Xtext User Guide

Xtext User Guide

Chapter 1. Overview

1. What’s Xtext?

The TMF Xtext project provides a domain-specific language (the grammar language) for description of textual programming languages and domain-specific languages. It is tightly integrated with the Eclipse Modeling Framework (EMF) and leverages the Eclipse Platform in order to provide language-specific tool support.

In contrast to common parser generators (like e.g. JavaCC or ANTLR), the grammar language is used to derive much more than just a parser and lexer (lexical analyzer).

From a grammar the following is derived:

- incremental, ANTLR 3 based parser and lexer
- Ecore-based meta models (optional)
- a serializer, used to serialize instances of such meta models back to a parseable textual representation
- a linker
- an implementation of the EMF Resource interface (based on the parser and the serializer)
- a full-fledged integration of the language into Eclipse IDE
  - syntax coloring
  - navigation (F3, etc.)
  - code completion
  - outline views
  - 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. For example if you don’t like the default code assistant implementation, all you need to do is to come up with an alternative implementation of the corresponding service and configure it via dependency injection.

2. Xtext concepts from a bird’s eye view

text

3. The Grammar Language

The grammar language is the corner stone of Xtext and is defined in itself – of course.

It is a DSL carefully designed for description of textual languages, based on LL(*)-Parsing that is like Antlr3's parsing strategy and supported by packrat parsers. The main idea is to describe the concrete syntax and how an EMF-based in-memory model is created during parsing.

3.1. First an 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];
```

In the following the different concepts of the grammar language are explained. We refer to this grammar when appropriate.

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

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

3.3. 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 Meta-Model Inference) but it is also possible to instantiate existing Ecore models. You can even mix this, use multiple existing ECore models and infer some others from one grammar.

3.3.1. EPackage generation

The easiest way to get started is to let Xtext infer the meta 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
```

Which means: generate an EPackage with name `secretcompartment` and nsURI `http://www.eclipse.org/secretcompartment` (these are the properties needed to create an EPackage). See Meta-Model Inference for details.

3.3.2. EPackage import

If you already have created such an EPackage somehow, you can import it using either a namespace URI or a resource URI (URIs are an EMF concept).

3.3.2.1. Using namespace URIs to import existing EPackages

You can import existing EPackages using the following syntax:

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

Note that if you use a namespace URI, the corresponding EPackage needs to be installed into the workbench, so that the editor can find it. At runtime (i.e. when starting the generator) you need to make sure that the corresponding EPackage is registered in `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="foo.bar.MyPackage"/>
</bean>
```

3.3.2.2. Using classpath URIs to import existing EPackages

Xtext provides a new resource URI scheme, which is backed by the Java classpath. If you want to refer to an ECore file `MyEcore.ecore`, provided in a package `foo.bar`, you could write

```
import "classpath:/foo/bar/MyEcore.ecore"
```

Using the classpath scheme is considered the preferred way.

3.3.3. Using multiple packages / meta model aliases

If you want to use multiple EPackages you need to specify aliases like so:

```
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 them 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 no two EClassifiers with the same name. In such cases none of them are referable. It is even possible to have multiple ''import''s and one ''generate'' 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 from 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 tackle 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.

3.4. Rules

The default parsing is based on a homegrown packrat parser. It can be substituted by an Anltr 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. At the moment it is advised to download the plugin de.itemis.xtext.antlr (from update site `http://www.itemis.com/xtext/updatesite/milestones`) 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

3.4.1. Terminal Rules

In the first phase, i.e. lexing, a sequence of characters (the text input) is transformed into a sequence of so called tokens. Each token consists of one or more characters and was matched by a particular terminal rule and represents an atomic symbol. 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 Mixins). Terminal rules are also referred to as token rules or lexer rules. There is an informal naming convention that terminal-rule names are all uppercase.

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, which is called 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 [[Xtext/Documentation/ValueConverter|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 datatype rule, instead of adding another terminal rule.

3.4.1.1. 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. [[Xtext/Documentation/ValueConverter|value converters]]).

The implementation needs to be registered as a service (cf. Service Framework).

