assignment operator) expects a feature of type EBoolean and sets it to true if the right hand side was consumed independently from the concrete value of the right hand side.

The used assignment operator does not effect the cardinality of the expected symbols on the right hand side.

Cross References

A unique feature of Xtext is the ability to declare crosslinks in the grammar. In traditional compiler construction the crosslinks are not established during parsing but in a later linking phase. This is the same in Xtext, but we allow to specify crosslink information in the grammar. This information is used by the linker. The syntax for crosslinks is:

```
CrossReference :
  '[' type=TypeRef ('|' ^terminal=CrossReferenceableTerminal )? ']'
;
```

For example, the transition is made up of two cross references, pointing to an event and a state:

```
Transition :
  event=[Event] '=>' state=[State]
;
```

It is important to understand that the text between the square brackets does not refer to another rule, but to a type! This is sometimes confusing, because one usually uses the same name for the rules and the returned types. That is if we had named the type for events differently like in the following the cross reference needs to be adapted as well:

```
Transition :
  event=[MyEvent] '=>' state=[State]
;

Event returns MyEvent : ....;
```

Looking at the syntax definition of cross references, there is an optional part starting with a vertical bar (pipe) followed by 'CrossReferenceableTerminal'. This is the part describing the concrete text, from which the crosslink later should be established. If the terminal is omitted, it is expected to be an ID.

You may even use alternatives as the referencable terminal. This way, either an ID or a STRING may be used as the referencable terminal, as it is possible in many SQL dialects.

```
TableRef: table=[Table|(ID|STRING)];
```

Have a look at the linking section in order to understand how linking is done.

Simple Actions

By default the object to be returned by a parser rule is created lazily on the first assignment. Then the type of the EObject to be created is determined from the specified return type or the rule name if no explicit return type is specified. With Actions however, the creation of returned EObject can be made explicit. Xtext supports two kinds of Actions:

1. simple actions, and
2. assigned actions.

If at some point you want to enforce the creation of a specific type you can use alternatives or simple actions. In the following example TypeB must be a subtype of TypeA. An expression `A ident` should create an instance of TypeA, whereas `B ident` should instantiate TypeB.

Example with alternatives:

```
MyRule returns TypeA :
  "A" name=ID |
  MyOtherRule
;

MyOtherRule returns TypeB :
  "B" name = ID
;
```

Example with simple actions:

```
MyRule returns TypeA :
  "A" name=ID |
  "B" {TypeB} name=ID
;
```

Generally speaking, the instance is created as soon as the parser hits the first assignment. However, actions allow to explicitly instantiate any EObject. The notation `{TypeB}` will create an instance of TypeB and assign it to the result of the parser rule. This allows parser rules without any assignment and object creation without the need to introduce unnecessary rules.

Unassigned rule calls

We previously explained, that the EObject to be returned is created lazily when the first assignment occurs or when a simple action is evaluated. There is another way one can set the EObject to be returned, which we call an “unassigned rule call”.

Unassigned rule calls (the name suggests it) are rule calls to other parser rules, which are not used within an assignment. If there is no feature the returned value shall be assigned to, the value is assigned to the “to-be-returned” result of the calling rule.

With unassigned rule calls one can, for instance, create rules which just dispatch between several other rules:

```
AbstractToken :
  TokenA |
  TokenB |
  TokenC
;
```

As `AbstractToken` could possibly return an instance of `TokenA`, `TokenB` or `TokenC` its type must by a super type of these types. It is now for instance as well possible to further change the state of the AST element by assigning additional things.

Example:

```
AbstractToken :
  ( TokenA |
    TokenB |
    TokenC ) (cardinality=('?'|'+'|'*'))?
;
```

This way the ‘cardinality’ is optional (last question mark) and can be represented by a question mark, a plus, or an asterisk. It will be assigned to either an EObject of type `TokenA`, `TokenB`, or `TokenC` which are all subtypes of `AbstractToken`. The rule in this example will never create an instance of `AbstractToken` directly.

Assigned Actions

LL parsing has some significant advantages over LR algorithms. The most important ones for Xtext are, that the generated code is much simpler to understand and debug and that it is easier to recover from errors. Especially ANTLR has a very nice generic error recovery mechanism. This allows to construct an AST even if there are syntactic errors in the text. You wouldn't get any of the nice IDE features as soon as there is one little error, if we hadn't error recovery.

However, LL also has some drawbacks. The most important one is that it does not allow left recursive grammars. For instance, the following is not allowed in LL based grammars, because "Expression '+' Expression" is left recursive:

```
Expression :
  Expression '+' Expression |
  '(' Expression ')' |
  INT
;
```

Instead one has to rewrite such things by "left-factoring" it:

```
Expression :
  TerminalExpression ('+' Expression)?
;

TerminalExpression :
  '(' Expression ')' |
  INT
;
```

In practice this is always the same pattern and therefore not that problematic. However, by simply applying the Xtext AST construction features we've covered so far, a grammar ...

```
Expression :
  {Operation} left=TerminalExpression (op='+' right=Expression)?
;

TerminalExpression returns Expression:
  '(' Expression ')' |
  {IntLiteral} value=INT
;
```

... would result in unwanted elements in the AST. For instance the expression " (42) " would result in a tree like this:

```
Operation {
  left=Operation {
    left=IntLiteral {
      value=42
    }
  }
}
```

Typically one would only want to have one instance of `IntLiteral` instead.

One can solve this problem using a combination of unassigned rule calls and assigned actions:

```
Expression :
  TerminalExpression ({Operation.left=current}
    op='+' right=Expression)?
;

TerminalExpression returns Expression:
  '(' Expression ')' |
  {IntLiteral} value=INT
;
```

In the example above `{Operation.left=current}` is a so called tree rewrite action, which creates a new instance of the stated EClass (Operation in this case) and assigns the element currently to-be-returned (`current` variable) to a feature of the newly created object (in this case feature 'left' of the Operation instance). In Java these semantics could be expressed as:

```
Operation temp = new Operation();
temp.setLeft(current);
current = temp;
```

3.2.5. Hidden terminal symbols

Because parser rules describe not a single token, but a sequence of patterns in the input, it is necessary to define the interesting parts of the input. Xtext introduces the concept of hidden tokens to handle semantically unimportant things like whitespaces, comments, etc. in the input sequence gracefully. It is possible to define a set of terminal symbols, that are hidden from the parser rules and automatically skipped when they are recognized. Nevertheless, they are transparently woven into the node model, but not relevant for the semantic model.

Hidden terminals may (or may not) appear between any other terminals in any cardinality. They can be described per rule or for the whole grammar. When reusing a single grammar its definition of hidden tokens is reused as well. The grammar `org.eclipse.xtext.common.Terminals` comes with a reasonable default and hides all comments and whitespace from the parser rules.

If a rule defines hidden symbols, you can think of a kind of scope that is automatically introduced. Any rule that is called from the declaring rule uses the same hidden terminals as the calling rule, unless it defines other hidden tokens itself.

```
Person hidden(WS, ML_COMMENT, SL_COMMENT):
  name=Fullname age=INT ';'
;

Fullname:
  (firstname=ID)? lastname=ID
;
```

The sample rule “Person” defines multiline comments (ML_COMMENT), single-line comments (SL_COMMENT), and whitespace (WS) to be allowed between the ‘Fullname’ and the ‘age’. Because the rule ‘Fullname’ does not introduce another set of hidden terminals, it allows the same symbols to appear between ‘firstname’ and ‘lastname’ as the calling rule ‘Person’. Thus, the following input is perfectly valid for the given grammar snippet:

```
John /* comment */ Smith // line comment
/* comment */
      42      ; // line comment
```

A list of all default terminals like WS can be found in section Grammar Mixins.

3.2.6. Data type rules

Data type rules are parsing-phase rules, which create instances of EDataType instead of EClass. Thinking about it, one may discover that they are quite similar to terminal rules. However, the nice thing about data type rules is that they are actually parser rules and are therefore

1. context sensitive and
2. allow for use of hidden tokens.

If you, for instance, want to define a rule to consume Java-like qualified names (e.g. “foo.bar.Baz”) you could write:

```
QualifiedName :
  ID ('.' ID)*
;
```

This looks similar to the terminal rule we’ve defined above in order to explain rule calls. However, the difference is that because it is a parser rule and therefore only valid in certain contexts, it won’t conflict with the rule ID. If you had defined it as a terminal rule, it would possibly hide the ID rule.

In addition when this has been defined as a data type rule, it is allowed to use hidden tokens (e.g. “/* comment **/”) between the IDs and dots (e.g. `foo/* comment */. bar . Baz"`)

Return types can be specified in the same way as in terminal rules:

```
QualifiedName returns ecore::EString :
  ID ('.' ID)*
;
```

Note that if a rule does not call another parser rule and does neither contain any actions nor assignments, it is considered to be a data type rule and the data type EString is implied if none has been explicitly declared.

For conversion again value converters are responsible (cf. value converters).

3.2.7. Enum Rules

Enum rules return enumeration literals from strings. They can be seen as a shortcut for data type rules with specific value converters. The main advantage of enum rules is their simplicity, type safety and therefore nice validation. Furthermore it is possible to infer enums and their respective literals during the Ecore model transformation.

If you want to define a ChangeKind org.eclipse.emf.ecore.change/model/Change.ecore with ‘ADD’, ‘MOVE’ and ‘REMOVE’ you could write:

```
enum ChangeKind :
  ADD | MOVE | REMOVE
;
```

It is even possible to use alternative literals for your enums or reference an enum value twice:

```
enum ChangeKind :
  ADD = 'add' | ADD = '+' |
  MOVE = 'move' | MOVE = '->' |
  REMOVE = 'remove' | REMOVE = '-'
;
```

Please note, that Ecore does not support unset values for enums. If you formulate a grammar like

```
Element: "element" name=ID (value=SomeEnum)?;
```

with the input of

```
element Foo
```

the resulting value of the element `Foo` will hold the enum value with the internal representation of ‘0’ (zero). When generating the EPackage from your grammar this will be the first literal you define. As a workaround you could introduce a dedicated none-value or order the enums accordingly. Note that it is not possible to define an enum literal with an empty textual representation.

```
enum Visibility:
  package | private | protected | public
;
```

You can overcome this by modifying the infered Ecore model through a model to model transformation.

3.3. Ecore model inference

The Ecore model (or meta model) of a textual language describes the structure of its abstract syntax trees (AST).

Xtext uses Ecore’s EPackages to define Ecore models. Ecore models are declared to be either inferred (generated) from the grammar or imported. By using the ‘generate’ directive, one tells Xtext to derive an EPackage from the grammar.

3.3.1. Type and Package Generation

Xtext creates

- an EPackage
  - for each generate-package declaration. After the directive ‘generate’ a list of parameters follows. The ‘name’ of the EPackage will be set to the first parameter, its ‘nsURI’ to the second parameter.

  An optional alias as the third parameter allows to distinguish generated EPackages later. Only one generated package declaration per alias is allowed.

- an EClass
  - for each return type of a parser rule. If a parser rule does not define a return type, an implicit one with the same name as the rule itself is assumed. You can specify more than one rule that return the same type but only one EClass will be generated.
  - for each type defined in an action or a cross reference.
- an EEnum
  - for each return type of an enum rule.
- an EDatatype
  - for each return type of a terminal rule or a data type rule.

All EClasses, EEnums and EDatatypes are added to the EPackage referred to by the alias provided in the type reference they were created from.

3.3.2. Feature and Type Hierarchy Generation

While walking through the grammar, the algorithm keeps track of a set of the currently possible return types to add features to.

- Entering a parser rule the set contains only the return type of the rule.
- Entering a group in an alternative the set is reset to the same state it was in when entering the first group of this alternative.
- Leaving an alternative the set contains the union of all types at the end of each of its groups.
- After an optional element, the set is reset to the same state it was before entering it.
- After a mandatory (non-optional) rule call or mandatory action the set contains only the return type of the called rule or action.
- An optional rule call does not modify the set.
- A rule call is optional, if its cardinality is ‘?’ or ‘*’.

While iterating the parser rules Xtext creates

- an EAttribute in each current return type
  - of type EBoolean for each feature assignment using the ‘?=’ operator. No further EReferences or EAttributes will be generated from this assignment.
  - for each assignment with the ‘=’ or ‘+=’ operator calling a terminal rule. Its type will be the return type of the called rule.
- an EReference in each current return type
  - for each assignment with the ‘=’ or ‘+=’ operator in a parser rule calling a parser rule. The EReference’s type will be the return type of the called parser rule.
  - for each assigned action. The reference’s type will be set to the return type of the current calling rule.

Each EAttribute or EReference takes its name from the assignment or action that caused it. Multiplicities will be 0...1 for assignments with the ‘=’ operator and 0...* for assignments with the ‘+=’ operator.

Furthermore, each type that is added to the currently possible return types automatically extends the current return type of the parser rule. You can specify additional common super types by means of “artificial” parser rules, that are never called, e.g.

```
CommonSuperType:
  SubTypeA | SubTypeB | SubTypeC;
```

3.3.3. Enum Literal Generation

For each alternative defined in an enum rule, the transformer creates an enum literal, when another literal with the same name cannot be found. The ‘literal’ property of the generated enum literal is set to the right hand side of the declaration. If it is omitted, you will get an enum literal with equal ‘name’ and ‘literal’ attributes.

```
enum MyGeneratedEnum:
  NAME = 'literal' | EQUAL_NAME_AND_LITERAL;
```

3.3.4. Feature Normalization

In the next step the generator examines all generated EClasses and lifts up similar features to super types if there is more than one subtype and the feature is defined in every subtypes. This does even work for multiple super types.

3.3.5. Customized Post Processing

As a last step, the generator invokes the post processor for every generated Ecore model. The post processor expects an Xtend file with name `<MyDsl>PostProcessor.ext` (if the name of the grammar file is `MyDsl.xtext`) in the same folder as the grammar file. Further, for a successful invocation, the Xtend file must declare an extension with signature `process(xtext::GeneratedMetamodel)`. E.g.

```
process(xtext::GeneratedMetamodel this) :
  process(ePackage)
;

process(ecore::EPackage this) :
  ... do something
;
```

The invoked extension can then augment the generated Ecore model in place. Some typical use cases are to:

- set default values for attributes,
- add container references as opposites of existing containment references, or
- add operations with implementation using a body annotation.

Great care must be taken to not modify the Ecore model in a way preventing the Xtext parser from working correctly (e.g. removing or renaming model elements).

3.3.6. Error Conditions

The following conditions cause an error

- An EAttribute or EReference has two different types or different cardinality.
- There is an EAttribute and an EReference with the same name in the same EClass.
- There is a cycle in the type hierarchy.
- An new EAttribute, EReference or super type is added to an imported type.
- An EClass is added to an imported EPackage.
- An undeclared alias is used.
- An imported Ecore model cannot be loaded.

3.4. Importing existing Ecore models

With the import directive in Xtext you can refer to existing Ecore models and reuse the types that are declared in an EPackage. Xtext uses this technique itself to leverage Ecore data types.

```
import 'http://www.eclipse.org/emf/2002/Ecore' as ecore
```

Specify an explicit return type to reuse such imported types. Note that this even works for terminal rules.

```
terminal INT returns ecore::EInt : ('0'..'9')+;
```

3.5. Grammar Mixins

Xtext supports the reuse of existing grammars. Grammars that are created via the Xtext wizard use `org.eclipse.xtext.common.Terminals` by default which introduces a common set of terminal rules and defines reasonable defaults for hidden terminals.

```
grammar org.xtext.example.MyDsl with org.eclipse.xtext.common.Terminals

generate myDsl 'http://www.xtext.org/example/MyDsl'

... some rules
```

Mixing in another grammar makes the rules defined in that grammar referable. It is also possible to overwrite rules from the used grammar.

Example :

```
grammar my.SuperGrammar
...
RuleA : "a" stuff=RuleB;
RuleB : "{" name=ID "}";

grammar my.SubGrammar with my.SuperGrammar

Model : (ruleAs+=RuleA)*;

// overrides my.SuperGrammar.RuleB
RuleB : '[' name=ID ']';
```

Note that declared terminal rules always get a higher priority then imported terminal rules.

