Xtext Documentation

May 20, 2011

Contents

1 Overview

1.1 What is Xtext?

No matter if you want to create a small textual domain-specific language (DSL) or you want to implement a full-blown general purpose programming language. With Xtext you can create your very own languages in a snap. Also if you already have an existing language but it lacks decent tool support, you can use Xtext to create a sophisticated Eclipse-based development environment providing editing experience known from modern Java IDEs in a surprisingly short amount of time. We call Xtext a language development framework.

1.2 How Does It Work?

Xtext provides you with a set of domain-specific languages and modern APIs to describe the different aspects of your programming language. Based on that information it gives you a full implementation of that language running on the JVM. The compiler components of your language are independent of Eclipse or OSGi and can be used in any Java environment. They include such things as the parser, the type-safe abstract syntax tree (AST), the serializer and code formatter, the scoping framework and the linking, compiler checks and static analysis aka validation and last but not least a code generator or interpreter. These runtime components integrate with and are based on the Eclipse Modeling Framework (EMF), which effectively allows you to use Xtext together with other EMF frameworks like for instance the Graphical Modeling Project GMF.

In addition to this nice runtime architecture, you will get a full blown Eclipse-IDE specifically tailored for your language. It already provides great default functionality for all aspects and again comes with DSLs and APIs that allow to configure or change the most common things very easily. And if that's not flexible enough there is Guice to replace the default behaviour with your own implementations.

1.3 Xtext is Highly Configurable

Xtext uses the lightweight dependency injection (DI) framework Google Guice to wire up the whole language as well as the IDE infrastructure. A central, external module is used to configure the DI container. As already mentioned, Xtext comes with decent default implementations and DSLs and APIs for the aspect that are common sweet spots for customization. But if you need something completely different, Google Guice gives you the power to exchange every little class in a non-invasive way.

1.4 Who Uses Xtext?

Xtext is used in many different industries. It is used in the field of mobile devices, automotive development, embedded systems or Java enterprise software projects and game development. People use Xtext-based languages to drive code generators that target Java, C, C++, C#, Objective C, Python, or Ruby code. Although the language infrastructure itself runs on the JVM, you can compile Xtext languages to any existing platform. Xtext-based languages are developed for well known Open-Source projects such as Maven, Eclipse B3, the Eclipse Webtools platform or Google's Protocol Buffers and the framework is also widely used in research projects.

1.5 Who is Behind Xtext?

Xtext is a professional Open-Source project. We, the main developers and the project lead, work for itemis, which is a well known consulting company specialized on model-based development. Therefore we are able to work almost full-time on the project. Xtext is an Eclipse.org project. Besides many other advantages this means that you don't have to fear any IP issues, because the Eclipse Foundation has their own lawyers who take care that no intellectual property is violated.

You may ask: Where does the money for Open-Source development come from? Well, we provide professional services around Xtext. Be it training or on-site consulting, be it development of prototypes or implementation of full-blown IDEs for programming languages. We do not only know the framework very well but we are also experts in programming and domain-specific language design. Don't hesitate to get in contact with us (www.itemis.com).

1.6 What is a Domain-Specific Language

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

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

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

There are a couple of well-known examples of DSLs. For instance SQL is actually a DSL which focuses on querying relational databases. Other DSLs are regular expressions or even languages provided by tools like MathLab. Also most XML languages

are actually domain-specific languages. The whole purpose of XML is to allow for easy creation of new languages. Unfortunately, XML uses a fixed concrete syntax, which is very verbose and yet not adapted to be read by humans. Into the bargain, a generic syntax for everything is a compromise.

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

2 The Grammar Language

The grammar language is the corner stone of Xtext. It is a domain-specific language, carefully designed for the description of textual languages. ▬< Updated upstream The main idea is to describe the concrete syntax and how it is mapped to an in-memory representation - the semantic model. This model will be produced by the parser on-the-fly when it consumes an input file.