This is also the point where you can remove things like quotes form string literals or the caret ('^') from identifiers.

3.4.2. 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 quantifier operator. There are four different quantities

1. exactly one (the default no operator)
2. one or none (operator "?")
3. any (operator "*")

4. one or more (operator “+”)

3.4.2.1. 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.

3.4.2.2. 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’.

3.4.2.3. Wildcard

If you want to allow any character you can simple write a dot: Example:

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

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

3.4.2.4. Until Token

With the until token it is possible to state that everything should be consumed until a certain token occurs. The multi line comment is implemented using it:

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

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

3.4.2.5. Negated Token

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

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

3.4.2.6. 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)*;
```

3.4.2.7. Alternatives

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

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

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

3.4.2.8. Groups

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

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

That is the 4-digit hexadecimal code of ascii characters.

3.5. Parser Rules

The parser reads in a sequence of terminals and walks through the parser rules. That's why a parser rule – contrary to a terminal rule – does not produce a single terminal token but a tree of non-terminal and terminal tokens that lead to a so called parse tree (in Xtext it is a.k.a node model). Furthermore, parser rules are handled as kind of a building plan to create EObjects that form the semantic model (the linked! AST). The different constructs like actions and assignments are used to derive types and initialize the semantic objects accordingly.

3.5.1. Extended Backus-Naur Form expressions

In parser rules (as well as in datatype rules) not all the expressions available for terminal rules can be used. Character ranges, wildcards, the until token and the negation are currently only available for terminal rules. Available in parser rules as well as in terminal rules are

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

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.

3.5.1.1. Assignments

Assignments are used to assign parsed information to a feature of the current EClass. The current EClass is specified by the return type resp. if not explicitly stated it is implied that the type's name equals the rule's name.

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
```

Where the left hand side refers to a feature of the current EClass (in this case EClass 'State'). 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 to the type of the expression on the right. As ID returns an EString the feature name needs to be of type EString as well.

3.5.2. 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) awaits a multiple feature and adds the value on the right hand to that feature, which is, of course, a list feature
3. the "?=" sign (boolean add operator) awaits a feature if type EBoolean and sets it to true if the right hand side was consumed (no matter with what values)

3.5.2.1. Cross References

A unique feature of Xtext is the ability to declare cross links in the grammar. In traditional compiler construction the cross links are not established during parsing but in a later linking phase. This is the same in Xtext, but we allow to specify cross link information in the grammar, which is used during the linking phase. The syntax for cross links 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 the type! This is sometimes confusing, because one usually uses the same name for the rules and the types. That is if we had named the type for events differently like in the following the cross references 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 followed by 'CrossReferenceableTerminal'. This is the part describing the concrete text, from which the crosslink later should be established. By default (that's why it's optional) it is "|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.

3.5.2.2. Simple Actions

By default the object to be returned by 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 not explicit return type is specified). With Actions however, creation of EObject can be made explicit. Xtext supports two kinds of Actions:

1. simple actions
2. assigned actions.

If at some point you want to enforce 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 EClass. 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 stub-rules.

3.5.2.3. 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” reference.

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 to 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=('?'|'+'|'*'))?;
```

Thus, to state the cardinality is optional (last question mark) and can be represented by a question mark, a plus, or an asterisk.

3.5.2.4. Tree Rewrite 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 and especially Antlr has a very nice generic error recovery mechanism. This allows to have AST constructed 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 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 ('+' TerminalExpression)?;

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

In practice this is always the same pattern and therefore not that problematic. However, by simply applying Xtext’s AST construction we know so far like so ...

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

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

... one would get unwanted elements in the resulting 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 actions:

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

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 'left'). In Java the semantics could be expressed like so:

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

3.6. 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 [[#Grammar_Mixins | 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 multiple-line comments (ML_COMMENT), single-line comments (SL_COMMENT), and whitespaces (WS) to be allowed between the ‘Fullname’ and the ‘age’. Because ‘Fullname’ does not introduce another set of hidden terminals, it allows the same symbols to appear between ‘firstname’ and ‘lastname’ as the calling rule ‘pPerson’. Thus, the following input is perfectly valid for the given grammar snippet:

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

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

3.7. Datatype rules

Datatype rules are parsing-phase rules, which create instances of EDatatype as terminal rules do. The nice thing about datatype 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)*;
```

Which 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 hide the ID rule.