3.6. Default tokens

Xtext ships with a default set of predefined, reasonable and often required terminal rules. This grammar is defined as follows:

```
grammar org.eclipse.xtext.common.Terminals
  hidden(WS, ML_COMMENT, SL_COMMENT)

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

terminal ID :
  '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')* ;
terminal INT returns ecore::EInt: ('0'..'9')+ ;
terminal STRING :
  '"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'"'|"'"|'\\') | !('\\'|'"') )* '"' |
  "'" ( '\\' ('b'|'t'|'n'|'f'|'r'|'"'|"'"|'\\') | !('\\'|"'") )* "'";
terminal ML_COMMENT : '/*' -> '*/' ;
terminal SL_COMMENT  : '//' !('\n'|'\r')* ('\r'? '\n')? ;
terminal WS  : (' '|'\t'|'\r'|'\n')+ ;
terminal ANY_OTHER: . ;
```

Chapter 4. Configuration

4.1. The Generator

Xtext provides lots of generic implementations for your language’s infrastructure but also uses code generation to generate some of the components. Those generated components are for instance the parser, the serializer, the Ecore model and a couple of convenient base classes for content assist etc.

The generator also contributes to shared project resources such as the plugin.xml, Manifest.MF and the Guice modules.

Xtext’s generator leverages the modeling workflow engine (MWE) from EMFT.

4.1.1. A short introduction to MWE

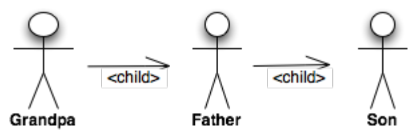
The nice thing about MWE is that it just instantiates Java classes and the configuration is done through setter and adder methods. Given the following Java class :

```
package foo;

public class Person {

  private String name;

  public void setName(String name) {
    this.name = name;
  }

  private final List<Person> children = new ArrayList<Person>();

  public void addChild(Person child) {
    this.children.add(child);
  }
}
```

one can create a family tree this way:

```
<x class="foo.Person">
  <name value="Grandpa"/>
  <child class="foo.Person" name="Father">
    <child name="Son"/>
  </child>
</x>
```

These couple of lines will, when interpreted by MWE, result in an object tree consisting of three instances of `foo.Person`:



The root element can have an arbitrary name, with one exception: If the name is `workflow` and no class attribute is provided, it is assumed that an instance of `org.eclipse.emf.mwe.internal.core.Workflow` shall be instantiated. This instance will be the root of a workflow model used in generator workflow configurations. However, as you can see in the example above one can instantiate arbitrary Java object models. This is conceptually very close to the dependency injection and the XML language in the Spring Framework.

As explained above the element *name* ' `x`' is ignored in this case. The attribute `class` tells MWE which class to use in order to create the object node. The created object is populated according to the XML description: For each XML attribute MWE calls a corresponding setter or adder method passing in the value (there are configurable value converters, but usually Boolean and String is all you need). The same procedure is applied for each child element. In the case of XML elements a single attribute `value` is used and interpreted as if it was an attribute in the parent element. That is:

```
<foo><name value="bar"/></foo>
```

means the same as

```
<foo name="bar"/>
```

Obviously the latter is far more readable and should be preferred. However, as soon as you want to add multiple values to the same 'adder' method you will need to use the first syntax because you cannot define the same attribute twice in XML.

If an element does not have an attribute `value`, the engine looks for an attribute called `class`. If it finds one, the class is instantiated by means of the default constructor (there is no support for factories as of now.). If not, the class is inferred by looking at the argument's type of the current setter method.

Due to this shortcut it is valid to write:

```
<child name="Son"/>
```

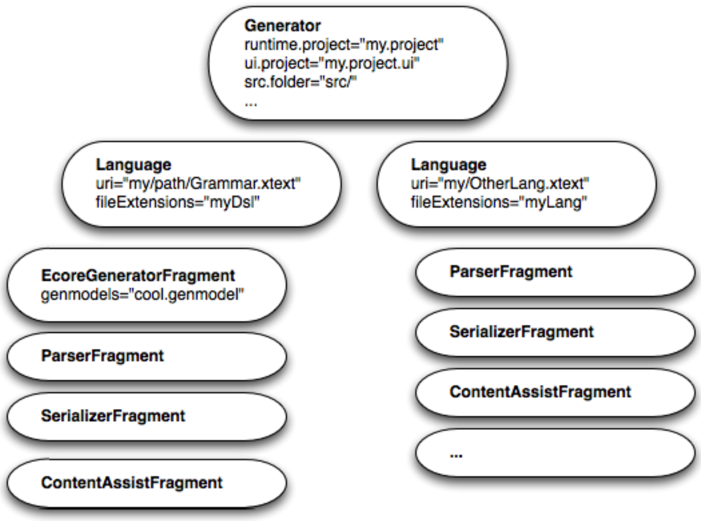
The `addChild` method takes a `foo.Person` as an argument. As this is a concrete class which has a default constructor it can be instantiated.

Tip *Whenever you are in an *.mwe file and wonder what kind of configuration the underlying component may accept: Just use JDT's open type action (CTRL+Shift+T) open the source file of the class in question, use the quick outline view (CTRL+O, use CTRL+O twice to see inherited members as well) and type 'set' or 'add' and you will see the available modifiers. Note that we plan to replace the XML syntax with an Xtext-based implementation as soon as possible.*

This is the basic idea of the MWE language. There are of course a couple of additional concepts and features in the language and we also have not yet talked about the runtime workflow model. Please consult the MWE reference documentation (available through Eclipse help) for additional information.

4.1.2. General Architecture

Now that you know a bit about MWE, you are ready to learn about the concepts and architecture of Xtext's language generator. An instance of Xtext's generator is configured with general information about source folders and projects and consists of any number of language configurations. For each language configuration a URI pointing to its grammar file and the file extensions for the DSL must be provided. In addition, a language is configured with a list of IGeneratorFragments. The whole generator is composed of theses fragments. We have fragments for generating parsers, the serializer, the EMF code, the outline view, etc.



4.1.2.1. Generator Fragments

Each fragment gets the grammar of the language as an EMF model passed in. A fragment is able to generate code in one of the configured locations and contribute to several shared artifacts. The main interface IGeneratorFragment is supported by a convenient abstract base class

AbstractGeneratorFragment, which by default delegates to an Xpand template with the same qualified name as the class and delegates some of the calls to Xpand template definitions.

We suggest to have a look at the fragment we have written for label providers (LabelProviderFragment). It is pretty trivial and at the same time uses the most important call backs. In addition, the structure is not cluttered with too much extra noise so that the whole package (as of Xtext 0.7.0) can serve as a template to write your own fragment.

4.1.2.2. Configuration

As already explained we use MWE from EMFT in order to instantiate, configure and execute this structure of components. In the following we see an exemplary Xtext generator configuration written in MWE configuration code:

```
<workflow>
 <property file="org/xtext/example/GenerateMyDsl.properties"/>

 <property name="runtimeProject" value="../${projectName}"/>

 <bean class="org.eclipse.emf.mwe.utils.StandaloneSetup"
  platformUri="${runtimeProject}/.."/>

 <component class="org.eclipse.emf.mwe.utils.DirectoryCleaner"
  directory="${runtimeProject}/src-gen"/>
 <component class="org.eclipse.emf.mwe.utils.DirectoryCleaner"
  directory="${runtimeProject}.ui/src-gen"/>

 <component class="org.eclipse.xtext.generator.Generator">
  <pathRtProject value="${runtimeProject}"/>
  <pathUiProject value="${runtimeProject}.ui"/>
  <projectNameRt value="${projectName}"/>
  <projectNameUi value="${projectName}.ui"/>

  <language uri="${grammarURI}" fileExtensions="${file.extensions}">
   <!-- Java API to access grammar elements
    (required by several other fragments) -->
   <fragment class=
    "org.eclipse.xtext.generator.grammarAccess.GrammarAccessFragment"/>

   <!-- a lot more simple fragments -->
   <!-- ...  -->

   <!-- a sample fragment with a property -->
   <fragment class=
    "org.eclipse.xtext.generator.validation.JavaValidatorFragment">
    <composedCheck value="org.eclipse.xtext.validation.ImportUriValidator"/>
   </fragment>

      <!-- more simple fragments -->
   <!-- ...  -->
  </language>
 </component>
</workflow>
```

Here the root element is a workflow which accepts *bean* s and *component* s. The `<property/>` element is a first class concept of MWE's configuration language and essentially acts as a preprocessor, which replaces all occurrences of `${propertyName}` with the given value. Property declarations can be defined in-line (`<property name="foo" value="bar"/>`) or by means of a property file import (`<property file="foo.properties"/>`) which is performed before the actual tree is created. In this example we first import a properties file and after that declare a property `runtimeProject` which already uses a property imported from the previously imported file.

The method `Workflow.setBean()` does nothing but provides a means to apply global side-effects, which unfortunately is required by some projects. In this example we do a so called *EMF stand alone* *setup*. This class initializes a bunch of things for a non-OSGi environment that are otherwise configured by means of extension points, e.g. EMF's `EPackage.Registry`.

Following the `<bean/>` element there are three `<component/>` elements. The `Workflow.addComponent()` method awaits instances of `IWorkflowComponent`, which is the primary concept of MWE's workflow model. Xtext's generator is an instance of `IWorkflowComponent` and can therefore be used within MWE workflows.

4.1.3. Standard generator fragments

In the following table the most important standard generator fragments are listed. Please refer to the Javadocs for more detailed documentation.

Class	Generated Artifacts	Related Documentation
EcoreGeneratorFragment	EMF code for generated models	Model inference
XtextAntlrGeneratorFragment	ANTLR grammar, parser, lexer and related services	
GrammarAccessFragment	Access to the grammar	
ResourceFactoryFragment	EMF resource factory	
ParseTreeConstructorFragment	Model-to-text serialization	Serialization
JavaScopingFragment	Java-based scoping	Java-based scoping
JavaValidatorFragment	Java-based model validation	Java-based validation
CheckFragment	Xpand/Check-based model validation	Check-based validation
FormatterFragment	Code formatter	Declarative formatter
LabelProviderFragment	Label provider	Label provider
OutlineNodeAdapterFactoryFragment	Outline view configuration	Outline
TransformerFragment	Outline view configuration	Outline
JavaBasedContentAssistFragment	Java-based content assist	Content assist
XtextAntlrUiGeneratorFragment	Content assist helper based on ANTLR	Content assist
SimpleProjectWizardFragment	New project wizard	Project wizard

Important *Due to IP-Problems with the code generator shipped with ANTLR 3 we're not allowed to ship this fragment at Eclipse. Therefore you'll have to download it separately from http://download.itemis.com or use the update site at* ***http://download.itemis.com/updates/****.*

4.2. Dependency Injection in Xtext with Google Guice

In Xtext, there are many Java classes which implement logic, behavior, or supply configuration. These classes implement Java interfaces and are typically only supposed to be accessed through these interfaces, which makes their implementations interchangeable. This is where Xtext utilizes Google Guice:

- Guice manages the instantiation of the classes: instead of constructing a class with `new`, the `@Inject` annotation instructs Guice to supply an object for a certain interface. Such an object is called Service.
- Guice allows to configure which implementations are to be supplied for which interface. This configuration consists of so-called modules.

4.2.1. Services

The most parts of Xtext are implemented as *services*. A service is an object which implements a certain interface and which is instantiated and provided by Guice. Nearly every concept of the Xtext framework can be understood as this sort of *service*: The XtextEditor, the XtextResource, the IParser and even fine grained concepts as the PrefixMatcher for the content assist are configured and provided by Guice.

Xtext ships with generic default implementations for most of the services or uses generator fragments to automatically generate service implementations for a grammar. Thereby, Xtext strives to provide meaningful implementations out of the box and to allow customization wherever needed. Developers are encouraged to subclass existing services and configure them for their languages in their modules.

When Guice instantiates an object, it also supplies this instance with all its dependent services. All a service does is to request "some implementation for a certain interface" using the `@Inject`-annotation. Based on the modules configuration Guice decides which class to instantiate or which object to reuse.

For example, Guice can automatically initialize member variables with the needed services.

```
public class MyLanguageLinker extends Linker {

  @Inject
  private IScopeProvider scopeProvider;

  @Inject(optional=true)
  private IXtext2EcorePostProcessor postProcessor;

  (...)
}
```

Furthermore, Guice can pass the needed services as method parameters or event into a constructor call.

```
public class MyLanguageGrammarAccess implements IGrammarAccess {

  private final GrammarProvider grammarProvider;

  private TerminalsGrammarAccess gaTerminals;

  @Inject
  public MyLanguageGrammarAccess(GrammarProvider grammarProvider,
    TerminalsGrammarAccess gaTerminals) {
    this.grammarProvider = grammarProvider;
    this.gaTerminals = gaTerminals;
  }

  (...)
}
```

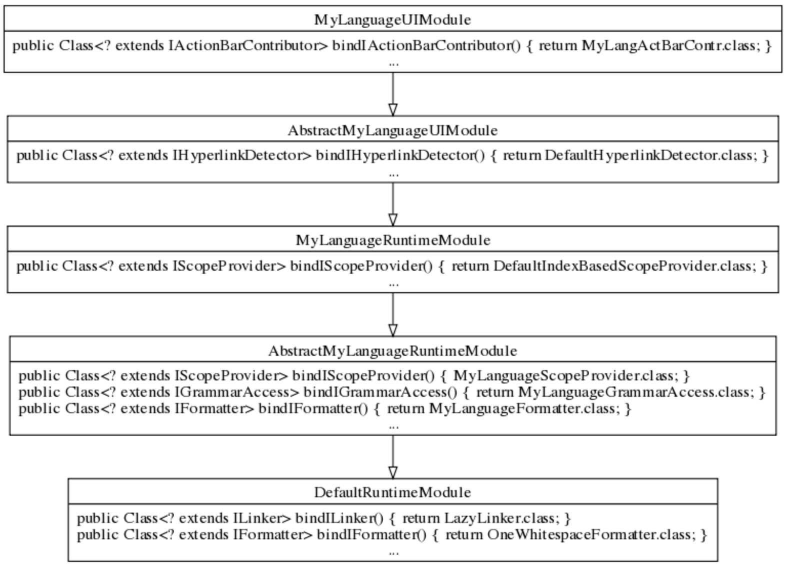
For further details, please refer to the Google Guice Documentation

4.2.2. Modules

The configuration of services for a language built with Xtext is done via modules:

- Modules bind arbitrary Java interfaces to their implementation classes or directly to instances of their implementation classes. Such a binding is sort of a configurable mapping.
- A module itself is a plain Java class.
- Modules can inherit from each other and override bindings that are declared in super-modules. This concept is put on top of plain Guice modules by Xtext.
- In Xtext, there is a generic default module for all languages, there are automatically generated modules and there are modules which are intended to be customized manually.
- Furthermore, Xtext distinguishes between modules for runtime-services and modules related to services needed for the user interface. The UI module extends the runtime module.

In total, this leads to five modules for a typical Xtext Language. They are visualized in the image below. The image is further explained in the following subsections.



```
public Class<? extends IActionBarContributor> bindIActionBarContributor() { return MyLangActBarContr.class; }
...
```

```
public Class<? extends IHyperlinkDetector> bindIHyperlinkDetector() { return DefaultHyperlinkDetector.class; }
...
```

```
public Class<? extends IScopeProvider> bindIScopeProvider() { return DefaultIndexBasedScopeProvider.class; }
...
```

```
public Class<? extends IScopeProvider> bindIScopeProvider() { MyLanguageScopeProvider.class; }
public Class<? extends IGrammarAccess> bindIGrammarAccess() { return MyLanguageGrammarAccess.class; }
public Class<? extends IFormatter> bindIFormatter() { return MyLanguageFormatter.class; }
...
```

```
public Class<? extends ILinker> bindILinker() { return LazyLinker.class; }
public Class<? extends IFormatter> bindIFormatter() { return OneWhitespaceFormatter.class; }
...
```

4.2.2.1. Modules intended for customization

When the generator runs the first time, it creates two modules named `<MyLanguage>RuntimeModule` and `<MyLanguage>UIModule`. They are placed in the language's root-package in the `src/`-folder of the language's runtime-project and the language's UI-project. Both are initially empty and will never be overwritten by the generator. They are intended for customization. By default, they extend a generated module.

4.2.2.2. Generated Modules

The fully generated modules are called `Abstract<MyLanguage>RuntimeModule` and `Abstract<MyLanguage>UiModule` respectively. They contain all components which have been generated specifically for the language at hand. What goes into these modules depends on the fragments you use in the generator.

Note: *This modules are replaced on every subsequent execution of the generator. Don't put any custom code into them. Manually written code has go into the concrete subclasses.*

4.2.2.3. Default Module

Finally the fully generated modules extend the DefaultRuntimeModule, which contains all the default configuration. The default configuration consists of all components for which we have generic default implementations.