======= The main idea is to describe the concrete syntax and how it is mapped to an in-memory model created during parsing. ▬> Stashed changes

2.1 A First Example

To get an idea of how it works we'll start by implementing a simple example introduced by Martin Fowler. It's mainly about describing state machines that are 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 a draw 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
  resetting doorOpened D1OP
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
```

What we have are a bunch of declared events, commands, and states. Within states there are references to declared actions. Actions should be executed when entering the state. Furthermore, there are transitions consisting of a reference to an event and a state.

The first thing that you have to do in order to implement this tiny state machine example with Xtext, is to provide a grammar. It could look like this example:

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

generate secrets "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];
```

Martin Fowler uses this example throughout his book Domain Specific Languages to implement external and internal DSLs using different technologies. Note, that none of his implementations is nearly as readable and concise as the description in Xtext's grammar language above. That is of course because the grammar language is designed to do just that, i.e. it is specific to the domain of language descriptions.

2.2 The Syntax

In the following the different concepts and syntactical constructs of the grammar language are explained.

2.2.1 Language Declaration

Each Xtext grammar starts with a header that defines some properties of the grammar.

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

The first line declares the name of the language. Xtext leverages Java's classpath mechanism. This means that the name can be any valid Java qualifier. The file name needs to correspond to the language name and have the file extension .xtext. This means that the name has to be *SecretCompartments.xtext* and must be placed in a package *org.xtext.example* on your project's classpath. In other words, your .xtext file has to reside in a Java source folder to be valid.

The second aspect that can be deduced from the first line of the grammar is its relationship to other languages. An Xtext grammar can declare another existing grammar to be reused. The mechanism is called grammar mixin (§2.4)).

2.2.2 EPackage Declarations

Xtext parsers create in-memory object graphs while consuming text. Such object-graphs are instances of EMF Ecore models. An Ecore model basically consists of an *EPackage* containing *EClasses*, *EDataTypes* and *EEnums* (See the section on EMF (§9.1) for more details) and describes the structure of the instantiated objects. Xtext can infer Ecore models from a grammar (see Ecore model inference (§2.3)) but it is also possible to import existing Ecore models. You can even mix both approaches and use multiple existing Ecore models and infer some others from a single grammar. This allows for easy reuse of existing abstractions while still having the advantage of short turn arounds with derived Ecore models.

EPackage Generation

The easiest way to get started is to let Xtext infer the Ecore model from your grammar. This is what is done in the secret compartment example. The **generate** declaration in the grammar advises the framework to do so:

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

That statement could actually be read as: generate an *EPackage* with the *name* secrets and the *nsURI* "http://www.eclipse.org/secretcompartment". Actually these are the mandatory properties that are necessary to create an empty *EPackage*. Xtext will then add *EClasses* with features (*EAttributes* and *EReferences*) for the different parser rules in your grammar, as described in Ecore model inference (§2.3).

EPackage Import

If you already have an existing *EPackage*, you can import it using either a namespace URI or a resource URI. An URI (Uniform Resource Identifier) provides a simple and extensible means for identifying an abstract or physical resource. For all the nifty details about EMF URIs please refer to its documentation.

Using Resource URIs to Import Existing EPackages In order to import an existing Ecore model, you'll have to have the *.ecore file describing the *EPackage* you want to use somewhere in your workspace. To refer to that file you make use of the *platform:/resource* scheme. Platform URIs are a special EMF concept which allow to reference elements in the workspace independent of the physical location of the workspace. It is an abstraction that uses the Eclipse workspace concept as the logical root of each project.

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

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

If you want to mix generated and imported Ecore models you'll have to configure the generator fragment in your MWE file responsible for generating the Ecore classes (org.eclipse.xtext.generator.ecore.EcoreGeneratorFragment) with resource URIs that point to the generator models (§9.2) of the referenced Ecore models.

The **.genmodel* provides all kind of generator configuration used by EMF's code generator. Xtext will automatically create a generator model for derived *EPackages*, but if it references an existing, imported Ecore model, the code generator needs to know how that code was generated in order to generate valid Java code.