In addition having this defined as a datatype 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 like in terminal rules:

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

Note that if a rule does not call another parser rule and does not contain any actions nor assignments (see parser rules), it is considered a datatype rule and the datatype EString is implied if not explicitly declared differently.

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

3.8. Enum Rules

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

If you want to define a ChangeKind [[http://help.eclipse.org/stable/index.jsp?topic=/org.eclipse.emf.doc/references/javadoc/org/eclipse/emf/ecore/change/impl/package-summary.html 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. 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;
```

4. Meta-Model inference

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

Xtext uses Ecore EPackages to define meta models. Meta 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.

4.1. Type and Package Generation

Xtext creates

- an EPackage
  - for each generated 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 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 datatype 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.

4.2. 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 type will be the return type of the called parser rule.
  - for each 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/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 inherits from the current return type of the parser rule. You can specify additional common supertypes by means of "artificial" parser rules, that are never called, e.g.

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

4.3. Enum Literal Generation

For each alternative defined in an enum rule, the transformer creates an enum literal, when another 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 ommitted, you'll get an enum literal with equal 'name' and 'literal' attributes.

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

4.4. Feature Normalization

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

4.5. Customized Post Processing

As a last step, the generator invokes the post processor for every generated meta model. The post processor expects an Xtend file with name `MyDslPostProcessor.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) :
  ...
;
```

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
- add operations with implementation (using a body annotation)

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

4.6. Error Conditions

The following conditions cause an error

- An EAttribute or EReference has two different types or different cardinality.
- There are 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 supertype is added to an imported type.
- An EClass is added to an imported EPackage.
- An undeclared alias is used.
- An imported metamodel cannot be loaded.

5. Importing existing Meta Models

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

```
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 lexer rules.

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

6. Grammar Mixins

Xtext supports the reuse of existing grammars. Grammars that are created via the Xtext wizard extend `org.eclipse.xtext.common.Terminals` by default.

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

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

....
```

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

Example :

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

grammar my.SubGrammar with my.SuperGrammar

Model : (ruleAs+=RuleA)*;

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

Note that declared terminal rules always come before any imported / mixed-in terminal rules.

7. Default tokens

Xtext is shipped 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: . ;
```

8. The Generator

Text

9. Dependency Injection in Xtext with Google Guice

9.1. Modules

The Guice Injector configuration is done through the use of Modules (also a Guice concept). The generator provides two modules when first called, one for runtime ([MyLanguage]RuntimeModule) and one for UI ([MyLanguage]UIModule). These modules are initially empty and intended to be manually edited when needed. These are also the modules used directly by the setup methods. By default these modules extend a fully generated module.

9.1.1. Generated Modules

The fully generated modules (never touch them!) are called Abstract[MyLanguage]RuntimeModule and Abstract[MyLanguage]UiModule respectively. They contain all components which have been generated specifically for the language at hand. Examples are: the generated parsers, serializer or for UI a proposal provider for content assist is generated. What goes into these modules depends on how your generator is configured.

9.1.2. Default Modules

Finally the fully generated modules extend the `DefaultRuntimeModule` (resp. `DefaultUiModule`), which contains all the default configuration. The default configuration consists of all components for which we have generic default implementations (interpreting as opposed to generated). Examples are all the components used in linking, the outline view, hyperlinking and navigation.

9.1.3. Changing Configuration

We use the primary modules ([MyLanguage]RuntimeModule and [MyLanguage]UiModule) in order to change the configuration. The class is initially empty and has been generated only to allow for arbitrary customization.

In order to provide a simple and convenient way, in TMF Xtext every module extends AbstractXtextModule. This class allows to write bindings like so:

```
public Class<? extends MyInterface> bind[anyname]() {
     return MyInterfaceImpl.class;
 }
```

Such a method will be interpreted as a binding from `MyInterface` to `MyInterfaceImpl`. 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. 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 bind(Binding)` and do what every you want.

Chapter 2. Runtime Concepts

TMF Xtext itself and every language infrastructure developed with TMF Xtext is configured and wired-up using dependency injection (DI).

We use Google Guice as the underlying framework, and haven't built much on top of it as it pretty much does what we need. So instead of describing how google guice works, we refer to the website, where additional information can be found: http://code.google.com/p/google-guice/.

Using DI allows everyone to set up and change all components. This does not mean that everything which gets configured using DI (we use it a lot) is automatically public API. But we don't forbid use of non-public API, as we think you should decide, if you want to rely on stable API only or use things which might be changed (further enhanced ;-)) in future. See Xtext API Documentation.

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.

The setup method also registers the ResourceFactory and generated EPackage with 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.

2. Setup within Eclipse / Equinox

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>
```

3. Validation

Validation (a.k.a Static Analysis) 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

3.1. 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()`

3.2. Cross-link Validation

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

Any unresolvable cross-links will be reported through:

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

3.3. Custom Validation

In addition to the afore mentioned kinds of validations, 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` (`org.eclipse.emf.ecore.EValidator.Registry.INSTANCE`) can only be filled programatically, that means that there's no equinox extension point similar to the EPackage- and ResourceFactory registries.

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're 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 contains 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 (the name says it) declarative way. 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 constraints method is, just by declaring a typed parameter. This also let's you avoid any castings. In addition to the reflective invocation of test methods the AbstractDeclarative provides a couple of convenience assertions.

All in all this is very similar to how Junit work. 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);
  }
}
```

4. 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 cross link in the grammar (at least in the meta model)
2. specification of linking semantics

4.1. Declaration of cross links

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

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

Example:

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

The meta model derivation 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 (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 "EntityWithÄÖÜ"
```

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

4.2. Specification of linking semantics

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

```
@Stable(since = "0.7.0", subClass = AbstractLinkingService.class)
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.
   *
   * @param object
   * @param reference
   * @param context
   * @return the text representation.
   */
  String getLinkText(
    EObject object,
    EReference reference,
    EObject context);
```

```
}
```

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

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

4.3. Default linking semantics

The default implementation for all languages, looks within the current file for an EObject of the respective type (‘Entity’) which 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;`).

4.3.1. Default Imports

There is a default implementation for inter-resource referencing, which as well uses convention:

Given the grammar :

```
Model : (imports+=Import)* (stuff+=(Ref|Entity))*;
 Import : 'import' importURI=STRING ';';
 Ref : 'ref' referencedObject=[Entity|ID] ';';
 Entity : 'entity' name=ID ';';
```

With this grammar in place it would be possible to write three files in the new DSL 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 resulting default scope list is as follows:

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

So, the outer scope is asked for an Entity named `Foo`, as it does not contain such a declaration itself its parent is asked and so on. The default implementation of `IScopeProvider` creates this kind of scope chain.

5. Scoping

An `IScopeProvider` is responsible for providing an `IScope` for a given `EObject` and its `EReference`, for which all candidates shall be returned.

```
@Stable(since = "0.7.0", subClass = AbstractScopeProvider.class)
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 an element shall be referenced
   * @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);

  /**
   * Returns a scope for a given context. The scope contains any visible,
   * type-compatible element.
   * @param context the element from which an element shall be referenced
   * @param type the (super)type of the elements.
   * @return {@link IScope} representing the inner most {@link IScope} for
   *         the passed context and type.
   */
  public IScope getScope(EObject context, EClass type);
}
```

An `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 so:
   //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 .