4.2.2.4. Changing Configuration

We use the primary modules (`<MyLanguage>RuntimeModule` and `<MyLanguage>UiModule`) in order to change the configuration. These classes are initially empty and have been generated to allow customization.

In order to provide a simple and convenient way, the default module extends the AbstractGenericModule. It does not provide any bindings itself but comes up with a convenient and declarative way to specify mappings. The default API provided by Guice is based on fluent API and a builder pattern. This is also very readable but does not allow submodules to change any of the bindings. The AbstractGenericModule allows to declare and override bindings in all subclasses like this:

```
public Class<? extends IFooService> bindIFooService() {
    return MyFooServiceImpl.class;
}
```

Such a method will be interpreted as a binding from `IFooService` to `MyFooServiceImpl.class`. Note that you simply have to override a method from a super class (e.g. from the generated or default module) in order to change the respective binding. For example, in the picture above, the `DefaultRuntimeModule` configures the IFormatter to be implemented by the

OneWhitespaceFormatter. The `AbstractMyLanguageModule` overrides this binding by mapping the IFormatter to `MyLanguageFormatter`. The type to type binding will create a new instance of the given target type for each dependency. If you want to make it a singleton and thereby ensure only one instance to be created and reused for all dependencies you can add the following annotation:

```
@SingletonBinding
public Class<? extends IFooService> bindIFooService() {
  return MyFooServiceImpl.class;
}
```

In addition if the creation of the type causes any necessary side effects, so you want it to be instantiated eagerly, you can set the `eager` property to true. This is shown in the following snippet:

```
@SingletonBinding(eager=true)
public Class<? extends IFooService> bindIFooService() {
  return MyFooServiceImpl.class;
}
```

One more way of specify a binding is currently supported: If you need to control the way, the instance is created, you can declare a type to object mapping:

```
public IFooService bindIFooService() {
  return new MyFooServiceImpl();
}
```

Note that, although this is a convenient and simple way, you have of course also the full power of Guice, i.e. you can override the Guice method `void configure(Binder)` and use the afore mentioned fluent API to do whatever you want.

Chapter 5. Runtime Concepts

TMF Xtext itself and every language infrastructure developed with Xtext is configured and wired-up using dependency injection. Xtext may be used in different environments which introduce different constraints. Especially important is the difference between OSGi managed containers and plain vanilla Java programs. To honor these differences Xtext uses the concept of `ISetup`-implementations in normal mode and uses Eclipse's extension mechanism when it should be configured in an OSGi environment.

5.1. Runtime setup (ISetup)

For each language there is an implementation of `ISetup` generated. It implements a method called `doSetup()`, which can be called to do the initialization of the language infrastructure. This class is intended to be used for runtime and unit testing, only.

The setup method returns an `Injector`, which can further be used to obtain a parser, etc. It also registers the ResourceFactory and generated EPackages at the respective global registries provided by EMF. So basically you can just run the setup and start using EMF API to load and store models of your language.

5.2. Setup within Eclipse-Equinox (OSGi)

Within Eclipse we have a generated `Activator`, which creates a Guice injector using the modules. In addition an `IExecutableExtensionFactory` is generated for each language, which is used to create `ExecutableExtensions`. This means that everything which is created via extension points is managed by Guice as well, i.e. you can declare dependencies and get them injected upon creation.

The only thing you have to do in order to use this factory is to prefix the class with the factory `<MyLanguageName>ExecutableExtensionFactory` name followed by a colon.

```
<extension
  point="org.eclipse.ui.editors">
  <editor
    class="<MyLanguageName>ExecutableExtensionFactory:
      org.eclipse.xtext.ui.core.editor.XtextEditor"
    contributorClass=
      "org.eclipse.ui.editors.text.TextEditorActionContributor"
    default="true"
    extensions="ecoredsl"
    id="org.eclipse.xtext.example.EcoreDsl"
    name="EcoreDsl Editor">
  </editor>
</extension>
```

5.3. Logging

Xtext uses Apache's log4j for logging. It is configured using a so called log4j.properties, which is looked up in the root of the Java classpath. If you want to change or provide configuration at runtime (i.e. non-OSGI), all you have to do is putting such a log4j.properties in place and make sure that it is not overridden by other log4j.properties in previous class path entries.

In OSGi you provide configuration by creating a fragment for org.apache.log4j. In this case you need to make sure that there's no second fragment contributing a log4j.properties file.

5.4. Validation

Static analysis or validation is one of the most interesting aspects when developing a programming language. The users of your languages will be grateful if they get informative feedback as they type. In Xtext there are basically three different kinds of validation.

5.4.1. Syntactical Validation

The syntactical correctness of any textual input is validated automatically by the parser. The error messages are generated by the underlying parser technology and cannot be customized using a general hook. Any syntax errors can be retrieved from the Resource using the common EMF API:

- `org.eclipse.emf.ecore.resource.Resource.getErrors()`
- `org.eclipse.emf.ecore.resource.Resource.getWarnings()`

5.4.2. Crosslink Validation

Any broken crosslinks can be checked generically. As crosslink resolution is done lazily (see linking), any broken links are resolved lazily as well. If you want to validate whether all links are valid, you will have to navigate through the model so that all installed EMF proxies get resolved. This is done automatically in the editor.

Any unresolvable crosslinks will be reported and can be obtained through:

- `org.eclipse.emf.ecore.resource.Resource.getErrors()`
- `org.eclipse.emf.ecore.resource.Resource.getWarnings()`

5.4.3. Custom Validation

In addition to the afore mentioned kinds of validation, which are more or less done automatically, you can specify additional constraints specific for your Ecore model. We leverage existing EMF API (mainly `EValidator`) and have put some convenience stuff on top. Basically all you need to do is to make sure that an `EValidator` is registered for your `EPackage`. The registry for `EValidators` (`EValidator.Registry.INSTANCE`) can only be filled programmatically. That means contrary to the EPackage and `Resource.Factory` registries there is no Equinox extension point to populate the validator registry.

For Xtext we provide a generator fragment for the convenient Java-based `EValidator` API. Just add the following fragment to your generator configuration and you are good to go:

```
<fragment class=
  "org.eclipse.xtext.generator.validation.JavaValidatorFragment"/>
```

The generator will provide you with two Java classes. An abstract class generated to `src-gen/` which extends the library class `AbstractDeclarativeValidator`. This one just registers the EPackages for which this validator introduces constraints. The other class is a subclass of that abstract class and is generated to the `src/` folder in order to be edited by you. That’s where you put the constraints in.

The purpose of the `AbstractDeclarativeValidator` is to allow you to write constraints in a declarative way – as the class name already suggests. That is instead of writing exhaustive if-else constructs or extending the generated EMF switch you just have to add the `@Check` annotation to any method and it will be invoked automatically when validation takes place. Moreover you can state for what type the respective constraint method is, just by declaring a typed parameter. This also lets you avoid any type casts. In addition to the reflective invocation of validation methods the `AbstractDeclarativeValidator` provides a couple of convenient assertions.

All in all this is very similar to how JUnit works. Here is an example:

```

public class DomainmodelJavaValidator
  extends AbstractDomainmodelJavaValidator {

  @Check
  public void checkTypeNameStartsWithCapital(Type type) {
    if (!Character.isUpperCase(type.getName().charAt(0)))
      warning("Name should start with a capital",
        DomainmodelPackage.TYPE__NAME);
  }
}
```

5.4.4. Quickfixes

For validations written using the `AbstractDeclarativeValidator` it is possible to provide corresponding quickfixes in the editor. To be able to implement a quickfix for a given diagnostic (a warning or error) the underlying *cause* of the diagnostic must be known (i.e. what actual problem does the diagnostic represent?), otherwise the fix doesn't know what needs to be done. As we don't want to deduce this from the diagnostic's error message we associate a problem specific *code* with the diagnostic.

In the following example (from DomainmodelJavaValidator) the diagnostic's *code* is given by the last argument to the `warning()` method and it is a reference to the static `int` field `INVALID_TYPE_NAME` in the validator class.

```
    warning("Name should start with a capital",
      DomainmodelPackage.TYPE__NAME, INVALID_TYPE_NAME);
```

Now that the validation has a unique code identifying the problem we can register quickfixes against it. We start by adding the `org.eclipse.xtext.ui.generator.quickfix.QuickfixProviderFragment` to our workflow and after regenerating the code we should find an empty class `MyDslQuickfixProvider` in our DSL's UI project.

Continuing with the `INVALID_TYPE_NAME` problem from the Domainmodel example we add a method with which the problem can be fixed (see also DomainmodelQuickfixProvider):

```
public class DomainmodelQuickfixProvider extends AbstractDeclarativeQuickfixProvider {

  @Fix(code = DomainmodelJavaValidator.INVALID_TYPE_NAME, label = "Capitalize name",
       description = "Capitalize name of type")
  public void fixName(Type type, IMarker marker) {
    type.setName(type.getName().toUpperCase());
  }
}
```

By using the correct signature (see below) and annotating the method with the `Fix` annotation referencing the code we specified in the validator, Xtext knows that this method implements a fix for the problem. This also allows us to annotate multiple methods as fixes for the same problem.

The Fix annotation accepts the following arguments:

- **code** - the code of the validation this fix is applicable to
- **label** - the label text to display for this fix in the editor
- **description** - the description to display for this fix
- **icon** - the name of the icon file (in the UI plug-in's `icons` directory) to display for this fix

In addition to the `Fix` annotation our fix method also needs to have the correct signature: Its first argument needs to be of a type which is compatible with the type of the object the validation was created for (the validation's *source*) and its second argument is of type *IMarker*.

The fix method will be invoked inside a modification transaction and the first argument will be passed as the actual object the validation was created for and the second argument will be passed as the Eclipse marker that corresponds to the validation. This makes it very easy for the fix method to modify the model as necessary. After the method returns the model as well as the Xtext editor's content will be updated accordingly. If the method fails (throws an exception) the change will not be committed.

5.4.5. Validation with the Check language

In addition to the Java-based validation code you can use the language Check (from M2T/Xpand) to implement constraint checks against your model. To do so, you have to configure the generator with the CheckFragment. Please note, that you can combine both types of validation in your project.

```
<fragment class=
  "org.eclipse.xtext.generator.validation.CheckFragment"/>
```

After regenerating your language artifacts you will find three new files “YourLanguageChecks.chk”, “YourLanguageFastChecks.chk” and “YourLanguageExpensiveChecks.chk” in the `src/` folder in the sub-package `validation`. The checks in these files will be executed when saving a file, while typing (FastChecks) or when triggering the validation explicitly (ExpensiveChecks). When using Check the example of the previous chapter could be written like this.

```
context Type#name WARNING "Name should start with a capital":
  name.toFirstUpper() == name;
```

Each check works in a specific context (here: `Type`) and can further denote a feature to which a warning or error should be attached to (here: `name`). Each check could either be a `WARNING` or an `ERROR` with a given string to explain the situation. The essential part of each check is an invariant that must hold true for the given context. If it fails the check will produce an issue with the provided explanation.
Please read more about the Check language as well as the underlying expression language in Xpand’s reference documentation which is shipped as Eclipse help.

5.4.6. Test Validators

If you have implemented your validators by extending AbstractDeclarativeValidator, there are helper classes which may assist you when testing your validators.

Testing validators typicallally works as follows:

1. The test creates some models which intentionally violate some constraints.
2. The test runs some choosen @Check-methods from the validator.
3. The test asserts whether the @Check-methods have raised the expected warnings and errors.

To create models, you can either use EMF’s `ResourceSet` to load models from your hard disk or you can utilize the `<MyLanguage>Factory` (which EMF generates for each `EPackage`) to construct the needed model elements manually. While the fist option has the advantages that you can edit your models in your textual concrete syntax, the second option has the advantage that you can create partial models.

To run the @Check-methods and ensure they raise the intended errors and warnings, you can utilize ValidatorTester as shown by the following example:

Validator:

```
public class MyLanguageValidator extends AbstractDeclarativeValidator {
  @Check
  public void checkFooElement(FooElement element) {
    if(element.getBarAttribute().contains("foo"))
      error("Only Foos allowed", element, MyLanguagePackage.FOO_ELEMENT__BAR_ATTRIBUTE, 10
  }
}
```

JUnit-Test:

```
public class MyLanguageValidatorTest extends TestCase {

  private ValidatorTester<MyLanguageValidator> tester;

  @Override
  public void setUp() {
    MyLanguageValidator val = new MyLanguageValidator();
    new MyLanguageStandaloneSetup().createInjectorAndDoEMFRegistration().injectMembers(val
    tester = new ValidatorTester<TestingValidator>(val);
  }

  public void testError() {
    FooElement model = MyLanguageFactory.eINSTANCE.createFooElement()
    model.setBarAttribute("barbarbarbarfoo");

    tester.validator().checkFooElement(model);
    tester.diagnose().assertError(101);
  }

  public void testError2() {
    FooElement model = MyLanguageFactory.eINSTANCE.createFooElement()
    model.setBarAttribute("barbarbarbarfoo");

    tester.validate(model).assertError(101);
  }
}
```

This example uses JUnit 3, but since the involved classes from Xtext have no dependency on JUnit whatsoever, JUnit 4 and other testing frameworks will work as well. JUnit runs the `setUp()`-method before each testcase and thereby helps to create some common state. In this example, the validator (`MyLanguageValidator`) is instantiated manually and initialized via Google Guice's dependency injection. Then the `ValidatorTester` is created. It acts as a wrapper for the validator, ensures that the validator has a valid state and provides convenient access to the validator itself (`tester.validator()`) as well as to the utility classes which assert diagnostics created by the validator (`tester.diagnose()`). Please be aware that you have to call `validator()` before you can call `diagnose()`. However, you can call `validator()` multiple times in a row.

While `validator()` allows to call the validator's @Check-methods directly, `validate(model)` leaves it to the framework to call the applicable @Check-methods. However, to avoid side-effects between tests, it is recommended to call the @Check-methods directly.

`diagnose()` and `validate(model)` return an object of type @{org.eclipse.xtext/src/org/eclipse/xtext/validation/AssertableDiagnostics} which provides several `assert`-methods to verify whether the expected diagnostics are present:

- `assertError(int code)`: There must be one diagnostic with severity ERROR and the supplied error code.
- `assertErrorContains(String messageFragment)`: There must be one diagnostic with severity ERROR and its message must contain `messageFragment`.
- `assertError(int code, String messageFragment)`: Verifies severity, error code and messageFragment.
- `assertWarning(...)`: This method is available for the same combination of parameters as `assertError()`.
- `assertOK()`: Expects that no diagnostics (errors, warnings etc.) have been raised.
- `assertDiagnostics(int severity, int code, String messageFragment)`: Verifies severity, error code and messageFragment.
- `assertAll(DiagnosticPredicate... predicates)`: Allows to describe multiple diagnostics at the same time and verifies that all of them are present. Class @{org.eclipse.xtext/src/org/eclipse/xtext/validation/AssertableDiagnostics} contains static `error()` and `warning()`-methods which help to create the needed `DiagnosticPredicate`. Example: `assertAll(error(123), warning("some part of the message"))`.

- `assertAny(DiagnosticPredicate predicate)`: Asserts that a diagnostic exists which matches the predicate.

5.5. Linking

The linking feature allows for specification of cross references within an Xtext grammar. The following things are needed for the linking:

1. declaration of a crosslink in the grammar (at least in the Ecore model)
2. specification of linking semantics

5.5.1. Declaration of crosslinks

In the grammar a cross reference is specified using square brackets.

```
CrossReference :
  '[' ReferencedEClass ('|' terminal=AbstractTerminal)? ']'
;
```

Example:

```
ReferringType :
  'ref' referencedObject=[Entity|(ID|STRING)]
;
```

The Ecore model inference would create an `EClass` ‘ReferringType’ with an `EReference` ‘referencedObject’ of type ‘Entity’ (containment=false). The referenced object would be identified either by an ID or a STRING and the surrounding information in the current context (see scoping).

Example: While parsing a given input string, say

```
ref Entity01
```

Xtext produces an instance of ‘ReferringType’. After this parsing step it enters the linking phase and tries to find an instance of '‘Entity’' using the parsed text ‘Entity01’. The input

```
ref "EntityWithƒ÷‹"
```

would work analogously. This is not an ID (umlauts are not allowed), but a STRING (as it is apparent from the quotation marks).

