Xtext User Guide

Xtext User Guide

Chapter 1. Overview

1. What is Xtext?

Xtext is a framework for development of domain-specific language. 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), 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 do not 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.

1.1. What is a domain-specific language (DSL)

A 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.

2. Xtext concepts from a bird’s eye view

text

Chapter 2. 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.

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];
```

2. The Syntax

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

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 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).

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 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.

2.2.1. 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). The whole algorithm used to derive complete Ecore models from Xtext grammars is described in section Meta-Model Inference.

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 (URIs are an EMF concept).

2.2.2.1. 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>
```

using namespace URIs is typically only interesting when using common Ecore models, such as Ecore itself or the UML metamodel. 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.

2.2.2.2. Using resource URIs to import existing EPackages

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

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

```
import "platform:/resource/my.project/modelFolder/MyEcore.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 to the genmodels for the referenced Ecore models. Example:

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

2.2.3. Using multiple packages / meta model aliases

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 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 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.

2.3. 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

2.3.1. 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). 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 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.

2.3.1.1. 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. 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 from string literals or the caret ('^') from identifiers.

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 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 "+")

2.3.2.1. 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.

2.3.2.2. 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'.

2.3.2.3. 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'.

2.3.2.4. 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 ‘*/’.

2.3.2.5. Negated Token

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

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

2.3.2.6. 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)*;
```

2.3.2.7. 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’)

2.3.2.8. 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.

2.4. 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 a as kind of a building plan for how 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.

2.4.1. 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.

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.

2.4.1.1. 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 with the type of the expression on the right. As ID returns an EString the feature name needs to be of type EString as well.

2.4.2. 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) expects a multi valued 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) expects a feature of type EBoolean and sets it to true if the right hand side was consumed (no matter with what values)

2.4.2.1. 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.

2.4.2.2. 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 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.

2.4.2.3. 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.

2.4.2.4. 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 the Xtext AST construction features we’ve covered so far, a grammar ...

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

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 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 these semantics could be expressed as:

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

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 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 ‘Person’. 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.

2.6. 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).

2.7. 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 infer enums and their respective literals during the metamodel 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. 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;
```

3. 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.

3.1. 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.

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 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;
```

3.3. 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;
```

3.4. 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.

3.5. 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).

3.6. 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.

4. 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')+;
```

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.

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

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

....
```

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)*;

// overwrites my.SuperGrammar.RuleB
```

```
RuleB : '[' name=ID ']';
```

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

6. 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: . ;
```

Chapter 3. Configuration

1. The Generator

Xtext provides lots of generic implementations for you 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.

1.1. A short introduction to MWE

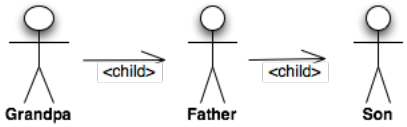
The nice thing about MWE is that it just instantiates java classes and the configuration is done through setter and adder methods. Given the following Java class :

```
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 like this:

```
<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 `x` is ignored in this case. The attribute `class` tells MWE which class to use in order to create the object node. That 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 there 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 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.

This is why it is ok to write:

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

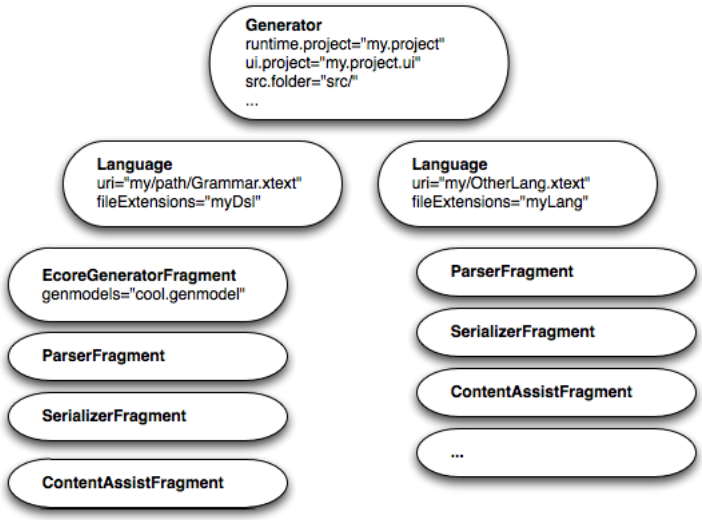
The `addChild` method takes a `foo.Person` as an argument. As this is a concrete class which has a default constructor it can be instantiated.

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 outline view (CTRL+O CTRL+O) 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.*

So 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 reference documentation (available through Eclipse help) for additional information.

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 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.



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.

1.2.2. Configuration

As already explained we use MWE from EMFT in order to build, 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>

   <!-- java-based API for content assistance -->
   <fragment class="org.eclipse.xtext.ui.generator.contentAssist.JavaBasedContentAssistFragment"/
  </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 occurences of ${propertyName} with the given value. Property declarations can be inlined (`<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 file previously.

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-Equinox 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.

1.3. Standard generator fragments

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

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

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 updatesite at* ***http://download.itemis.com/updates/****.*

2. Dependency Injection in Xtext with Google Guice

Xtext uses Google Guice in order to wire up and configure language infrstructures. This allows for very flexible customization of language infrastructure and at the same time makes the different pieces far more testable.

2.1. 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.

2.1.1. 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.

2.1.2. Default Modules

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 (interpreting as opposed to generated). Examples are all the components used in linking, the outline view, hyperlinking and navigation.

2.1.3. 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 this:

```
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 4. 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 lets you avoid any type casts. 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 "EntityWithƒ÷‹"
```

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. Each string in a model which is assigned to an EAttribute with the name ‘importURI’, will be interpreted as a 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 using 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 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 `EReference` (declared or inherited by the object’s `EClass`). The returned `IScope` should contain all target candidates for the given object and cross reference.

```
@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 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](#) .

5.1. Declarative Scope Provider

As always there is an implementation allowing to specify scoping in a declarative way (extend `org.eclipse.xtext.scoping.impl.AbstractDeclarativeScopeProvider` for this purpose). It looks up methods which have either of the following two signatures:

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

```
IScope scope_<TypeToReturn>(<ContextType> ctx, EClass type)
```

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 evaluating the scope of 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 matching the context object. Thus in the example the signature first matched would be:

```
IScope scope_State(Transition this, EClass type)
```

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, EClass type)
```

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

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. Serialization

Serialization is the process of transforming an EMF model to 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 (whitespaces, comments, etc.) into the token stream. This is done by the Hidden TOken Merger.
3. Adding needed whitespaces or replacing all whitespaces using a Formatter.

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

7.1. The Contract

The contract of serialization is that when a model is serialized to its textual representation and then loaded (parsed) again, the loaded model equals the original model. Please be aware that this does *not* imply, that when loading a textual representation and serializing it again that both textual representations equal each other. 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 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.

7.2. Parse Tree Constructor

The Parse Tree Constructor usually doesn't 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 it 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 Ecore model. Example:

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

This example introduces the constraint `sval.size() == ival.size()`. Models which violate this constraint are valid EMF models, but they can not be serialized. To check whether a model complies with all constraints introduced by the grammar, there is currently only the way 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, that

- 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 models 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 is missing in 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 approach. 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 can not tell which dead end is actually erroneous. Therefore, the error message lists dead ends of the longs 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 right branch first and detects wrong branches early. One way to archive this is to avoid having many rules which return the same type and which are called from within the same grammar-alternative.

7.3. 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 their default value.

7.4. Unassigned Text

Unassigned Text are data rule calls or terminal rule calls which do not reside within an association. Example:

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

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

Valid DSL-Scripts 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 a decision which value to write during serialization. This decision can be be made by implementing the `IUnassignedTextSerializer`.

7.5. Cross Reference Serializer

The Cross Reference Serializer specifies which values are to be writting 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.

7.6. 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 (whitespaces, linebreaks, 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.

7.7. Token Stream

The Parse Tree Constructor, the Hidden Token Merger and the Formatter use Token Streams for their output, and the latter two for their input as well. This makes them chainable. Token Streams can be converted to `String` using the `TokenStringBuffer` and to `java.io.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 whitespaces 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;
}
```

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);
}
```

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

Chapter 5. 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

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 of – 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.

2.1. 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 resource 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's 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 a EAttribute that's 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.

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:

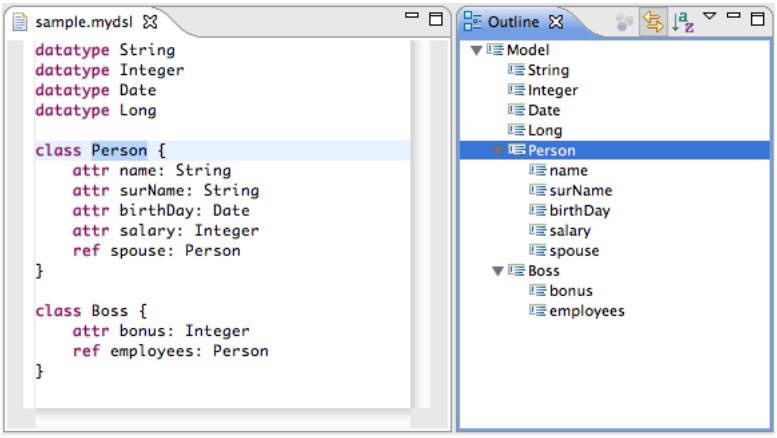
```
<template name="transition" description="event transition"
    id="transition"
    context="org.eclipse.xtext.example.FowlerDsl.Transition"
    enabled="true"
  >${event:CrossReference('Transition.event')} =>
    ${state:CrossReference('Transition.state')</
```