Example:

```
fragment = org.eclipse.xtext.generator.ecore.EcoreGeneratorFragment {
  referencedGenModels =
    "platform:/resource/my.project/model/SecretCompartments.genmodel"
}
```

Using Classpath URIs to Import Existing EPackages We like to leverage Java's classpath mechanism, because is is well understood and can be configured easily with Eclipse.

Furthermore it allows us to package libraries as jars. If you want to reference an existing **.ecore* file which is contained in a jar, you can make use of the classpath URI scheme we've introduced. For instance if you want to reference Java elements, you can use the JvmType Ecore model which is shipped as part of Xtext.

Example:

```
import 'classpath:/model/JvmTypes.ecore'as types
```

As with platform resource URIs you'll also have to tell the generator where the corresponding **.genmodel* can be found:

```
fragment = org.eclipse.xtext.generator.ecore.EcoreGeneratorFragment {
  referencedGenModels =
    "classpath:/model/JvmTypes.genmodel"
}
```

See the section on Referring Java Types (§7) for a full explanation of this useful feature.

Using Namespace URIs to Import Existing EPackages You can also use nsURI in order to import existing *EPackage*. Note that this is generally not preferrable, because you'll have to have the corresponding EPackage installed in the workbench. There's mainly just one exception, that is Ecore itself. So if you refer to Ecore it is best to use its nsURI :

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

Ecore Model Aliases for EPackages

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

```
generate secrets '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::SomeType*). We'll see later where such type references occur.

It is also supported to put multiple *EPackage* imports into one alias. This is no problem as long as there are not any two *EClassifiers* with the same name. In that case none of them can be referenced. It is even possible to *import* multiple and *generate* one Ecore model and declare all of them with 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 be generated into the inferred package.

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.

2.2.3 Rules

Basically parsing can be separated in the following phases.

1. Lexing
2. Parsing
3. Linking
4. Validation

Terminal Rules

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

In the secret compartments example there are no explicitly defined terminal rules, since it only uses the *ID* rule which is inherited from the grammar *org.eclipse.xtext.common.Terminals* (cf. Grammar Mixins (§2.4)). Therein the *ID* rule is defined as follows:

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

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

The caret is used to escape an identifier if there are conflicts with existing keywords. It is removed by the *ID* rule's ValueConverter (§4.7).

This is the simplified 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 shadow each other. This is especially important for newly introduced rules in connection with imported rules from used grammars.

It's almost in any case recommended to use data type rules instead. Let's assume you want to add a rule to allow fully qualified names in addition to simple IDs. Since a qualified name with only one segment looks like a plain ID, you should implement it as a data type rule (§2.2.6), instead of adding another terminal rule. The same rule of thumb applies to floating point literals, too.

Return Types Each terminal rule returns an atomic value (an Ecore EDataType). By default, it's assumed that an instance of *ecore::EString*) should be returned. However, if you want to provide 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 framework how to convert the parsed string to a value of the declared data type, you need to provide your own implementation of org.eclipse.xtext.conversion.IValueConverterService (cf. value converters (§4.7)). The value converter is also the service that allows to remove escape sequences or semantically unnecessary character like quotes from string literals or the caret ('^') from identifiers. Its implementation needs to be registered as a service (cf. Service Framework (§3.2)).

Extended Backus-Naur Form Expressions

Terminal rules are described using *Extended Backus-Naur Form*-like (EBNF) expressions. The different expressions are described in the following. Each of these expressions allows to define a cardinality. There are four different possible cardinalities:

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

Keywords / Characters Keywords are a kind of terminal 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.

The following standard Java notations for special characters are allowed: $\backslash n$, $\backslash r$, $\backslash t$, $\backslash b$, $\backslash f$ and the quoted unicode character notation, such as *\u123*.

Character Ranges A character range can be declared using the .. operator.

Example:

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

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

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

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

The rule above would allow expressions like *foo*, *f0o* or even *f_o*.

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 this way:

terminal ML_COMMENT: '/*'−> '*/';

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

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

terminal BETWEEN_HASHES: '#'(!'#')* '#';

Rule Calls Rules can refer to other rules. This is simply done by using 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 DOUBLE : INT '.'INT;

Note: It is generally not a good idea to implement floating point literals with terminal rules. You should use data type rules instead for the above mentioned reasons.

Alternatives Alternatives allow to define multiple valid options in the input file. For instance, the whitespace rule uses alternatives like this:

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

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

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

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

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

Terminal Fragments

Since terminal rules are used in an unscoped context, it's not easily possible to reuse parts of their definition. Fragments solve this problem. They allow the same EBNF elements as terminal rules do but may not be consumed by the lexer. Instead, they have to be used by other terminal rules. This allows to extract repeating parts of a definition:

```
terminal fragment ESCAPED_CHAR : '\\' ('n'|'t'|'r'|'\\');
terminal STRING :
            '"' ( ESCAPED_CHAR | !('\\'|'"') )* '"' |
            "'" ( ESCAPED_CHAR | !('\\'|"'") )* "'"
;
```

EOF - End Of File

The EOF (End Of File) token may be used to describe that the end of the input stream is a valid situation at a certail point in a terminal rule. This allows to consume the complete remaining input of a file starting with a special delimiter.

terminal UNCLOSED_COMMENT : '/*'(!**EOF**)* **EOF**;

2.2.4 Parser Rules

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

Extended Backus-Naur Form Expressions

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

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

1. Groups (§2.2.3),
2. Alternatives (§2.2.3),
3. Keywords (§2.2.3) and
4. Rule Calls (§2.2.3).

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

Assignments Assignments are used to assign the consumed information to a feature of the currently produced object. The type of the current object, its *EClass*, is specified by the return type of the parser rule. If it is not explicitly stated it is implied that the type's name equals the rule's name. The type of the assigned feature is inferred from the right hand side of the assignment.

Example:

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

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

```
name=ID
```

The left hand side refers to a feature *name* of the current object (which has the *EClass State* in this case). The right hand side can be a rule call, a keyword, a cross-reference (§2.2.4) or an alternative comprised by the former. The type of the feature needs to be compatible with the type of the expression on the right. As *ID* returns an *EString* in this case, the feature *name* needs to be of type *EString* as well.

Assignment Operators

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

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

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

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

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

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

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

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

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

;

Event **returns** MyEvent : ....;

Looking at the syntax definition for cross-references, there is an optional part starting with a vertical bar (pipe) followed by *CrossReferenceableTerminal*. This is the part describing the concrete text, from which the crosslink later should be established. If the terminal is omitted, it is expected to be the rule with the name *ID* - if one can be found. The terminal is mandatory for languages that do not define a rule with the name *ID*.

Have a look at the linking section (§4.5) in order to understand how linking is done.

Unordered Groups The elements of an unordered group can occur in any order but each element must appear once. Unordered groups are separated by &. The following rule Modifier allows to parse simplified modifiers of the Java language:

```
Modifier:
    static?='static'? & final?='final'? & visibility=Visibility;

enum Visibility:
    PUBLIC='public' | PRIVATE='private' | PROTECTED='protected';
```

Therefore, the following sequences of tokens are valid:

```
public static final
static protected
final private static
public
```

However, since no unordered groups are used in the rule Modifier, the parser refuses to accept this input lines:

```
static final static // ERROR: static appears twice
public static final private // ERROR: visibility appears twice
final // ERROR: visibility is missing
```

Note that if you want an element of an unordered group to appear once or not at all, you have to choose a cardinality of ?. In the example, the visibility is mandatory, while **static** or **final** are optional. Elements with a cardinality of $*$ or + have to appear continuously without interruption, i.e.