5.5.2. Specification of linking semantics

The `ILinker` implementation that is used by default is the LazyLinker. It installs `EObject` proxies for all crosslinks, which are then resolved on demand. The actual cross reference resolution is done by `LazyLinkingResource.getEObject(String)` and delegates to an implementation of the `ILinkingService`. Although the default linking behavior is appropriate in many cases there might be scenarios where this is not sufficient. For each grammar a custom linking service can be implemented and configured. It fulfills the following interface:

```
public interface ILinkingService {

  /**
   * Returns all {@link EObject}s referenced by the given link text in the
   * given context. But does not set the references or modifies the passed
   * information somehow
   */
  List<EObject> getLinkedObjects(
    EObject context, EReference reference,
    AbstractNode node) throws IllegalNodeException;

  /**
   * Returns the textual representation of a given object as it would be
   * serialized in the given context.
   * @return the text representation.
   */
  String getLinkText(EObject object,
    EReference reference, EObject context);
}
```

The method `getLinkedObjects` is directly related to this topic whereas `getLinkText` addresses complementary functionality: it is used for serialization.

A simple implementation of the linking service (the DefaultLinkingService) is shipped with Xtext and used for any grammar per default. It uses the default implementation of `IScopeProvider` to compute the linking candidates.

5.5.3. Default linking semantics

The default implementation for all languages looks within the current file for an EObject of the respective type. In the example above this would be an “Entity” which by convention has a `name` attribute set to ‘Entity01’.

Given the grammar :

```
Model :
  (stuff+=(Ref|Entity))*
;

Ref :
  'ref' referencedObject=[Entity|ID] ';'
;

Entity :
  'entity' name=ID ';'
;
```

In the following model:

```
ref Entity01;
entity Entity01;
```

the `ref` would be linked to the declared entity (`entity Entity01;`). Nearly any aspect is configurable, especially the name of the identifying attribute may be overridden for a particular type.

5.5.3.1. Default Imports

There is a default implementation for inter-resource references, which as well uses convention. Each string in a model which is assigned to an EAttribute with the name `importURI`, will be interpreted as an URI and used to be loaded using the `ResourceSet` of the current resource.

For example, given the following grammar :

```
Model :
  (imports+=Import)*
  (stuff+=(Ref|Entity))*
;

Import :
  'import' importURI=STRING ';'
;

Ref :
  'ref' referencedObject=[Entity|ID] ';'
;

Entity :
  'entity' name=ID ';'
;
```

It would be possible to write three files in that language where the first references the other two, like this:

```
//file model.dsl
import "model1.dsl";
import "model2.dsl";

ref Foo;
entity Bar;
```

```
//file model1.dsl
entity Stuff;
```

```
//file model2.dsl
entity Foo;
```

The linking candidates for the reference `Foo` will be `Bar`, `Stuff` and `Foo` in that order. They will be computed by the ScopeProvider.

5.6. Scoping

An `IScopeProvider` is responsible for providing an `IScope` for a given context `EObject` and `EReference`. The returned `IScope` should contain all target candidates for the given object and cross reference.

```
public interface IScopeProvider {

  /**
   * Returns a scope for the given context. The scope provides access to
   * the compatible visible EObjects for a given reference.
   *
   * @param context the element from which a scope shall be computed
   * @param reference the reference to be used to filter the elements.
   * @return {@link IScope} representing the inner most {@link IScope} for
   *         the passed context and reference.
   */
  public IScope getScope(EObject context, EReference reference);

}
```

A single `IScope` represents an element of a linked list of scopes. That means that a scope can be nested within an outer scope. For instance Java has multiple kinds of scopes (object scope, type scope, etc.).

For Java one would create the scope hierarchy as commented in the following example:

```
// file contents scope
import static my.Constants.STATIC;

public class ScopeExample { // class body scope
  private Object field = STATIC;

  private void method(String param) { // method body scope
    String localVar = "bar";
    innerBlock: { // block scope
      String innerScopeVar = "foo";
      Object field = innerScopeVar;
      // the scope hierarchy at this point would look like this:
      //  blockScope{field,innerScopeVar}->
      //  methodScope{localVar, param}->
      //  classScope{field}-> ('field' is overlayed)
      //  fileScope{STATIC}->
      //  classpathScope{'all qualified names of accessible static fields'} ->
      //  NULLSCOPE{}
      //
    }
    field.add(localVar);
  }
}
```

In fact the class path scope should also reflect the order of class path entries. For instance:

```
classpathScope{stuff from bin/}
-> classpathScope{stuff from foo.jar/}
-> ...
-> classpathScope{stuff from JRE System Library}
-> NULLSCOPE{}
```

Please find the motivation behind this and some additional details in this blog post .

The default implementation would produce this hierarchy of scopes for the model from the last example in the previous chapter:

```
//file model.dsl
import "model1.dsl";
import "model2.dsl";

ref Foo;
entity Bar;
```

```
//file model1.dsl
entity Stuff;
```

```
//file model2.dsl
entity Foo;
```

```
Scope (model.dsl) {
  parent : Scope (model1.dsl) {
    parent : Scope (model2.dsl) {}
  }
}
```

When enumerating the scope's content, the first, most specialized scope would return `Bar`, its parent would provide `Stuff` and the outermost scope adds `Foo`. The linker will iterate the scope in that order and abort when it finds a matching `ScopedElement`.

5.6.1. DeclarativeScopeProvider

As always there is an implementation that allows to specify scoping in a declarative way. It looks up methods which have either of the following two signatures:

```
IScope scope_<RefDeclaringEClass>_<Reference>(<ContextType> ctx, EReference ref)
```

```
IScope scope_<TypeToReturn>(<ContextType> ctx, EReference ref)
```

The former is used when evaluating the scope for a specific cross reference and here `<ContextReference>` corresponds to the name of this reference (prefixed with the name of the reference's declaring type and separated by an underscore). The `ref` parameter represents this cross reference.

The latter method signature is used when computing the scope for a given element type and is applicable to all cross references of that type. Here `<TypeToReturn>` is the name of that type which also corresponds to the `type` parameter.

So if you for example have a state machine with a *Transition* object owned by its source *State* and you want to compute all reachable states (i.e. potential target states), the corresponding method could be declared as follows (assuming the cross reference is declared by the *Transition* type and is called *target*):

```
IScope scope_Transition_target(Transition this, EReference ref)
```

If such a method does not exist, the implementation will try to find one for the context object's container. Thus in the example this would match a method with the same name but *State* as the type of the first parameter. It will keep on walking the containment hierarchy until a matching method is found. This container delegation allows to reuse the same scope definition for elements in different places of the containment hierarchy. Also it may make the method easier to implement as the elements comprising the scope are quite often owned or referenced by a container of the context object. In the example the *State* objects could for instance be owned by a containing *StateMachine* object.

If no method specific to the cross reference in question was found for any of the objects in the containment hierarchy, the implementation will start looking for methods matching the other signature (with the *EClass* parameter). Again it will first attempt to match the context object. Thus in the example the signature first matched would be:

```
IScope scope_State(Transition this, EReference ref)
```

If no such method exists, the implementation will again try to find a method matching the context object's container objects. In the case of the state machine example you might want to declare the scope with available states at the state machine level:

```
IScope scope_State(StateMachine this, EReference ref)
```

This scope can now be used for any cross references of type *State* for context objects owned by the state machine.

There are currently two different scope provider implmentations available which support these semantics:

1. AbstractDeclarativeScopeProvider
2. AbstractDeclarativeQualifiedNameScopeProvider

5.6.2. QualifiedNameScopeProvider

The qualified name scoping is based on qualified names and name spaces. It adds name space support to your language, which is comparable and similar to the one in Scala and C#. Scala and C# both allow to have multiple nested packages within one file and you can put imports per namespace, so that imported names are only visible within that namespace. See the domain model example its scope provider extends AbstractDeclarativeQualifiedNameScopeProvider.

5.6.2.1. IQualifiedNameProvider

The QualifiedNameScopeProvider makes use of the so called IQualifiedNameProvider service. It computes qualified names for EObjects. The default implementation (DefaultDeclarativeQualifiedNameProvider) uses a simple name look up and concats the result to the qualified name of its parent object. See its JavaDoc and the code for more details.

5.6.2.2. Importing name spaces

The QualifiedNameScopeProvider looks up EAttributes with name 'importNamespace' and interprets such as import statements. By default qualified names with or without a wildcard at the end are supported. For an import of a qualified name the simple name is made available as we know from e.g. Java, where

```
import java.util.Set;
```

makes it possible to refer to ‘java.util.Set’ by its simple name ‘Set’. Contrary to Java the import is not active for the whole file but only for the namespace it is declared in and its child namespaces. That is why you can write the following in the example DSL:

```
package foo {
  import bar.Foo
  entity Bar extends Foo {
  }
}

package bar {
  entity Foo {}
}
```

Of course the declared elements within a package are as well referable by their simple name:

```
package bar {
  entity Bar extends Foo {}
  entity Foo {}
}
```

Of course the following would as well be ok:

```
package bar {
  entity Bar extends bar.Foo {}
  entity Foo {}
}
```

As the name suggests it uses the EMF index to find any EObjects which are not located in the current resource. The IndexBasedScopeProvider supports nested namespaces (similar to C# and Scala) and is used in the Domainmodel example (project org.eclipse.xtext.example.domainmodel). There is support for declarative overwriting of the default semantics if you subclass AbstractDeclarativeIndexBasedScopeProvider.

See the JavaDocs and this blog post for details.

5.7. Value Converter

Value converters are registered to convert the parsed text into a certain data type instance and vice versa. The primary hook is called IValueConverterService and the concrete implementation can be registered via the runtime Guice module. To do so override the corresponding binding in your runtime module like shown in this example:

```
@Override
public Class<? extends IValueConverterService> bindIValueConverterService() {
  return MySpecialValueConverterService.class;
}
```

5.7.1. Annotation based value converters

The most simple way to register additional value converters is to make use of AbstractDeclarativeValueConverterService, which allows to declaratively register an `IValueConverter` via an annotated method.

The implementation for the default token grammar looks like

```
public class DefaultTerminalConverters
    extends AbstractDeclarativeValueConverterService {

  (..)

  @ValueConverter(rule = "ID")
  public IValueConverter<String> ID() {
    return new AbstractNullSafeConverter<String>() {
      @Override
      protected String internalToValue(String string, AbstractNode node) {
        return string.startsWith("^") ? string.substring(1) : string;
      }

      @Override
      protected String internalToString(String value) {
        if (GrammarUtil.getAllKeywords(getGrammar()).contains(value)) {
          return "^" + value;
        }
        return value;
      }
    };
  }
  ...  some other value converter
}
```

If you use the common terminals grammar `org.eclipse.xtext.common.Terminals` you should subclass `DefaultTerminalConverters` and override or add additional value converters by adding the respective methods.

Imagine, you would want to add a rule BIG_DECIMAL creating BigDecimals, it would look like this one:

```
@ValueConverter(rule = "BIG_DECIMAL")
public IValueConverter<BigDecimal> BIG_DECIMAL() {
  return new AbstractToStringConverter<BigDecimal>() {
    @Override
    protected BigDecimal internalToValue(String string, AbstractNode node) {
      return BigDecimal.valueOf(string);
    }
  };
}
```

5.8. Serialization

Serialization is the process of transforming an EMF model into its textual representation. Thereby, serialization complements parsing and lexing.

In Xtext, the process of serialization is split into three steps:

1. Matching the model elements with the grammar rules and creating a stream of tokens. This is done by the parse tree constructor.
2. Mixing existing hidden tokens (whitespace, comments, etc.) into the token stream. This is done by the hidden token merger.
3. Adding needed whitespace or replacing all whitespace using a formatter.

Serialization is invoked when calling XtextResource `.save(...)`. Furthermore, SerializerUtil provides resource-independent support for serialization.

5.8.1. The Contract

The contract of serialization says that a model that is serialized to its textual representation and then loaded (parsed) again should yield a loaded model that equals the original model. Please be aware that this does *not* imply, that loading a textual representation and serializing it back produces identical textual representations. For example, this is the case when a default value is used in a textual representation and the assignment is optional. Another example is:

```
MyRule:
  (xval+=ID | yval+=INT)*;
```

`MyRule` in this example reads `ID`- and `INT`-elements which may occur in an arbitrary order in the textual representation. However, when serializing the model all `ID`-elements will be written first and then all `INT`-elements. If the order is important it can be preserved by storing all elements in the same list – which may require wrapping the `ID`- and `INT`-elements into objects.

5.8.2. Parse Tree Constructor

The parse tree constructor usually does not need to be customized since it is automatically derived from the Xtext Grammar. However, it can be a good idea to look into it to understand its error messages and its runtime performance.

For serialization to succeed, the parse tree constructor must be able to *consume* every element of the to-be-serialized EMF model. To *consume* means, in this context, to write the element to the textual representation of the model. This can turn out to be a not-so-easy to fulfill requirement, since a grammar usually introduces implicit constraints to the EMF model. Example:

```
MyRule:
  (sval+=ID ival+=INT)*;
```

This example introduces the constraint `sval.size() == ival.size()`. Models which violate this constraint are sort of valid EMF models, but they can not be serialized. To check whether a model complies with all constraints introduced by the grammar, the only way is currently to invoke the parse tree constructor. If this changes at some day, there will be news in bugzilla 239565.

For the parse tree constructor, this can lead to the scenarios, where

- a model element can not be consumed. This can have the following reasons/solutions:
  - The model element should not be stored in the model.
  - The grammar needs an assignment which would consume the model element.
  - The transient value service could be used to indicate that this model element should not be consumed.
- an assignment in the grammar has no corresponding model element. The parse tree constructor considers a model element not to be present if it is *unset* or equals its default value. However, the parse tree constructor may serialize default values if this is required by a grammar constraint to be able to serialize another model element. The following solution may help to solve such a scenario:
  - A model element should be added to the model.
  - The assignment in the grammar should be made optional.

To understand error messages and performance issues of the parse tree constructor it is important to know that it implements a backtracking algorithm. This basically means that the grammar is used to specify the structure of a tree in which one path (from the root node to a leaf node) is a valid serialization of a specific model. The parse tree constructor's task is to find this path – with the condition, that all model elements are consumed while walking this path. The parse tree constructor's strategy is to take the most promising branch first (the one that would consume the most model elements). If the branch leads to a dead end (for example, if a model element needs to be consumed that is not present in the model), the parse tree constructor goes back the path until a different branch can be taken. This behavior has two consequences:

- In case of an error, the parse tree constructor has found only dead ends but no leaf. It cannot tell which dead end is actually erroneous. Therefore, the error message lists dead ends of the longest paths, a fragment of their serialization and the reason why the path could not be continued at this point. The developer has to judge on his own which reason is the actual error.
- For reasons of performance, it is critical that the parse tree constructor takes the most promising branch first and detects wrong branches early. One way to achieve this is to avoid having many rules which return the same type and which are called from within the same alternative in the grammar.

5.8.3. Transient Values

Transient values are values or model elements which are not persisted (written to the textual representation in the serialization phase). If a model contains model elements which can not be serialized with the current grammar, it is critical to mark them transient using the ITransientValueService, or serialization will fail. The default implementation marks all model elements transient, that are *unset* or equal to their default value.

5.8.4. Unassigned Text

Unassigned text can be necessary due to data type rule calls or terminal rule calls which do not reside within an assignment. Example:

```
PluralRule:
  'contents:' count=INT Plural;

terminal Plural:
  'item' | 'items';
```

Valid models for this example are `contents 1 item` or `contents 5 items`. However, it is not stored in the semantic model whether the keyword `item` or `items` has been parsed. This is due to the fact that the rule call `Plural` is unassigned. However, the parse tree constructor needs to decide which value to write during serialization. This decision can be be made by implementing the IUnassignedTextSerializer.

5.8.5. Cross Reference Serializer

The cross reference serializer specifies which values are to be written to the textual representation for cross references. This behavior can be customized by implementing ICrossReferenceSerializer. The default implementation delegates to ILinkingService, which may be the better place for customization.

5.8.6. Hidden Token Merger

After the parse tree constructor has done its job to create a stream of tokens which are to be written to the textual representation, the hidden token merger (IHiddenTokenMerger) mixes existing hidden tokens into this token stream. The default implementation uses the hidden tokens (whitespace, line breaks, comments) from the node model. The IHiddenTokenMerger is the factory for a token stream which is fed by the parse tree constructor and which writes to another token stream.