5.1. Declarative Scope Provider

As always there is an implementation allowing to specify scoping in a declarative way (`org.eclipse.xtext.crossref.impl.AbstractDeclarativeScopeProvider`). It looks up methods which have the following signature:

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

For example if you have a transition contained in a scope and you want to compute all reachable states the corresponding method could be declared as follows:

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

If such a method does not exist, the implementation will try to find one for the context's container. This allows to reuse the same scope for different elements and references. In the case of a state machine you might want to declare the scope with available states per state machine. This can simply be done using the following signature:

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

For a transition looking for possible states to link to, the implementation would first look for `scope_State(Transition,EReference)`, then for the container of the transition. Assuming that is a State the implementation would look for `scope_State(State,EReference)` and so on. Until it finds a matching method (e.g. `scope_State(StateMachine,EReference)`)

6. Value Converter

Value converters are registered to convert parsed text into a certain data type instance and back. The primary hook is called org.eclipse.xtext.conversion.IValueConverterService and the concrete implementation can be registered via the runtime Guice module (TODO reference to framework description).

6.1. Annotation based value converters

The most simple way to register additional value converters is to make use of `org.eclipse.xtext.conversion.impl.AbstractAnnotationBasedValueConverterService`, which allows to declaratively register `IValueConverter` via annotated methods.

The implementation for the default token grammar looks like

```
public class DefaultTerminalConverters
    extends AbstractAnnotationBasedValueConverterService {

  private Grammar grammar;

  @Inject
  public void setGrammar(IGrammarAccess grammarAccess) {
    this.grammar = grammarAccess.getGrammar();
  }

  protected Grammar getGrammar() {
    return grammar;
  }

  @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 overwrite or add addition value converter by adding the respective methods.

Imagine, you would want to add a rule creating BigDecimals:

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

7. Transient Values

text

8. 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.
     * @param resource
     * @param fragment
     * @return the EObject
     */
    EObject getEObject(Resource resource, String fragment);
    }
```

However, the currently available default fragment provider does nothing.

Chapter 3. 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] ")" ;
```

1. Managing Concurrency

text

2. Label Provider

text

3. 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:

3.1. 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 placeholders that should give a clue about the naming scheme, that is 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 in turn 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.*'' may be called.

3.2. 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);
}
```

4. 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 under `templates/templates.xml` containing templates in a format described in the eclipse help .

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 `org.eclipse.xtext.ui.common.editor.templates.XtextTemplateContextTypeRegistry` and configure it via Guice.

4.1. CrossReference Resolver

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

The syntax is as follows:

```
${someText: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.

5. Outline View

text

6. Navigation and Hyperlinking

text

7. Formatting (Pretty Printing)

text

Chapter 4. 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. 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.

This 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 serves as a shortcut to learn TMF Xtext.

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 and wanted to stick with many proven concepts but found the implementation to lack 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 just a proof of concept and 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 had not the needed focus.

In addition to this general lack of code quality, oAW Xtext suffers from some severe performance problems. The naive use of Xtend (see next section) made use of bigger models in Xtext editors unusable.

2. 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 nicer expression syntax than Java especially the existence of higher-order functions for collections are 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 a meta model is very convenient and type safe. In contrast using Java one usually has to write instanceof and cast orgies.

2.1. Xtend is hard to debug

While the aforementioned things allow for 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, however using the shipped debugger implies that the Xtend functions are called from a workflow. This is not and cannot be the case in Xtext. 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.

2.2. Xtend is slow

But the problematic debugging was not the main reason why there are no Xtend-based APIs in TMF Xtext. The main reason is that Xtend is too slow to be evaluated “live in the editor”, that is evaluating lots of Xtend code while you type. While Xtend’s performance is sufficiently good 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 wich function is the best match and it has to do so on each call. Also Xtend supports a plggable typesystem, where you can adapt to other existing type systems such as JavaBeans or Ecore. Last but not least the code is interpreted and not compiled once. The price we pay for all these nice features reduced performance.

In addition to these scalability problems we have designed seom core concepts around the idea of Iterables, which allows for lazy resolution of things. As Xtend does not know the concept of Iterators you’ld have work with lists all the time. This is far more expensive than chaining Iterables through filters and transformers like we do with Google Collections in TMF Xtext.

2.3. Declarative Java

To summarize this 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 Javaas good as possible to allow for relatively convenient, high performing implementations.

To allow convenience in Java we tried to mimic the nice things of Xtend in Java. This is done through two things

2.3.1. Polymorphic Dispatcher

Most of the APIs in TMF Xtext use polymorphic dispatching, which mimics the polymorphic dispatch know from Xtend. A ILabelProvider to be used in any Eclipse view can be writte like so for instance:

```
public class XtextLabelProvider extends DefaultLabelProvider {

 String text(Grammar grammar) {
  return GrammarUtil.getName(grammar);
 }

 String image(Grammar grammar) {
  return "language.gif";
 }

 String image(GeneratedMetamodel metamodel) {
  return "export.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.