```
Rule:
    values+=INT* & name=ID;
```

will parse these lines

```
0 8 15 x
x 0 8 15
```

but not does not consume the following sequence without raising an error

```
0 x 8 15 //wrong, as values may be interrupted by a name (ID)
```

Simple Actions The object to be returned by a parser rule is usually created lazily on the first assignment. Its type is determined from the specified return type of the rule which may have been inferred from the rule's name if no explicit return type is specified.

With Actions however, the creation of returned *EObject* can be made explicit. Xtext supports two kinds of Actions:

1. *Simple* Actions, and
2. *Assigned* Actions.

If you want to enforce the creation of an instance with specific type you can use 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*.

If you don't use actions, you'll have to define an alternative and delegate rules to guide the parser to the right types for the to-be-instantiated objects:

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

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

Actions however allow to make this explicit. Thereby they can improve the readability of grammars.

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

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

Note: If a parser rule does not instantiate any object because it does not contain an Action and no mandatory Assignment, you'll likely end up with unexpected situations for valid input files. Xtext detects this situation and will raise a warning for the parser rules in question.

Unassigned Rule Calls We previously explained, that the *EObject* to be returned is created lazily when the first assignment occurs or as soon as a simple action is evaluated. There is another to "find" the *EObject* to be returned. The concept is called *Unassigned Rule Call*.

Unassigned rule calls (the name suggests it) are rule calls to other parser rules, which are not used within an assignment. The return value of the called rule becomes the return value of the calling parser rule if it is not assigned to a feature.

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

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

As *AbstractToken* could possibly return an instance of *TokenA*, *TokenB* or *TokenC* its type must be a super type for all these types. Since the return value of the called rule becomes the result of the current rule, it is possible to further change the state of the AST element by assigning additional features.

Example:

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

This way the *cardinality* is optional (last question mark) and can be represented by a question mark, a plus, or an asterisk. It will be assigned to either an instance of type *TokenA*, *TokenB*, or *TokenC* which are all subtypes of *AbstractToken*. The rule in this example will never create an instance of *AbstractToken* directly but always return the instance that has been created by the invoked *TokenX* rule.

Assigned Actions Xtext leverages the powerful Antlr parser which implements an LL(∗) algorithm. Even though LL parsers have many advantages with respect to readability, debuggability and error recovery, there are also some drawbacks. The most important one is that it does not allow left recursive grammars. For instance, the following rule 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 too difficult. 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.

This problem can be solved by using a combination of unassigned rule calls and assigned actions:

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

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

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

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

2.2.5 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 optionally appear between any other terminals in any cardinality. They can be described per rule or for the whole grammar. When reusing a single grammar (§2.4) its definition of hidden tokens is reused, too. 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 transitively by the declaring rule uses the same hidden terminals as the calling rule, unless it defines hidden tokens itself.

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

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

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

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

A list of all default terminals like *WS* can be found in section Grammar Mixins (§2.4).

2.2.6 Data Type Rules

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

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

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

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

In contrast to a terminal rule this is only valid in certain contexts, i.e. it won't conflict with the rule *ID*. If you had defined it as a terminal rule, it would possibly hide the simple *ID* rule.

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

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

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

Note that rules that do not call other parser rules and do neither contain any actions nor assignments (§2.2.4), are considered to be data type rules and the data type *EString* is implied if none has been explicitly declared.

Value converters (§4.7) are used to transform the parsed string to the actually returned data type value.

2.2.7 Enum Rules

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

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

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

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

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

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

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

with the input of

element Foo

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

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

You can overcome this by modifying the infered Ecore model through a model to model transformation (§2.3.5). However, instead of post processing, an explicitly imported metamodel is recommend.

2.2.8 Syntactic Predicates

It's sometimes not easily possible to define an LL(∗) grammar for a given language that parses all possible valid input files and still produces abstract syntax graphs that mimic the actual structure of the files. There are even cases that cannot be described with an unambiguous grammar. There are solutions that allow to deal with this problem:

- Enable Backtracking: Xtext allows to enable backtracking for the Antlr parser generator. This is usually not recommended since it influences error message strategies at runtime and shadows actually existing problems in the grammar.
- Syntactic Predicates: The grammar language enables users to guide the parser in case of ambiguities. This mechanism is achieved by syntactic predicates. Since they affect only a very small part of the grammar, syntactic predicates are the recommended approach to handle Antlr error messages during the parser generation.

The classical example for ambiguous language parts is the *Dangling Else Problem*. A conditional in a programming language usually looks like this:

```
if (isTrue())
    doStuff();
else
    dontDoStuff();
```

The problems becomes more obvious as soon as nested conditions are used:

```
if (isTrue())
    if (isTrueAsWell())
        doStuff();
    else
        dontDoStuff();
```

Where does the else branch belong to? This question can be answered by a quick look into the language specication which tells that the else branch is part of the inner condition. However, the parser generator cannot be convinced that easy. We have to guide it to this decision point by means of syntactic predicates which are expressed by a leading => operator.