5.8.7. Token Stream

The parse tree constructor, the hidden token merger and the formatter use an ITokenStream for their output, and the latter two for their input as well. This makes them chainable. Token streams can be converted to a `String` using the TokenStringBuffer and to an `OutputStream` using the TokenOutputStream. Maybe there will be an implementation to reconstruct a node model as well at some point in the future. While providing fast output due to the stream pattern, token streams allow easy manipulation of the stream, such as mixing in whitespace or manipulating them.

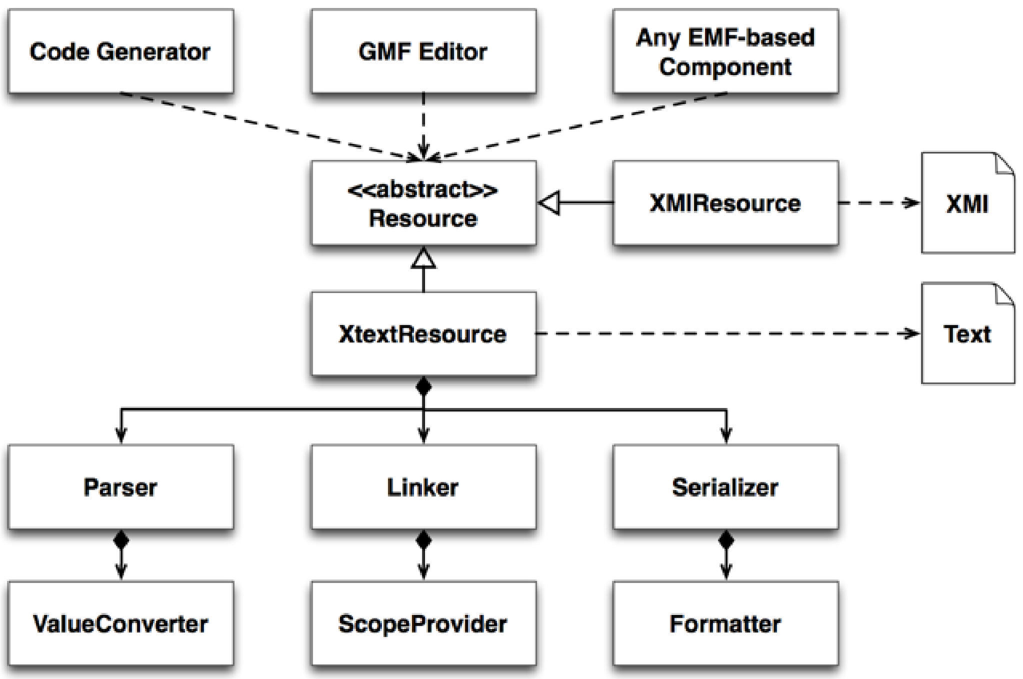
```
public interface ITokenStream {
  public void close() throws IOException;
  public void writeHidden(
    EObject grammarElement, String value) throws IOException;
  public void writeSemantic(
    EObject grammarElement, String value) throws IOException;
}
```

5.9. Integration with EMF

Xtext relies heavily on EMF internally, but it can also be used as the serialization back-end of other EMF-based tools.

5.9.1. XtextResource Implementation

Xtext provides an implementation of EMF's resource, the XtextResource. This does not only encapsulate the parser that converts text to an EMF model but also the serializer working the opposite direction. That way, an Xtext model just looks like any other Ecore-based model from the outside, making it amenable for the use by other EMF based tools. In fact, the Xpand templates in the generator plug-in created by the Xtext wizard do not make any assumption on the fact that the model is described in Xtext, and they would work fine with any model based on the same Ecore model of the language. So in the ideal case, you can switch the serialization format of your models to your self-defined DSL by just replacing the resource implementation used by your other modeling tools.



The generator fragment ResourceFactoryFragment registers a factory for the XtextResource to EMF's resource factory registry, such that all tools using the default mechanism to resolve a resource implementation will automatically get that resource implementation.

Using a self-defined textual syntax as the primary storage format has a number of advantages over the default XMI serialization, e.g.

- You can use well-known and easy-to-use tools and techniques for manipulation, such as text editors, regular expressions, or stream editors.
- You can use the same tools for version control as you use for source code. Merging and diffing is performed in a syntax the developer is familiar with.
- It is impossible to break the model such that it cannot be reopened in the editor again.
- Models can be fixed using the same tools, even if they have become incompatible with a new version of the Ecore model.

Xtext targets easy to use and naturally feeling languages. It focuses on the lexical aspects of a language a bit more than on the semantic ones. As a consequence, a referenced Ecore model can contain more concepts than are actually covered by the Xtext grammar. As a result, not everything that is possibly expressed in the EMF model can be serialized back into a textual representation with regards to the grammar. So if you want to use Xtext to serialize your models as described above, it is good to have a couple of things in mind:

- Prefer optional rule calls (cardinality ? or *) to mandatory ones (cardinality + or default), such that missing references will not obstruct serialization.
- You should never use an Xtext-Editor on the same model instance as a self-synchronizing other editor, e.g. a canonical GMF editor. The Xtext parser replaces re-parsed subtrees of the AST rather than modifying it, so elements will become stale. Additional information, such as graphical properties in the case of GMF, attached to these elements are likely to be lost. As the Xtext editor continuously re-parses the model on changes, this will happen rather often. It is safer to synchronize editors on file changes.
- Implement an IFragmentProvider to make the XtextResource return stable fragments for its contained elements, e.g. based on composite names rather than order of appearance.

5.9.2. Fragment Provider (referencing Xtext models from other EMF artifacts)

Although inter-Xtext linking is not done by URIs, you may want to be able to reference your `EObject` from non-Xtext models. In those cases URIs are used, which are made up of a part identifying the resource. Each `EObject` contained in a resource can be identified by a so called *fragment*.

A fragment is a part of an EMF URI and needs to be unique per resource.

The generic XMI resource shipped with EMF provides a generic path-like computation of fragments. With an XMI or other binary-like serialization it is also common and possible to use UUIDs.

However with a textual concrete syntax we want to be able to compute fragments out of the given information. We don't want to force people to use UUIDs (i.e. synthetic identifiers) or relative generic paths (very fragile), in order to refer to `EObjects`.

Therefore one can contribute a so called `IFragmentProvider` per language.

```
public interface IFragmentProvider extends ILanguageService {

  /**
   * Computes the local ID of the given object.
   * @param obj
   *            The EObject to compute the fragment for
   * @return the fragment, which can be an arbitrary string but must be
   *         unique within a resource. Return null to use default
   *         implementation
   */
  String getFragment(EObject obj);

  /**
   * Locates an EObject in a resource by its fragment.
   * @return the EObject
   */
  EObject getEObject(Resource resource, String fragment);
}
```

Note that the currently available default fragment provider does nothing (i.e. falls back to the default behavior of EMF).

Chapter 6. IDE concepts

For the following part we will refer to a concrete example grammar in order to explain certain aspect of the UI more clearly. The used example grammar is as follows:

```
grammar org.eclipse.text.documentation.Sample
    with org.eclipse.xtext.common.Terminals

generate gen 'http://www.eclipse.org/xtext/documentation/Sample' as gen

Model :
  "model" intAttribute=INT (stringDescription=STRING)? "{"
     (rules += AbstractRule)*
  "}"
;

AbstractRule:
  RuleA | RuleB
;

RuleA :
   "RuleA" "(" name = ID ")" ;

RuleB return gen::CustomType:
   "RuleB" "(" ruleA = [RuleA] ")" ;
```

6.1. Label Provider

A nice part of the Eclipse tooling that comes with Xtext is the outline view. It shows the structure of your model as a tree and allows quick navigation to model elements. Thus it helps to get an overview on the current state in the editor at a glance. To make the appearance of the outline more appealing it is very easy possible to provide customization for the label and the image that is used for an element. Actually this customization will be used at various places in your IDE, for example in the window that displays completion proposals when content assist was invoked. (Customizing the structure of the outline is described in a separate chapter).

The LabelProvider is the service which is used to compute the image for model elements and – as its name suggests – the label that represents an element. What you basically have to do is to provide an implementation for two methods which read `Image getImage(Object)` and `String getText(Object)` respectively. As this tends to be cumbersome due to `instanceof` and cast orgies, Xtext ships with a reasonable and convenient default implementation.

6.1.1. DefaultLabelProvider

The default implementation of the `LabelProvider` interface utilizes the polymorphic dispatcher idiom to implement an external visitor as the requirements of the LabelProvider are kind of a best match for this pattern. It comes down to the fact that the only thing you need to do is to implement a method that matches a specific signature. It either provides a image filename or the text to be used to represent your model element. Have a look at following example to get a more detailed idea about the `DefaultLabelProvider`.

```
public class SampleLabelProvider extends DefaultLabelProvider {

  String text(RuleA rule) {
  return "Rule: " + rule.getName();
 }

 String image(RuleA rule) {
  return "ruleA.gif";
 }

 String image(RuleB rule) {
  return "ruleB.gif";
 }

}
```

The declarative implementation of the label itself is pretty straightforward. The image in turn is expected to be found in a file named `icons/<result of image-method>` in your plugin. This path is actually configurable by google guice. Have a look at the PluginImageHelper to learn about the customizing possibilities.

What is especially nice about the default implementation is the actual reason for its class name: It provides very reasonable defaults. To compute the label for a certain model element, it will at first have a look for an EAttribute that is called `name` and try to use this one. If it cannot find a feature like this, it will try to use the first feature, that can be used best as a label. At worst it will return the class name of the model element, which is kind of unlikely to happen.

More advanced usage patterns of the `DefaultLabelProvider` include a dispatching to an error handler called `String error_text(Object, Exception)` and `String error_image(Object, Exception)` respectively.

6.2. Content Assist

The Xtext generator, amongst other things, generates the following two content assist (CA) related artifacts:

- an abstract proposal provider class named `'Abstract[Language]ProposalProvider'` generated into the `src-gen` folder within the `ui` project
- a concrete descendent in the `src`-folder of the `ui` project `ProposalProvider`

First we will investigate the generated `Abstract[Language]ProposalProvider` with methods that look like this:

6.2.1. ProposalProvider

```
public void complete[TypeName]_[FeatureName](
  EObject model,
  Assignment assignment,
  ContentAssistContext context,
  ICompletionProposalAcceptor acceptor) {
  // clients may override
}

public void complete_[RuleName](
  EObject model,
  RuleCall ruleCall,
  ContentAssistContext context,
  ICompletionProposalAcceptor acceptor) {
  // clients may override
}
```

The snippet above indicates that the generated ProposalProvider class contains a `complete*`-method for each assigned feature in the grammar and for each rule. The brackets are place-holders that should give a clue about the naming scheme used to create the various entry points for clients. The generated proposal provider falls back to some default behavior for cross references. Furthermore it inherits the logic that was introduced in reused grammars.

Clients who want to customize the behavior may override the methods from the `AbstractProposalProvider` or introduce new methods with specialized parameters. The framework dispatches method calls according to the current context to the most concrete implementation, that can be found.

It is important to know, that for a given offset in a model file, many possible grammar elements exist. The framework dispatches to the method declarations for any valid element. That means, that a bunch of '‘complete.*’' methods may be called.

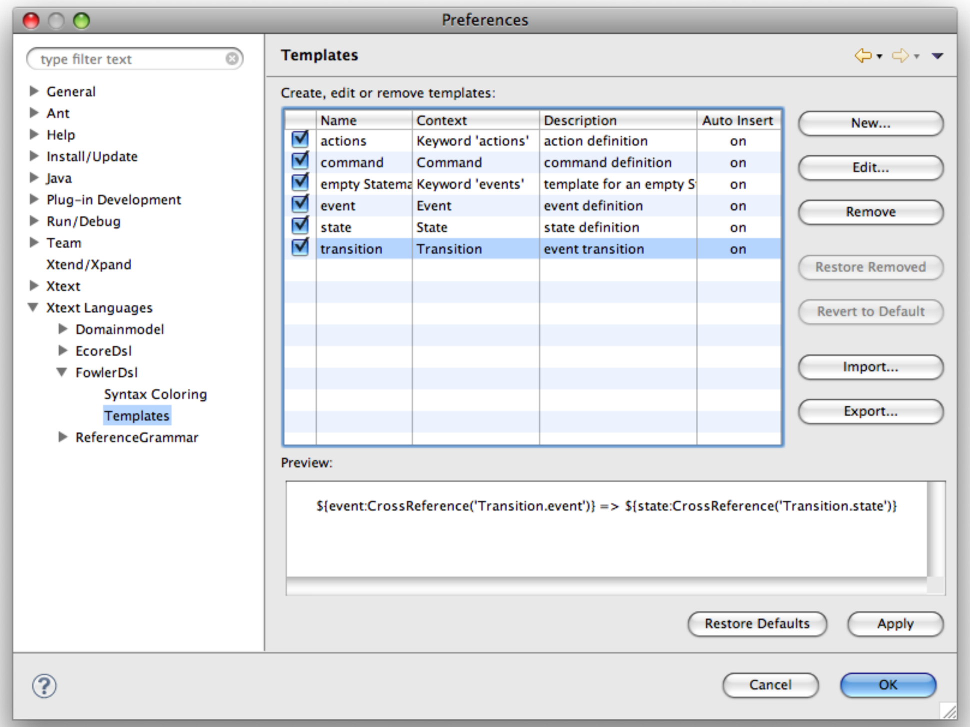
6.2.2. Sample Implementation

To provide a dummy proposal for the description of a model object, you may introduce a specialization of the generated method and implement it as follows. This will give ‘Description for model #7’ for a model with the intAttribute '7'

```
public void completeModel_StringDescription (
  Model model,
  Assignment assignment,
  ContentAssistContext context,
  ICompletionProposalAcceptor acceptor) {
  // call implementation in superclass
  super.completeModel_StringDescription(
    model,
    assignment,
    context,
    acceptor);

  // compute the plain proposal
  String proposal = "Description for model #" + model.getIntAttribute();

  // convert it to a valid STRING-terminal
  proposal = getValueConverter().toString(proposal, "STRING");

  // create the completion proposal
  // the result may be null as the createCompletionProposal(..) methods
  // check for valid prefixes
  // and terminal token conflicts
  ICompletionProposal completionProposal =
    createCompletionProposal(proposal, contentAssistContext);

  // register the proposal, the acceptor handles null-values gracefully
  acceptor.accept(completionProposal);
}
```

6.3. Template Proposals

Xtext-based editors automatically support code templates. That means that you get the corresponding preference page where users can add and change template proposals. If you want to ship a couple of default templates, you have to put a file named `templates.xml` inside the `templates` directory of the generated ui-plugin. This file contains templates in a format as described in the Eclipse online help .



Templates

Create, edit or remove templates:

	Name	Context	Description	Auto Insert
✓	actions	Keyword 'actions'	action definition	on
✓	command	Command	command definition	on
✓	empty Statema	Keyword 'events'	template for an empty S	on
✓	event	Event	event definition	on
✓	state	State	state definition	on
✓	transition	Transition	event transition	on

Preview:

```
${event:CrossReference('Transition.event')} => ${state:CrossReference('Transition.state')}
```

By default Xtext registers ContextTypes for each Rule (`.[RuleName]`) and for each keyword (`.kw_[keyword]`), as long as the keywords are valid identifiers. If you don't like these defaults you'll have to subclass XtextTemplateContextTypeRegistry and configure it via Guice.

In addition to the standard template proposal extension mechanism, Xtext ships with a predefined set of TemplateVariableResolvers to resolve special variable types inside a given template (i.e. TemplateContext). Besides the standard template variables available in `org.eclipse.jface.text.templates.GlobalTemplateVariables` like ${user}, ${date}, ${time}, ${cursor}, etc., these TemplateVariableResolver support the automatic resolving of CrossReferences (type `CrossReferences`) and Enumerations (type `Enum`) like it is explained in the following sections.

It is best practice to edit the templates in the preferences page, export them into the `templates.xml`-file and put this one into the `templates` folder of your ui-plugin. However, these templates will not be visible by default. To fix it, you have to manually edit the xml-file and insert an ID attribute for each template element.

6.3.1. CrossReference TemplateVariableResolver

Xtext comes with a specific template variable resolver `org.eclipse.jface.text.templates.TemplateVariableResolver)` called `CrossReferenceResolver`, which can be used to place cross references within a template.