2.3.2. Google Collections

The other thing we love about Xtend is its convenient way to navigate a model. This is something which can't be done with Java, as it lacks closures and in general requires to write lots of boilerplate such as superfluos type information, etc. So this is where you have to make compromises. Anyway, we think this could be improved a bit by using Google Collections which is (the name suggests it) a collections framewokr written by google. It's open-source and there's a JSR proposing to add the framework to the JDK, which would IMHO be a very good addition.

It provides lots of nice static factory methods similar to what we have in java.util.Collections and java.util.Arrays, contains higher-order functions based on a function type included in the library and a couple of very good collection implementations such as multi maps and immutable implementations of the various collection types. With this it is possible to write a chain of filters and transformers like so.

```
  ArrayList<String> names = newArrayList("foo", "bar", "honk");
Iterable<String> filtered = filter(names, not(isEqualTo("bar")));
Iterable<Integer> lengths = transform(filtered,new Function<String, Integer>() {
 public Integer apply(String from) {
  return from.length();
 }
});
```

or in a more functional way :

```
transform(
 filter(
  newArrayList("foo", "bar", "honk"),
  not(isEqualTo("bar"))
 ),
 new Function<String, Integer>() {
  public Integer apply(String from) {
   return from.length();
```

```
  }
 }
);
```

From a syntactical point of view Google Collections is in no way a replacement for real closures and a non-verbose expression language like we have in Xtend, but it's a big improvement over traditional Java programming with java.util.* and it performs much better than Xtend.

2.4. Conclusion

Just to get it right, Xtend is a very nice language and we still use it for it's main purpose: code generation and model transformation. The whole generator in TMF Xtext is written in Xpand and Xtend and it's performance is at least in our experience sufficient for that use case. Actually we were able to increase 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.

3. Differences

In this section differences between oAW Xtext and TMF Xtext are outlined and explained. We'll start from the primary APIs such as the grammar language and the validation hook and finish with the different secondary hooks for customizing linking and several UI apsects, such as outline view and content assist.

3.1. Differences in the grammar language

When looking at a TMF Xtext gramar 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 meta models. 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 and had therefore be part of the grammar in order to have self-describing grammars and allow for sophisticated static analysis, etc..

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

3.1.1. 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. The return type is inferred and if not specified EString is assumed, however you can now simply create a parser rule returning other EDatatypes.

3.1.2. Enum rules

Enum rules have not changed significantly. The keyword has changed to be all lower case ('enum' instead of 'Enum'). Also the right handside 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 ommit the right handside in this case and write:

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

3.1.3. Native rules

The most significant improvement to oAW Xtext is that we could replace the blackbox native rules with full-blown EBNF syntax. That is native rules become terminal rules and or no longer written as a string literal containing ANTLR syntax but are a part of the language.

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.

3.1.4. No built-in terminals

In oAW Xtext common terminals lie ID, INT, STRING, ML_COMMENT, SL_SOMMENT and WS (whitespace) were hard coded into the grammar language and couldn't be removed. Also overriding was error-prone and challenged. In TMF Xtext these terminals are important through the newly introduced grammar mixin mechanism. This means that they are still there but they are now libraries. You don't have to use them and you can come up with your own reusable rules.

3.1.5. No URI terminal rule anymore

Although with grammar-mixins we would have been able to implement the URI terminal rule again, we decided to remove it as we only wanted to mark the model some how to identify what information to use in orde to load referenced models. Instead we decided to solve this similar to how we do resolution by name: We use convention. That is if you've used the URI token like so:

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

you'll have to rewrite it like so

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

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