```
Condition:
    'if' condition=Expression
        'then' then=Expression
        (=>'else' else=Expression)?
```

The parser understands the predicate basically like this: If you are at this particular decision point and you don't know what to do, look for the **else** keyword and if it's present. Don't try to choose the other option that would start with an **else** keyword, too.

Well choosen predicates allow to solve most ambiguities and backtracking can often be disabled.

2.3 Ecore Model Inference

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

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

2.3.1 Type and Package Generation

Xtext creates

- an *EPackage*
  - for each generate-package declaration. After the directive *generate* a list of parameters follows. The *name* of the *EPackage* will be set to the first parameter, its *nsURI* to the second parameter. An optional alias as the third parameter allows to distinguish generated *EPackages* later. Only one generated package declaration per alias is allowed.
- an *EClass*
  - for each return type of a parser rule. If a parser rule does not define a return type, an implicit one with the same name as the rule itself is assumed. You can specify more than one rule that return the same type but only one *EClass* will be generated.
  - for each type defined in an action or a cross-reference.
- an *EEnum*
  - for each return type of an enum rule.
- an *EDataType*
  - for each return type of a terminal rule or a data type rule.

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

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

While iterating the parser rules Xtext creates

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

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

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

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

2.3.3 Enum Literal Generation

For each alternative defined in an enum rule, the transformer creates an enum literal, as long as no other literal with the same name can be found. The *literal* property of the generated enum literal is set to the right hand side of the declaration. If it is omitted, an enum literal with equal *name* and *literal* attributes is inferred.

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

2.3.4 Feature Normalization

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

2.3.5 Customized Post Processing

As a last step, the generator invokes the post processor for every generated Ecore model. The post processor expects an Xtend1 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) :
  ... do something
;
```

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

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

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

Note: If you face the situation where you think that post processing may help, you should strongly consider to use imported packages instead of generated packages.

2.3.6 Error Conditions

The following conditions cause an error

- An *EAttribute* or *EReference* has two different types or different cardinality.

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

2.4 Grammar Mixins

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

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

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

Statemachine: ..
```

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

Example :

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

grammar my.SubGrammar with my.SuperGrammar

Model : (ruleAs+=RuleA)*;

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

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

2.5 Common Terminals

Xtext ships with a default set of predefined, reasonable and often required terminal rules. The grammar for these common terminal rules 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'|'u'|""|"'"|'\\') | !('\\'|"") )* "" |
 "'" ( '\\'('b'|'t'|'n'|'f'|'r'|'u'|""|"'"|'\\') | !('\\'|"'") )* "'";
terminal ML_COMMENT :
 '/*' -> '*/';
terminal SL_COMMENT :
 '//' !('\n'|'\r')* ('\r'? '\n')?;
terminal WS :
 (' '|'\t'|'\r'|'\n')+;
terminal ANY_OTHER:
 .;
```

3 Configuration

3.1 The Language Generator

Xtext provides a lot of generic implementations for your language's infrastructure but also uses code generation to generate some of the components. Those generated components are for instance the parser, the serializer, the inferred Ecore model (if any) 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 (§3.2.1).

Xtext's generator uses a special DSL called MWE2 - the modeling workflow engine (§5) to configure the generator.

3.1.1 A Short Introduction to MWE2

MWE2 allows to compose object graphs declaratively in a very compact manner. The nice thing about it is that it just instantiates Java classes and the configuration is done through public setter and adder methods as one is used to from Java Beans encapsulation principles. An in-depth documentation can be found in the chapter MWE2 (§5).