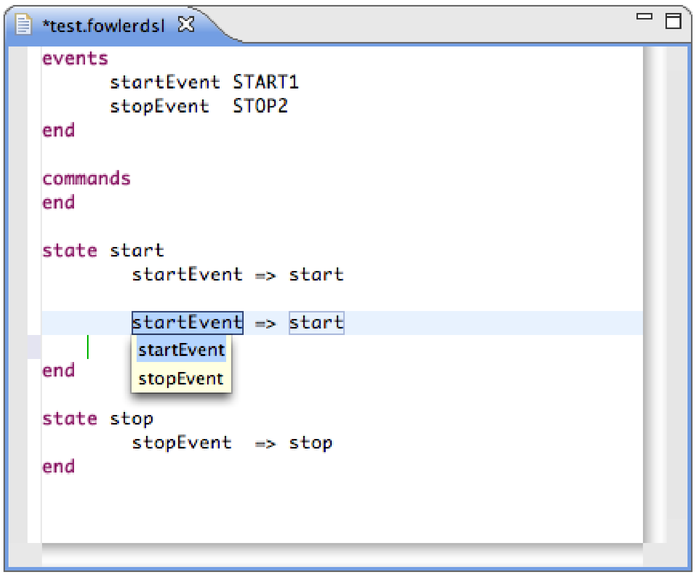
The syntax is as follows:

```
${displayText:CrossReference('MyType.myRef')}
```

For example the following template:

```
<template name="transition" description="event transition" id="transition"
    context="org.eclipse.xtext.example.FowlerDsl.Transition" enabled="true">
    ${event:CrossReference('Transition.event')} =>
     ${state:CrossReference('Transition.state')
</template>
```

yields the text `event => state` and allows selecting any events and states using a drop down.



```
*test.fowlerdsl
events
	startEvent START1
	stopEvent  STOP2
end

commands
end

state start
		startEvent => start

		startEvent => start
		startEvent
end		stopEvent

state stop
		stopEvent  => stop
end
```

6.3.2. Enumeration TemplateVariableResolver

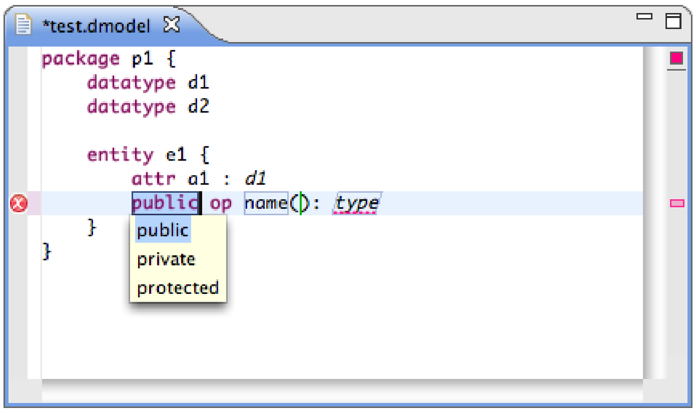
The EnumTemplateVariableResolver resolves a template variable to `EEnumLiteral` literals which are assignment-compatible to the enumeration type declared as the first parameter of the the `EnumTemplateVariable`.
The syntax is as follows:

```
${displayText:Enum('Visibility')
```

For example the following template (taken from the domainmodel example):

```
<template name="Operation" description="template for an Operation"
        id="org.eclipse.xtext.example.Domainmodel.Operation"
        context="org.eclipse.xtext.example.Domainmodel.Operation"
         enabled="true">
        ${visibility:Enum('Visibility')} op ${name}(${cursor}):
           ${type:CrossReference('Operation.type')}
</template>
```

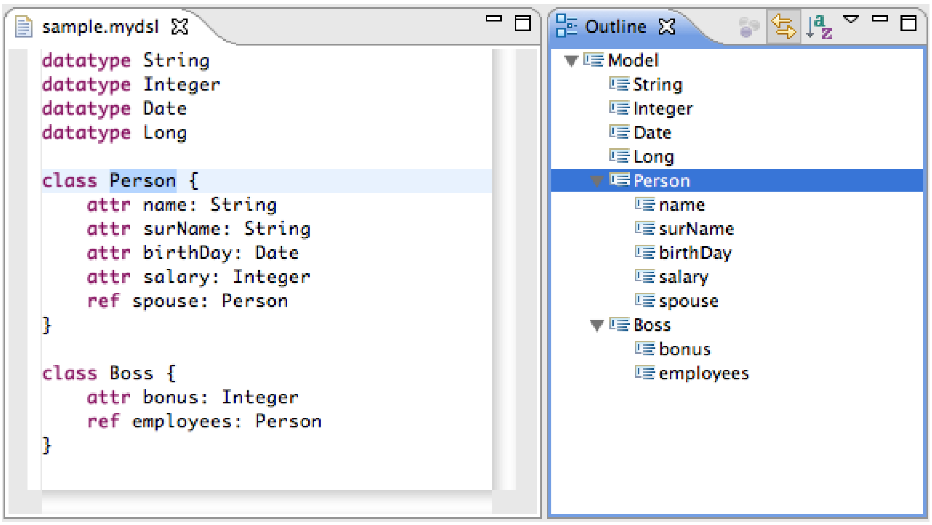
yields the text `public op name(): type` where the display text 'public' is replaced with a drop down filled with the literal values as defined in the Visibility `EENumeration`. Also, 'name' and 'type' are placeholders.



```
*test.dmodel
package p1 {
    datatype d1
    datatype d2

    entity e1 {
        attr a1 : d1
        public op name(): type
    }   public
}       private
        protected
```

6.4. Outline View

Xtext provides an outline view to help you navigate your models. By default, it provides a hierarchical view on your model and allows you to sort tree elements alphabetically. Selecting an element in the outline will highlight the corresponding element in the text editor. Users can choose to synchronize the outline with the editor selection by clicking the *Link with Editor* button.



```
datatype String
datatype Integer
datatype Date
datatype Long

class Person {
    attr name: String
    attr surName: String
    attr birthDay: Date
    attr salary: Integer
    ref spouse: Person
}

class Boss {
    attr bonus: Integer
    ref employees: Person
}
```

```
Model
    String
    Integer
    Date
    Long
    Person
        name
        surName
        birthDay
        salary
        spouse
    Boss
        bonus
        employees
```

You can customize various aspects of the outline by providing implementation for its various interfaces. The following sections show how to do this.

6.4.1. Influencing the outline structure

In its default implementation, the outline view shows the containment hierarchy of your model. This should be sufficient in most cases. If you want to adjust the structure of the outline, i.e., by omitting a certain kind of node or by introducing additional even virtual nodes, you customize the outline by implementing ISemanticModelTransformer.

The Xtext wizard creates an empty transformer class (`MyDslTransformer`) for your convenience. To transform the semantic model delivered by the Xtext parser, you need to provide transformation methods for each of the EClasses that are of interest:

```
public class MyDslTransformer extends
    AbstractDeclarativeSemanticModelTransformer {
  /**
   * This method will be called by naming convention:
   * - method name must be createNode
   * - first param: subclass of EObject
   * - second param: ContentOutlineNode
   */
  public ContentOutlineNode createNode(
      Attribute semanticNode, ContentOutlineNode parentNode) {
    ContentOutlineNode node = super.newOutlineNode(semanticNode, parentNode);
    node.setLabel("special " + node.getLabel());
    return node;
  }

  public ContentOutlineNode createNode(
      Property semanticNode, ContentOutlineNode parentNode) {
    ContentOutlineNode node = super.newOutlineNode(semanticNode, parentNode);
    node.setLabel("pimped " + node.getLabel());
    return node;
  }

  /**
   * This method will be called by naming convention:
   * - method name must be getChildren
   * - first param: subclass of EObject
   */
  public List<EObject> getChildren(Attribute attribute) {
    return attribute.eContents();
  }

  public List<EObject> getChildren(Property property) {
    return NO_CHILDREN;
  }
}
```

To make sure Xtext picks up your new outline transformer, you have to register your implementation with your UI module:

```
public class MyDslUiModule extends AbstractMyDslUiModule {

  @Override
  public Class<? extends ISemanticModelTransformer>
    bindISemanticModelTransformer() {
    return MyDslTransformer.class;
  }
  ...
}
```

6.4.2. Filtering

Often, you want to allow users to filter the contents of the outline to make it easier to concentrate on the relevant aspects of the model. To add filtering capabilities to your outline, you need to add AbstractFilterActions to the outline. Actions can be contributed by implementing and registering a DeclarativeActionBarContributor.

To register a `DeclarativeActionBarContributor`, add the following lines to your `MyDslUiModule` class:

```
public class MyDslUiModule extends AbstractMyDslUiModule {

  @Override
  public Class<? extends IActionBarContributor> bindIActionBarContributor() {
    return MyDslActionBarContributor.class;
  }
  ...
}
```

The action bar contributor will look like this:

```
public class MyDslActionBarContributor extends
    DeclarativeActionBarContributor {
  public Action addFilterParserRulesToolbarAction(
      XtextContentOutlinePage page) {
    return new FilterFooAction(page);
  }
}
```

Filter actions must extend `AbstractFilterAction` (this ensures that the action toggle state is handled correctly):

```
public class FilterFooAction extends AbstractFilterAction {

  public FilterFooAction(XtextContentOutlinePage outlinePage) {
    super("Filter Foo", outlinePage);
    setToolTipText("Show / hide foo");
    setDescription("Show / hide foo");
    setImageDescriptor(Activator.getImageDescriptor("icons/foo.gif"));
    setDisabledImageDescriptor(
     Activator.getImageDescriptor("icons/foo.gif"));
  }

  @Override
  protected String getToggleId() {
    return "FilterFooAction.isChecked";
  }

  @Override
  protected ViewerFilter createFilter() {
    return new FooOutlineFilter();
  }

}
```

The filtering itself will be performed by `FooOutlineFilter`:

```
public class FooOutlineFilter extends ViewerFilter {

  @Override
  public boolean select(
      Viewer viewer, Object parentElement, Object element) {
    if ((parentElement != null)
        && (parentElement instanceof ContentOutlineNode)) {
      ContentOutlineNode parentNode = (ContentOutlineNode) parentElement;
      EClass clazz = parentNode.getClazz();
      if (clazz.equals(MyDslPackage.Literals.ATTRIBUTE)) {
        return false;
      }
    }
    return true;
  }

}
```

6.4.3. Context menus

You might want to register context menu actions for specific elements in the outline, e.g. to allow users of your DSL to invoke a generator or to validate the selected element. As all elements in the outline are ContentOutlineNodes, you cannot easily register an Object contribution. (Besides, using the extension point `org.eclipse.ui.popupMenus` is regarded somewhat old school – you should rather use the new command and expression framework, as depicted below).

So to register context menus for specific node types of your Ecore model, you need to do the following:

- register a IContentOutlineNodeAdapterFactory which will translate ContentOutlineNodes to their underlying node type
- register a menu contribution to add a command / handler pair to the context menu for the specific node types you're interested in.

6.4.3.1. Registering an `IContentOutlineNodeAdapterFactory`

Create a subclass of DefaultContentOutlineNodeAdapterFactory - this class contains some base infrastructure to manage adapters for your node types. In your subclass, you need to override `getAdapterList()` and return an array of classes. This is the list of classes you want to add context menus to:

```
public class MyDslContentOutlineNodeAdapterFactory extends
    DefaultContentOutlineNodeAdapterFactory {
  @SuppressWarnings("unchecked")
  private static final Class[] types = { Attribute.class };

  @SuppressWarnings("unchecked")
  public Class[] getAdapterList() {
    return types;
  }
}
```

Register this class with your `MyDslUiModule` to make it available to your DSL editor:

```
public class MyDslUiModule extends AbstractMyDslUiModule {
  ...
  public Class<? extends IContentOutlineNodeAdapterFactory>
      bindIContentOutlineNodeAdapterFactory() {
    return org.example.myDSLContentOutlineNodeAdapterFactory.class;
  }
  ...
}
```

6.4.3.2. Registering a menu contribution

After you have registered the IContentOutlineNodeAdapterFactory, you can add command / handler pairs to the context menu.

First, you need to define a command – it will serve as a handle to glue together the handler and the menu contribution:

```

<extension
  point="org.eclipse.ui.commands">
  <command
    id="org.example.mydsl.ui.editor.outline.SampleOutlineCommand"
    name="Sample Command"
    description="Just a sample command">
  </command>
</extension>
```

Next, you need to define a handler which will eventually execute the code to operate on the selected node. Please pay special attention to the attribute `commandId` - it must match the `id` attribute of your command.

```
<extension
  point="org.eclipse.ui.handlers">
  <handler
    class="org.example.mydsl.ui.editor.outline.SampleOutlineNodeHandler"
    commandId="org.example.mydsl.ui.editor.outline.SampleOutlineCommand">
  </handler>
</extension>
```

Finally, define a `menuContribution` to add the command to the context menu:

```
<extension
  point="org.eclipse.ui.menus">
  <menuContribution
    locationURI="popup:org.eclipse.xtext.ui.common.outline?after=additions">
    <command
      commandId="org.example.mydsl.ui.editor.outline.SampleOutlineCommand"
      label="Sample action registered for Attributes">
      <visibleWhen checkEnabled="false">
        <iterate>
          <adapt type="org.example.mydsl.Attribute" />
        </iterate>
      </visibleWhen>
    </command>
  </menuContribution>
</extension>
```

Again, pay attention to the `commandId` attribute. The connection between your node type(s) and the menu contribution is made by the part `<adapt type="org.example.mydsl.Attribute" />`.

6.5. Hyperlinking

The Xtext editor provides hyperlinking support for any tokens corresponding to cross references in your grammar definition. This means that you can either control-click on any of these tokens or hit F3 while the cursor position is at the token in question and this will take you to the referenced model element. As you'd expect this works for references to elements in the same resource as well as for references to elements in other resources. In the latter case the referenced resource will first be opened using the corresponding editor.

6.5.1. Location Provider

When navigating a hyperlink Xtext will also select the text region corresponding to the referenced model element. Determining this text region is the responsibility of the ILocationInFileProvider. The default implementation (DefaultLocationInFileProvider) implements a best effort strategy which can be summarized as:

1. If the model element's type (i.e. EClass) declares a feature *name* then return the region of the corresponding token(s). As a fallback also check for a feature *id*.
2. If the model element's parse tree contains any top-level tokens corresponding to *ID* rule calls in the grammar then return a region spanning all those tokens.
3. As a last resort return the region corresponding to the first keyword token of the referenced model element.

6.5.1.1. Customized Location Provider

As the default strategy is a best effort it may not always result in the selection you want. If that's the case you can override the `ILocationInFileProvider` binding in the UI module as in the following example:

```
public class MyDslUiModule extends AbstractMyDslUiModule {

  @Override
  public Class<? extends ILocationInFileProvider>
    bindILocationInFileProvider() {
    return MyDslLocationInFileProvider.class;
  }
  ...
}
```

Often the default strategy only needs some guidance (e.g. selecting the text corresponding to another feature than *name*). In that case you can simply subclass `DefaultLocationInFileProvider` and override the methods `getIdentifierFeature` and / or `useKeyword` to guide the first and last steps of the strategy as described above (see XtextLocationInFileProvider for an example).

6.6. Formatting (Pretty Printing)

A formatter can be implemented via the IFormatter service. Technically speaking, a formatter is a Token Stream which inserts/removes/modifies hidden tokens (whitespace, line-breaks, comments).

The formatter is invoked during the serialization phase and when the user triggers formatting in the editor (for example, using the CTRL+SHIFT+F shortcut).

Xtext ships with two formatters:

- The OneWhitespaceFormatter simply writes one whitespace between all tokens.
- The AbstractDeclarativeFormatter allows advanced configuration using a FormattingConfig. Both are explained in the next chapter.

6.6.1. Declarative Formatter

A declarative formatter can be implemented by sub-classing {org.eclipse.xtext/src/org.eclipse.xtext.formatting.impl.AbstractDeclarativeFormatter}, as shown in the following example:

```
public class ExampleFormatter extends AbstractDeclarativeFormatter {

  @Override
  protected void configureFormatting(FormattingConfig c) {
    ExampleLanguageGrammarAccess f = getGrammarAccess();

    c.setAutoLinewrap(120);

    // Line
    c.setLinewrap(2).after(f.getLineAccess().getSemicolonKeyword_1());
    c.setNoSpace().before(f.getLineAccess().getSemicolonKeyword_1());

    // TestIndentation
    c.setIndentation(
      f.getTestIndentationAccess().getLeftCurlyBracketKeyword_1(),
        f.getTestIndentationAccess().getRightCurlyBracketKeyword_3());
    c.setLinewrap().after(
      f.getTestIndentationAccess().getLeftCurlyBracketKeyword_1());
    c.setLinewrap().after(
      f.getTestIndentationAccess().getRightCurlyBracketKeyword_3());

    // Param
    c.setNoLinewrap().around(f.getParamAccess().getColonKeyword_1());
    c.setNoSpace().around(f.getParamAccess().getColonKeyword_1());

    // comments
    c.setNoLinewrap().before(f.getSL_COMMENTRule());
  }
}
```

The formatter has to implement the method `configureFormatting(...)` which is supposed to declaratively set up a FormattingConfig.