```
  template>
```

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

5. Outline View

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 correpsonding element in the text editor. Users can choose to synchronize the outline with the editor selection by clicking the *Link with Editor* button.



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

5.1. 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 (artificial / 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 meta classes 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 AbstractXtextUiModule {

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

5.2. 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:

```
/**
 * Use this class to register components to be used within the IDE.
 */
public class MyDslUiModule extends org.xtext.example.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):

```
import org.eclipse.jface.viewers.ViewerFilter;
import org.eclipse.xtext.ui.common.editor.outline.XtextContentOutlinePage;
import org.eclipse.xtext.ui.common.editor.outline.actions.AbstractFilterAction;
import org.eclipse.xtext.xtext.ui.Activator;
```

```
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/fltr_foo.gif"));
    setDisabledImageDescriptor(Activator.getImageDescriptor("icons/fltr_foo.gif"));
  }

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

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

}
```

The filtering itself will be performed by `FooOutlineFilter`:

```
import org.eclipse.emf.ecore.EClass;
import org.eclipse.jface.viewers.Viewer;
import org.eclipse.jface.viewers.ViewerFilter;
import org.eclipse.xtext.XtextPackage;
import org.eclipse.xtext.ui.common.editor.outline.ContentOutlineNode;

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;
  }

}
```

5.3. Context menus

(stay tuned)

5.4. Sorting

(stay tuned)

6. Navigation and Hyperlinking

text

7. 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 (whitespaces, linebreaks, 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.

7.1. Declarative Formatter

A declarative formatter can be implemented by sub-classing `AbstractDeclarativeFormatter`, as shown in the following example:

```
public class ExampleFormatter extends AbstractDeclarativeFormatter {

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

7.1.1. General FormattingConfig Settings

- `setAutoLinewrap(int)` defines the amount of characters after which a linebreak 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 whitespaces.
- `setWhitespaceRule(AbstractRule)` defines the grammar rule which is used to match whitespaces. This is needed by the formatter to identify whitespaces and to insert whitespaces. The default is the rule named `WS`.
- `setIndentation(startele, endele)` increases the level of indentation when `startele` is matched and decreases the level when `endele` is matched. The matching of elements happens in the same way as it does for formatting rules.

7.1.2. 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 constructs 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 linebreak 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 this 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 one token in the textual representation can be matched. This are:

- Terminal rules for comments.
- Keywords, Associations, Terminal RuleCalls, Datatype RuleCalls.

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

- `setLinewrap()`: Inserts a linebreak at this position.
- `setLinewrap(int)`: Inserts the specified number of linebreak 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.

Chapter 6. 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.

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.

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 meta 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 (see 'Differences in Validation'). 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 (TODO: ref).

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!

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 a meta model is very convenient and type safe at the same time. In Java one usually has to write instanceof and cast orgies.

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).

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.

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.

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.

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.

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 meta 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:

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 does neither include assignments nor calls any 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 EDatatype as well (see TODO-REF).

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

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;
```

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 (TODO-REF).

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 meta 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.

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.

4.2. Differences in Validation

TMF Xtext still supports implementing validation using Xpand's Check. However it is no longer using the EMF meta model (typesystem) but now uses the so called JavaBeansMetamodel. This means that the types in Check and Xtend correspond to Java types. This requires minor changes (namely the namespaces to be imported are now the qualified name of the generated Java classes). Example: If your Check file looked like this in oAW Xtext :

```
import myMetamodel;
context Type ERROR "foo" : name!=null;
```

it becomes something like the following in TMF Xtext :

```
import my::pack::to::myMetaModel;
context Type ERROR "foo" : name!=null;
```

Where `my::pack::to::myMetaModel` refers to the package the generated EClasses are in. In other words there's a Java class `my.pack.to.myMetaModel.Type`. We changed this in order to allow use of any Java API from within Check and Xtend.

4.3. 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.

4.3.1. 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)

4.3.2. 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.

4.4. 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.

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.

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 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.

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_SOMMENT 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 libraries 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.

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.

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 7. 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 orginally comes from. Therefore Antlr v 2.x didn't get the needed approvement. Eclipse has a strict IP policy, which makes sure that everything provided by Eclipse can be consumed under the termns of the Eclipse Public Licence. 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 updatesite at:

* http://download.itemis.com/updates/milestones

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***

1. What if I don't 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.