Given the following simple Java class (POJO):

```
package com.mycompany;

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 with MWE2 easily by describing it in a declarative manner without writing a single line of Java code and without the need to compile classes:

```
module com.mycompany.CreatePersons

Person {
```

```
    name = "Grandpa"
    child = {
        name = "Father"
        child = {
            name = "Son"
        }
    }
}
```

These couple of lines will, when interpreted by MWE2, result in an object tree consisting of three instances of *com.mycompany.Person*. The interpreter will basically do the same as the following *main* method:

```
package com.mycompany;

public class CreatePersons {
    public static void main(String[] args) {
        Person grandpa = new Person();
        grandpa.setName("Grandpa");
        Person father = new Person();
        father.setName("Father");
        grandpa.addChild(father);
        Person son = new Person();
        son.setName("Son");
        father.addChild(son);
    }
}
```

fix image

!{width:50%}images/family_tree.png!

And this is how it works: The root element is a plain Java class name. As the module is a sibling to the class *com.mycompany.Person* it is not necessary to use use fully qualified name. There are other packages implicitly imported into this workflow as well to make it convenient to instantiate actual workflows and components, but these ones are covered in depth in the appropriate chapter (§5). The constructed objects are furthermore configured according to the declaration in the module, e.g. a second instance of Person will be created and added to the list of children of "Grandpa" while the third person - the class is inferred from the assigned feature - becomes a child of "Father". All three instances will have their respective *name* assigned via a reflective invocation of the *setName* method. If one wants to add another child to "Father", she can simply repeat the child assignment:

```
child = com.mycompany.Person {
    name = "Father"
    child = {
        name = "Son"
    }
    child = {
        name = "Daughter"
```

```
    }
}
```

As you can see in the example above MWE2 can be used to instantiate arbitrary Java object models without any dependency or limitation to MWE2 specific implementations.

*Tip Whenever you are in an *.mwe2 file and wonder what kind of configuration the underlying component may accept: Just use the Content Assist in the MWE2 Editor or navigate directly to the declaration of the underlying Java implementation by means of F3 (Go To Declaration).*

This is the basic idea of the MWE2 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 refer to the dedicated MWE2 reference documentation (§5) for additional information. We will now have a look at the component model used to configure the Language Generator.

3.1.2 General Architecture

A language generator is composed of so called 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 org.eclipse.xtext.generator.IGeneratorFragments. The whole generator is composed of theses fragments. We have fragments for generating parsers, the serializer, the EMF code, the outline view, etc.

fix image

Generator Fragments

The list of grammar fragments forms a chain of responsibility, that is they each get the chance to contribute to the generation of langauge infrastructure components and are called in the declared order. Each fragment gets the grammar of the language as an EMF model passed in and is able to generate code in one of the configured locations and contribute to several shared artifacts. A generator fragment must implement the interface org.eclipse.xtext.generator.IGeneratorFragment.

There is usually no need to write your own generator fragments and only rarely you might want to extend an existing one.

Configuration

As already explained we use MWE2 from EMFT in order to instantiate, configure and execute this structure of components. In the following we see an exemplary language generator configuration written in MWE2 configuration code:

```
module org.xtext.example.MyDsl

import org.eclipse.emf.mwe.utils.*
import org.eclipse.xtext.generator.*
import org.eclipse.xtext.ui.generator.*
```

```
var grammarURI = "classpath:/org/xtext/example/MyDsl.xtext"
var file.extensions = "mydsl"
var projectName = "org.xtext.example.mydsl"
var runtimeProject = "../${projectName}"

Workflow {
    bean = StandaloneSetup {
        platformUri = "${runtimeProject}/.."
    }

    component = DirectoryCleaner {
        directory = "${runtimeProject}/src-gen"
    }

    component = DirectoryCleaner {
        directory = "${runtimeProject}.ui/src-gen"
    }

    component = Generator {
        pathRtProject = runtimeProject
        pathUiProject = "${runtimeProject}.ui"
        projectNameRt = projectName
        projectNameUi = "${projectName}.ui"

        language = {
            uri = grammarURI
            fileExtensions = file.extensions

            // Java API to access grammar elements
            fragment = grammarAccess.GrammarAccessFragment {}