The FormattingConfig consist general settings and a set of rules:

6.6.1.1. General FormattingConfig Settings

- `setAutoLinewrap(int)` defines the amount of characters after which a line-break should be dynamically inserted between two tokens. The rule `setNoLinewrap()` can be used to suppress this behavior locally. The default is 80.
- `setIndentationSpace(String)` defines the string which is used for a single degree of indentation. The default is two whitespace.
- `setWhitespaceRule(AbstractRule)` defines the grammar rule which is used to match whitespace. This is needed by the formatter to identify whitespace and to insert whitespace. The default is the rule named `WS`.
- `setIndentation(start, end)` increases the level of indentation when the element `start` is matched and decreases the level when element `end` is matched. The matching of elements happens in the same way as it does for formatting rules.

6.6.1.2. FormattingConfig Rules

Per default, the Declarative Formatter inserts one whitespace between two tokens. Rules can be used to specify a different behavior. They consist of two parts: *When* to apply the rule and *what* to do.

To understand *when* a rule is applied think of a stream of tokens whereas each token is associated with the corresponding grammar element. The rules are matched against these grammar elements. The following matching criteria exist.

- `after(ele)`: The rule is executed after the grammar element `ele` has been matched. For example, if your grammar uses the keyword ";" to end lines, this can instruct the formatter to insert a line-break after the semicolon.
- `before(ele)`: The rule is executed before the matched element. For example, if your grammar contains lists which separate its values with keyword ",", you can instruct the formatter to suppress the whitespace before the comma.
- `around(ele)`: This is the same as `before(ele)` combined with `after(ele)`.
- `between(ele1, ele2)`: This matches if `ele2` directly follows `ele1`. There may be no other elements in between.
- `bounds(ele1, ele2)`: This is the same as `before(ele1)` combined with `after(ele2)`.
- `range(ele1, ele2)`: The rule is enabled when `ele1` is matched, and disabled when `ele2` is matched. Thereby, the rule is active for the complete region which is surrounded by `ele1` and `ele2`.

The parameter `ele` can be a grammar's `AbstractElement` or a grammar's `AbstractRule`. However, only elements which represent a token in the textual representation can be matched. This are:

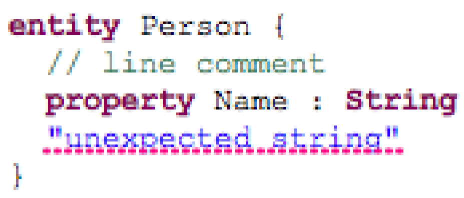
- terminal rules for comments.
- keywords, assignments, call to terminal or data-type rules.

After having explained how rules can be activated, this is what they can do:

- `setLinewrap()`: Inserts a line-break at this position.
- `setLinewrap(int)`: Inserts the specified number of line-breaks at this position.
- `setNoLinewrap()`: Suppresses automatic line wrap, which may occur when the line's length exceeds the defined limit.
- `setNoSpace()`: Suppresses the whitespace between tokens at this position. Be aware that between some tokens a whitespace is required to maintain a valid concrete syntax.

6.7. Syntax Coloring

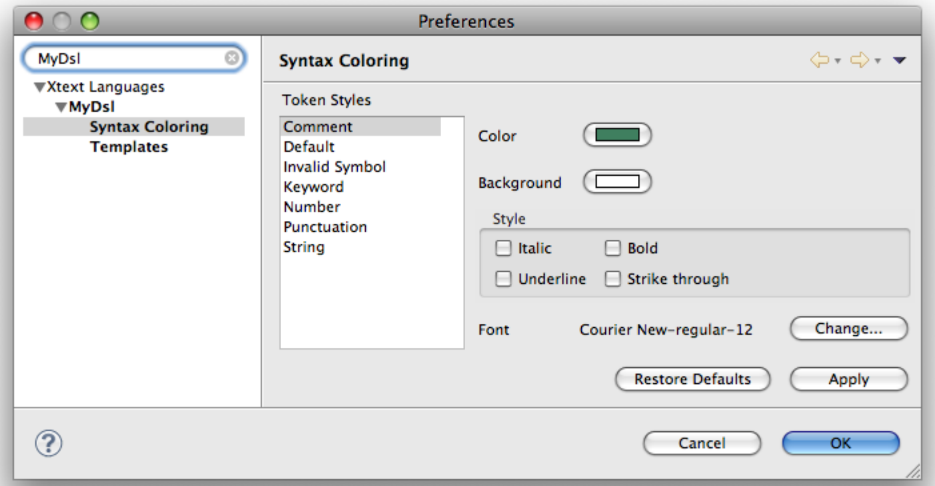
Besides the already mentioned advanced features like content assist and code formatting the powerful editor for your DSL is capable to mark-up your model-code to improve the overall readability. It is possible to use different colors and fonts according to the meaning of the different parts of your input file. One may want to use some decent colors for large blocks of comment while identifiers, keywords and strings should be colored differently to make it easier to distinguish between them. This kind of text decorating mark-up does not influence the semantics of the various sections but helps to understand the meaning and to find errors in the source code.



```
entity Person {
  // line comment
  property Name : String
  "unexpected string"
}
```

The highlighting is done in two stages. This allows for sophisticated algorithms that are executed asynchronously to provide advanced coloring while simple pattern matching may be used to highlight parts of the text instantaneously. The latter is called lexical highlighting while the first is based on the meaning of your different model elements and therefore called semantic highlighting.

When you introduce new highlighting styles, the preference page for your DSL is automatically configured and allows the customization of any registered highlighting setting. They are automatically persisted and reloaded on startup.



Font Courier New-regular-12

6.7.1. Lexical Highlighting

The lexical highlighting can be customized by providing implementations of the interface ILexicalHighlightingConfiguration and the abstract class AbstractTokenScanner. The latter fulfills the interface `ITokenScanner` from the underlying JFace Framework, which may be implemented by clients directly.

The `ILexicalHighlightingConfiguration` is used to register any default style without a specific binding to a pattern in the model file. It is used to populate the preferences page and to initialize the `ITextAttributeProvider`, which in turn is the component that is used to obtain the actual settings for a style's id. An implementation will usually be very similar to the DefaultLexicalHighlightingConfiguration and read like this:

```
public class DefaultLexicalHighlightingConfiguration
    implements ILexicalHighlightingConfiguration {

 public static final String KEYWORD_ID = "keyword";
 public static final String COMMENT_ID = "comment";

 public void configure(IHighlightingConfigurationAcceptor acceptor) {
  acceptor.acceptDefaultHighlighting(KEYWORD_ID, "Keyword",
    keywordTextStyle());
  acceptor.acceptDefaultHighlighting(COMMENT_ID, "Comment", // ...
  // ...
 }

 public TextStyle keywordTextStyle() {
  TextStyle textStyle = new TextStyle();
  textStyle.setColor(new RGB(127, 0, 85));
  textStyle.setStyle(SWT.BOLD);
  return textStyle;
 }

 // ...
}
```

Implementations of the `ITokenScanner` are responsible for splitting the content of a document into various parts, the so called tokens, and return the highlighting information for each identified range. It is critical that this is done very efficiently because this component is used on each keystroke. Xtext ships with a default implementation that is based on the lexer that is generated by ANTLR which is very lightweight and fast. This default implementation can be customized by clients easily. They simply have to bind another implementation of the AbstractAntlrTokenToAttributeIdMapper. To get an idea about it, have a look at the DefaultAntlrTokenToAttributeIdMapper.

6.7.2. Semantic Highlighting

The semantic highlighting stage is executed asynchronously in the background and can be used to calculated highlighting states based on the meaning of the different model elements. Users of the editor will notice a very short delay after they have edited the text until the styles are actually applied to the document. This keeps the editor responsive while providing aid when reading and writing your model.

As for the lexical highlighting there exist two interfaces whose implementations work closely together and allow the customization of the semantic highlighting. Namely these are the ISemanticHighlightingConfiguration and ISemanticHighlightingCalculator. While the configuration for the semantic highlighting works the same way as the `ILexicalHighlightingConfiguration`, the `ISemanticHighlightingCalculator` is the primary hook to implement the logic that will compute to-be-highlighted ranges based on the model elements.

The framework will pass the XtextResource and an IHighlightedPositionAcceptor to the calculator. It is ensured, that the resource will not be altered externally until the called method `provideHighlightingsFor` returns. The task is to navigate your semantic model and compute various ranges based on the node information and associate styles with them. This may read similar to the following snippet:

```
public void provideHighlightingFor(XtextResource resource,
    IHighlightedPositionAcceptor acceptor) {
 if (resource == null)
  return;

 Iterable<AbstractNode> allNodes = NodeUtil.getAllContents(
   resource.getParseResult().getRootNode());
 for (AbstractNode abstractNode : allNodes) {
  if (abstractNode.getGrammarElement() instanceof CrossReference) {
    acceptor.addPosition(node.getOffset(), node.getLength(),
      SemanticHighlightingConfiguration.CROSS_REF);
  }
 }
}
```

This example refers to an implementation of the `ISemanticHighlightingConfiguration` that registers a style for a cross reference. It is pretty much the same implementation as for the previously mentioned sample of an `ILexicalHighlightingConfiguration`.

```
public class SemanticHighlightingConfiguration
    implements ISemanticHighlightingConfiguration {

 public final static String CROSS_REF = "CrossReference";

 public void configure(IHighlightingConfigurationAcceptor acceptor) {
  acceptor.acceptDefaultHighlighting(CROSS_REF,
    "Cross References", crossReferenceTextStyle());
 }

 public TextStyle crossReferenceTextStyle() {
  TextStyle textStyle = new TextStyle();
  textStyle.setStyle(SWT.ITALIC);
  return textStyle;
 }
}
```

The implementor of an `ISemanticHighlightingCalculator` should be aware of performance to ensure a good user experience. It is probably not a good idea to traverse everything of your model when you will only register a few highlighted ranges that can be found easier with some typed method calls. It is strongly advised to use purposeful ways to navigate your model. The parts of Xtext's core that are responsible for the semantic highlighting are pretty optimized in this regard as well. The framework will only update the ranges that actually have been altered, for example. This optimizes the redraw process. It will even move, shrink or enlarge previously announced regions based on a best guess before the next semantic highlighting pass has been triggered after the user has changed the document.

6.8. Project Wizard

Optionally Xtext can also generate a project wizard for your DSL. Using this wizard a user can then with only a few clicks create a new project containing a sample model file and workflow. You enable the project wizard generation by adding the `SimpleProjectWizardFragment` fragment to the workflow:

```

<!-- project wizard fragment -->
<fragment class="org.eclipse.xtext.ui.generator.projectWizard.SimpleProjectWizardFragment"
 generatorProjectName="${projectName}.generator"
 modelFileExtension="mydsl"/>
```

Here:

- the `generatorProjectName` attribute is used to specify the generator project containing the workflows and templates used to generate artifacts from your DSL. The generated project wizard uses this to add a corresponding dependency to the created project.

- and the `modelFileExtension` specifies the default file extension associated with your DSL. When the user clicks the *Finish* button the project wizard will look for a file with the given extension in the created project and open it in the DSL editor.

After running the Xtext generator the DSL's UI project will contain a new Xpand template `MyDslNewProject.xpt` inside the `src` folder (under the package `<base-package>.ui.wizard`). This workflow will be run by the project wizard when the user clicks the *Finish* button and it is responsible for initializing the project with some sample content. The default Xpand template generated by the fragment will when executed by the wizard create a sample model file and a simple workflow to run the model through the generator project's `MyDslGenerator.mwe` workflow. As the Xpand template is in the `src` source folder you may of course modify it to generate whatever files you want. Just make sure to leave the top-level `main` definition in place, as that is the interface for the project wizard.

Note: To edit the generated Xpand template you shoud check that the JavaBeans meta model contributor is enabled under *Preferences > Xtend/Xpand*. Further you should also configure the UI project with the Xpand/Xtend nature using *Configure > Add Xpand/Xtend Nature* in the context menu.

6.8.1. Customizing the project wizard

To further customize the creation of the project (e.g. add a nature and a builder or some additional dependencies) you can implement your own *project creator*. The default project creator is represented by the generated class `MyDslProjectCreator` (suitable for subclassing) and will simply create a new project with Java and plug-in natures and execute the `main` definition of the Xpand template as described above.

To add more pages or input fields to the project wizard you should subclass the class `MyDslNewProjectWizard` as appropriate. Don't forget to register the subclass in the UI project's `plugin.xml`. You may also want to make this additional project info available to the Xpand template, in which case you should create a subclass for `MyDslProjectInfo` to hold that information, as this is the context object which gets passed to the template when it's executed.

Chapter 7. From oAW to TMF

TMF Xtext is a complete rewrite of the Xtext framework previously released with openArchitectureWare 4.3.1 (oAW). We refer to the version from oAW as oAW Xtext whereas the current Xtext version that is hosted at Eclipse.org will be called TMF Xtext to avoid confusion. oAW Xtext has been around for about 2 years before TMF Xtext was released in June 2009 and has been used by many people to develop little languages and corresponding Eclipse-based IDE support.

TMF Xtext has been improved in many aspects compared to the former version. While it integrates far better into EMF, it offers new fundamental features as well. The overhauled architecture leads to better performance when working with large models and since the whole framework is wired via dependency injection it is highly customizable. Last not least a test coverage of more than 2.000 unit tests provide confidence in the overall quality of TMF Xtext. We have been using the framework in production environments since one of the earlier milestones.

In this document we want to share the experience we made when migrating existing Xtext projects. The document describes the differences between oAW Xtext and TMF Xtext and is intended to be used as a guide to migrate from oAW Xtext to TMF Xtext. For people already familiar with the concepts of oAW Xtext it should also serve as a shortcut to learn TMF Xtext.

7.1. Why a rewrite?

The first thing you might wonder about is why we decided to reimplement the framework from scratch as opposed to use the existing code base and enhance it further on. We decided so because we had learned a lot of lessons from oAW Xtext. Although we wanted to stick with many proven concepts we found the implementation was lacking a solid foundation (the author of these lines is the original author of that non-solid code btw. :-)). The first version of oAW Xtext was basically a proof of concept which was so well received that it had been extended with all kinds of features (some were good, some were bad). Unfortunately code quality, clean and orthogonal concepts and test coverage did not receive the necessary focus.

In addition to this aspects of quality, oAW Xtext suffers from some severe performance problems. The extensive and naive use of Xtend (see next section) prevented many users to use oAW Xtext for growing real-world models.

7.2. Migration overview

Although a couple of things have changed we tried to keep good ideas and left many things unchanged. At the same time we wanted to clean up poor concepts and solve the main problems we and you had with oAW Xtext. From a bird's eye view if you want to migrate an existing oAW Xtext project to TMF Xtext, you mainly just need to rename the old grammar from *.xtxt to *.xtext and add two lines to the beginning of that document (see below for details). You might also have to change a few keywords, but all in all this is pretty easy and we've migrated a couple of oAW Xtext projects this way without problems. The other aspect where lots of code might have been written for is validation. In oAW Xtext we used Xpand's Check language to define constraints on the Ecore model. Even though this has been one major reason for the lack of scalability in Xtext we decided to keep the Check language as an option for compatibility reasons. Therefore, you do not need to translate your existing checks to a different language. Even better, you can overcome some performance issues by leveraging the newly introduced hooks to control the time of validation (while you type, on save, or on triggering an explicit action). Anyway, if you want to provide a slick user experience validation should run fast while you type. Therefore, we strongly encourage you to implement validation using our declarative Java approach.

We've developed and reviewed a lot of oAW Xtext projects and saw that most of the work was done in the grammar and in the validation view point. Other aspects such as outline view, label provider or content assist have been customized too, but they usually do not contain complicated Xtend code. In some projects the exception was linking and content assist which in oAW Xtext usually forces one to write a lot of duplicated code. While working on this we came up with a new concept called “scopes” that not only streamlines implementation in terms of redundancy. Scopes also increase the overall performance of Xtext. But since the concept of scopes was not carved out in oAW Xtext one usually implemented a cluttered and duplicated poor copy through linking and content assist. For obvious reasons, we didn't manage to come up with a good compatibility layer. So this is where most of the migration effort will go into if implemented customized linking. But we think the notion of scopes is such a valuable addition that it is worth the refactoring. Also, when looking at existing oAW Xtext projects we found that most projects either didn't change the default linking that much or they came up with their own linking framework anyway.

However, if we have completely misunderstood the situation and your oAW Xtext project cannot be migrated in a reasonable amount of time, please tell us. We want to help you!

7.3. Where are the Xtend-based APIs?

One of the nice things with oAW Xtext was the use of Xtend to allow customizing different aspects of the generated language infrastructure. Xtend is a part of the template language Xpand, which is shipped with oAW (and now is included in M2T Xpand). It provides a nicer expression syntax than Java. Especially the existence of higher-order functions for collections is extremely handy when working with models. In addition to the nice syntax, it provides dynamic polymorphic dispatch, which means that declaring e.g. label computation for an Ecore model is very convenient and type safe at the same time. In Java one usually has to write instanceof and cast orgies.

7.3.1. Xtend is hard to debug

While the aforementioned features allow the convenient specification of label and icon providers, outline views, content assist and linking, Xtend is interpreted and therefore hard to debug. Because of that Xpand is shipped with a special debugger facility. Unfortunately, this debugger cannot be used in the context of Xtext since it implies that the Xtend functions have to be called from a workflow. This is not and cannot be the case for Xtext Editors. As a result one has to debug his way through the interpreter, which is hard and inconvenient (even for us, who have written that interpreter).

7.3.2. Xtend is slow

But the problematic debugging in the context of Xtext was not the main reason why there are no Xtend-based APIs beside Check in TMF Xtext. The main reason is that Xtend is too slow to be evaluated “inside” the editor again and again while you type. While Xtend's performance is sufficient when run in a code generator, it is just too slow to be executed on keystroke (or 500ms after the last keystroke, which is when the reconciler reparses, links and validates the model). Xtend is relatively slow, because it supports polymorphic dispatch (the cool feature mentioned above), which means that for each invocation it matches at runtime which function is the best match and it has to do so on each call. Also Xtend supports a pluggable typesystem, where you can adapt to other existing type systems such as JavaBeans or Ecore. This is very powerful and flexible but introduces another indirection layer. Last but not least the code is interpreted and not compiled. The price we pay for all these nice features is reduced performance.

In addition to these scalability problems we have designed some core APIs (e.g. scopes) around the idea of Iterables, which allows for lazy resolution of elements. As Xtend does not know the concept of Iterators you would have to work with lists all the time. Copying collections over and over again is far more expensive than chaining Iterables through filters and transformers like we do with Google Collections in TMF Xtext.

7.3.3. Convenient Java

To summarize the dilemma we had to find a way to allow for convenient, scalable and debuggable APIs. Ultimately we wanted to provide neat DSLs for every view point, which provide all these things. However, we had to prioritize our efforts with the available resources in mind. As a result we found ways and means to tweak Java as good as possible to allow for relatively convenient, high performing implementations.

Java is fast and can easily be debugged but ranks behind Xtend regarding convenience. We address this with different approaches to make Java development in the context of Xtext as comfortable as possible.

Most of the APIs in TMF Xtext use polymorphic dispatching, which mimics the behavior known from Xtend. Another valuable feature of Xtend while working with oAW Xtext is static type checking while working with the inferred Ecore model whereas in Java the work with dynamic Ecore classes was rather cumbersome. Since TMF Xtext generates static Ecore classes per default you get static typing in Java as well. Additionally, the use of Google Collections reduces the pain when navigating over your model to extract information.

With these techniques an ILabelProvider that handles your own EClasses `Property` and `Entity` can be written like this:

```
public class DomainModelLabelProvider extends DefaultLabelProvider {

  String label(Entity e) {
    return e.getName();
  }

  String image(Property p) {
    return p.isMultiValue() ? "complexProperty.gif": "simpleProperty.gif";
  }

  String image(Entity e) {
    return "entity.gif";
  }

}
```

As you can see this is very similar to the way one describes labels and icons in oAW Xtext, but has the advantage that it is easier to test and to debug, faster and can be used everywhere an ILabelProvider is expected in Eclipse.

7.3.4. Conclusion

Just to get it right, Xtend is a very powerful language and we still use it for its main purpose: code generation and model transformation. The whole generator in TMF Xtext is written in Xpand and Xtend and its performance is at least in our experience sufficient for that use case. Actually we were able to increase the runtime performance of Xpand by about 60% for the Galileo release of M2T Xpand. But still, live execution in the IDE and on typing is very critical and one has to think about every millisecond in this area.

As an alternative to the Java APIs we also considered other JVM languages. We like static typing and think it is especially important when processing typed models (which evolve heavily). That's why Groovy or JRuby were no alternatives. Using Scala would have been a very good match, but we didn't want to require knowledge of Scala so we didn't use it and stuck to Java.

7.4. Differences

In this section differences between oAW Xtext and TMF Xtext are outlined and explained. We'll start from the APIs such as the grammar language and the validation and finish with the different hooks for customizing linking and several UI aspects, such as outline view and content assist. We'll also try to map some of the oAW Xtext concepts to their counterparts in TMF Xtext.

7.4.1. Differences in the grammar language

When looking at a TMF Xtext grammar the first time it looks like one has to provide additional information which was not necessary in oAW Xtext. In oAW Xtext *.xtxt files started with the first production rule where in TMF Xtext one has to declare the name of the language followed by declaration of one or more used/generated Ecore models:

TMF Xtext heading information

```
grammar my.namespace.Language with org.eclipse.xtext.common.Terminals
generate myDsl "http://www.namespace.my/2009/MyDSL"

FirstRule : ...
```

In oAW Xtext this information was provided through the generator (actually it is contained in the *.properties file) but we found that these things are very important for a complete description of a grammar. Therefore we made that information becoming a part of the grammar language in order to have self-describing grammars and allow for sophisticated static analysis, etc..

Apart from the first two lines the grammar languages of both versions are more or less compatible. The syntax for all the different EBNF concepts (alternatives, groups, cardinalities) is similar. Also assignments are syntactically and semantically identical in both versions. However in TMF Xtext some concepts have been generalized and improved:

7.4.1.1. String rules become Datatype rules

The very handy String rules are still present in TMF Xtext but we generalized them so that you don't need to write the 'String' keyword in front of them and at the same time these rules can not only produce EStrings but (as the name suggests) any kind of EDataType. Every parser rule that neither includes assignments nor calls any other that does, returns an EDataType containing the consumed data. Per default this is an EString but you can now simply create a parser rule returning other EDataTypes as well (see Datatype rules).

```
Float returns ecore::EDouble : INT ('.' INT)?;
```

7.4.1.2. Enum rules

Enum rules have not changed significantly. The keyword has changed to be all lower case ('enum' instead of 'Enum'). Also the right-hand side of the assignment is now optional. That is in oAW Xtext:

```
Enum MyEnum : foo='foo' | bar='bar';
```

becomes

```
enum MyEnum : foo='foo' | bar='bar';
```

and because the name of the literal equals the literal value one can omit the right-hand side in this case and write:

```
enum MyEnum : foo | bar;
```

7.4.1.3. Native rules

Another improvement is that we could replace the blackbox native rules with full-blown EBNF syntax. That is native rules become terminal rules and are no longer written as a string literal containing ANTLR syntax.

Example :

```
Native FOO : "'f' 'o' 'o'";
```

becomes

```
terminal FOO : 'f' 'o' 'o';
```

See the reference documentation for all the different expressions possible in terminal rules.

7.4.1.4. No URI terminal rule anymore

We decided to remove the URI terminal. The only reason for the existence was to mark the model somehow so that the framework knows what information to use in order to load referenced models. Instead we decided to solve this similar to how we imply other defaults: by convention.

So instead of using a special token which is syntactically a STRING token, the default import mechanism now looks for EAttributes of type EString with the name 'importURI'. That is if you've used the URI token like this:

```
Import : 'import' myReference=URI;
```

you’ll have to rewrite it that way

```
Import : 'import' importURI=STRING;
```

Although this changes your Ecore model, one usually never used this reference explicitly as it was only there to be used by the default import mechanism. So we assume and hope that changing the reference is not a big deal for you.

7.4.1.5. Return types

The syntax to explicitly declare the return type of a rule has changed. In oAW Xtext (where this was marked as ‘experimental’) the syntax was:

```
MyRule [MyType] : foo=ID;
```

in TMF Xtext we have a keyword for this :

```
MyRule returns MyType : foo=ID;
```

This is a bit more verbose, but at the same time more readable. And as you don’t have to write the return type in most situations, it’s good to have a more explicit, readable syntax.

7.4.2. Differences in Linking

The linking has been completely redesigned. In oAW Xtext linking was done in a very naive way: To find an element one queries a list of all ‘visible’ EObjects, then filters out what is not needed and tries to find a match by comparing the text written for the crosslink with the value returned by the `id()` extension. As a side-effect of `link_feature()` the reference is set.

The code about selecting and filtering `allElements()` usually has been duplicated in the corresponding content assist function, so that linking and content assist are semantically in sync. If you’re good (we usually were not) you externalized that piece of code and reused the same extension in content assist and linking.

To put it bluntly this approach could be summarized in two steps:

1. Give me the whole universe including every unregarded object in the uncharted backwaters of the unfashionable end of the western spiral arm of the galaxy and squeeze it into an Arraylist
2. From this, select the one I need

This was not only very expensive but also lacks an important abstraction: the notion of scopes.

7.4.2.1. The idea of scopes

In TMF Xtext we’ve introduced scopes and scope providers that are responsible for creating scopes. A scope is basically a set of name->value pairs. Scopes are implemented upon Iterables and are nested to build a hierarchy. With scopes we declare “visible” objects in a lazy and cost-saving way where the linker only navigates as far as necessary to find matching objects. The content assist reuses this set of visible objects to offer only reachable objects.

When the linking has to be customized scoping is where most of the semantics typically goes into. By implementing an IScopeProvider for your language linking and content assist will automatically be kept in sync since both use the scope provider.

The provided default implementation is semantically mostly identical to how the default linking worked in oAW Xtext:

1. Elements which have an attribute ‘name’ will be made visible by their name
2. Referenced resources will be put on the (outer) scope by using the ‘importURI’- naming convention and will only be loaded if necessary
3. The available elements are filtered by the expected type (i.e. the type of the reference to be linked)

7.4.2.2. Migration

We expect the migration of linking to be very simple if you’ve not changed the default semantics that much. We’ve already migrated a couple of projects and it wasn’t too hard to do so. If you have changed linking (and also content assist) a lot, you'll have to translate the semantics to the IScopeProvider concept. This might be a bit of work, but it is worth the effort as this will clean up your code base and better separate concerns.

7.4.3. Differences in UI customizing

In oAW Xtext several UI services such as content assist, outline view or the label provider have been customized using Xtend. In TMF Xtext there is no Xtend API for these aspects. Extensive model computations for the content assist is most probably not necessary anymore- it reuses scopes. And since we provide a declarative Java API that mimics the polymorphic dispatch and relies on static Ecore classes you will gain nearly the same expressiveness as before while increasing maintainability and performance.

Beside the API change in favor of Java we have to mention that in TMF Xtext the outline view does not support multiple view points so far. This is just because we didn't manage to get this included. We don't think that view points are a bad idea in general, but we decided that other things were more important.

7.5. New Features

This section provides an overview of new possibilities with TMF Xtext compared to oAW Xtext. Please note that this list is neither complete nor does it explain every aspect in detail to keep this document tight.

7.5.1. Dependency Injection with Google Guice

Beyond the mentioned architectural overhaul that carve out separate concerns in a meaningful way these different classes of TMF Xtext are wired using Google Guice and can easily be replaced by or combined with your own implementation. We could have foreseen some common needs for adaption but with this mechanism you can virtually change every aspect of Xtext without duplicating an unmanageable amount of code.

7.5.2. Improvements on Grammar Level

The Xtext grammar language introduces some new features, too. Read the chapter grammar language to understand the details about all the improvements that have been implemented.

Grammar mixins allow you to extend existing languages and change their concrete and abstract syntax. However the abstract syntax (i.e. the Ecore model) can only be extended. This allows you to reuse existing validations, code generators, interpreters or other code which has been written against those types.

In oAW Xtext common terminals like ID, INT, STRING, ML_COMMENT, SL_COMMENT and WS (whitespace) were hard coded into the grammar language and couldn't be removed and hardly overridden. In TMF Xtext these terminals are imported through the newly introduced grammar mixin mechanism from the shipped grammar `org.eclipse.xtext.common.Terminals` per default. This means that they are still there but reside in a library now. You don't have to use them and you can come up with your own set of reusable rules.

Reusing existing Ecore models in oAW Xtext didn't work well and we communicated this by flagging this feature as 'experimental'. In TMF Xtext importing existing Ecore models is fully supported. Moreover, it is possible to import a couple of different EPackages and generate some at the same time, so that the generated Ecore models extend or refer to the existing Ecore models.

The grammar language gained one new concept that is of great value when writing left-factored grammars (e.g. expressions). With actions one can do minor AST rewritings within a rule to overcome degenerated ASTs. You will find an in-depth explanation of actions in the dedicated chapter in the reference documentation.

7.5.3. Fine-grained control for validation

In order to make more expensive validations possible without slowing down the editor, TMF Xtext supports three different validation hooks.

1. FAST constraints are triggered by the reconciler (i.e. 500 ms after the last keystroke) and on save.
2. NORMAL constraints are executed on save only.
3. EXPENSIVE constraints are executed through an action which is available through the context menu.

Please note that when using Xtext models for code generation the checks of all three categories will be performed.

Beside this it is now possible to add information about the feature which is validated.

```
context Entity#name ERROR "Name should start with a capital "+this.name+"." :
  this.name.toFirstUpper() == this.name;
```

If you add the name of a feature prepended by a hash (‘#’) in Check, the editor will only mark the value of the feature (name), not the whole object (Entity). Both concepts, control over validation time as well as pointing to a specific feature, complement one another in Check and Java based validation.

7.6. Migration Support

In this document we tried to explain why we decided to change some aspects of Xtext’s architecture. We consider most changes as minor but when it comes to scopes you will face a conceptual enhancement that did not exist in oAW Xtext. We tried to explain why it is not easily possible to come up with an adapter for scoping.

That said you might not have the time to do the migration and wished to have advice for migrating, especially from oAW linking to TMF linking. You’re welcome to ask any questions in the newsgroup and we’ll try to help you as much as possible in order to get your projects migrated. Also, if you don’t want to do the migration yourself we (itemis AG) can do the work for you or help you with that.

Chapter 8. The ANTLR IP issue (or which parser to use?)

In order to be able to parse models written in your language, Xtext needs to provide a special parser for it. The parser is generated from the language grammar.

Currently it is recommended to use the ANTLR-based parser. ANTLR is a very sophisticated parser generator framework based on a so called LL(*) algorithm. It is fast, simple and at the same time has some very nice and sophisticated features. Especially its support for error recovery is much better than what other parser generators provide.

ANTLR comes in two parts: the runtime and the generator. Both are shipped under the BSD license and have a clean intellectual property history. However the generator is still implemented in an older version of ANTLR (v 2.x), where it was not possible for the Eclipse Foundation to be sure where exactly every line of code originated. Therefore ANTLR v 2.x didn't get the required approval. Eclipse has a strict IP policy, which makes sure that everything provided by Eclipse can be consumed under the terms of the Eclipse Public License. The details are described in this document.

Unfortunately as the generator of ANTLR V3 needs ANTLR V2 it is as well not yet IP approved. That is why we are not allowed to ship Xtext with the ANTLR generator (the runtime is IP approved), but have to provide it separately via update-site at:

- http://download.itemis.com/updates/

IMPORTANT : *Although if you use the non-IP approved ANTLR generator, you can still ship any languages and the IDEs you've developed with Xtext without any worrying, because the **ANTLR runtime is IP approved***

8.1. What if I do not want to use non IP-approved code

If you, against all recommendations, need to stick to a fully IP approved generator, you can use the parser generator we've developed. But be warned, although it's fully functional and equally fast, it does not have any error recovery. This makes the editing experience in the editor more or less unacceptable.