            /* more fragments to configure the language */
            ...
        }
    }
}
```

Here the root element is *Workflow* and is part of the very slim runtime model shipped with MWE2. It accepts *beans* and *components*. A *var* declaration is a first class concept of MWE2's configuration language and defines a variable whic can be reset from outside, i.e. when calling the module. It allows to externalize some common configuration parameters. Note that you can refer to the variables using the *${variable-name}* notation.

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

Following the *bean* assignment there are three *component* elements. The *Workflow.addComponent()* method accepts instances of *IWorkflowComponent*, which is the primary concept of MWE2's workflow model. The language generator component itself is an instance of *IWorkflowComponent* and can therefore be used within MWE2 workflows.

3.1.3 Standard Generator Fragments

In the following table the most important standard generator fragments are listed. Please refer to the Javadocs for more detailed documentation. Also have a look at what the Xtext wizard provides and how the workflow configuration in the various example languages look like.

_.Class	_.Generated Artifacts	_.Related Documentation
org.eclipse.xtext.generator.ecore.EcoreGeneratorFragment	EMF code for generated models	Model inference (§2.3)
org.eclipse.xtext.generator.parser.antlr.XtextAntlrGeneratorFragment	ANTLR grammar, parser, lexer and related services	
org.eclipse.xtext.generator.grammarAccess.GrammarAccessFragment	Access to the grammar elements	
org.eclipse.xtext.generator.resourceFactory.ResourceFactoryFragment	EMF Resource Factory	Xtext Resource (§9.3)
org.eclipse.xtext.generator.parseTreeConstructor.ParseTreeConstructorFragment	Model to Parse Tree Constructor	Serialization (§4.8)
org.eclipse.xtext.generator.scoping.ImportNamespacesScopingFragment	Index-based namespace scoping	Index-based namespace scoping (§4.6.1)
org.eclipse.xtext.generator.validation.JavaValidatorFragment	Java Validator	Model validation (§4.4.2)
org.eclipse.xtext.generator.formatting.FormatterFragment	Code formatter	Declarative formatter (§4.9.1)
org.eclipse.xtext.ui.generator.labeling.LabelProviderFragment	Label provider	Label provider (§6.1)
org.eclipse.xtext.ui.generator.outline.OutlineTreeProviderFragment	Outline tree provider configuration	Outline (§6.5)
org.eclipse.xtext.ui.generator.contentAssist.JavaBasedContentAssistFragment	Java based Content Assist	Content assist (§6.2)
org.eclipse.xtext.generator.parser.antlr.ContentAssist.ContentAssistFragment	Content Assist based on ANTLR	Content assist (§6.2)
org.eclipse.xtext.ui.generator.projectWizard.SimpleProjectWizardFragment	Project Wizard	Project wizard (§6.8)

3.2 Dependency Injection in Xtext with Google Guice

All Xtext components are assembled by means of Dependency Injection (DI). This means basically that whenever some code is in need for functionality (or state) from another component, one just declares the dependency rather then stating how to resolve it, i.e. obtaining that component.

For instance when some code wants to use a scope provider, it just declares a field (or method or constructor) and adds the @Inject annotation:

```
public class MyLanguageLinker extends Linker {

  @Inject
  private IScopeProvider scopeProvider;
```

}

It is not the duty of the code to care about where the *IScopeProvider* comes from or how it is created. When above's class is instantiated, Guice sees that it requires an instance of IScopeProvider and assigns it to the specified field or method parameter. This of course only works, if the object itself is created by Guice. In Xtext almost every instance is created that way and therefore the whole dependency net is controlled and configured by the means of Google Guice.

Guice of course needs to know how to instantiate real objects for declared dependencies. This is done in so called *Modules*. A *Module* defines a set of mappings from types to either existing instances, instance providers or concrete classes. Modules are implemented in Java. Here's an example:

```
public class MyDslRuntimeModule extends AbstractMyDslRuntimeModule {
    @Override
    public void configure(Binder binder) {
        super.configure(binder);
        binder.bind(IScopeProvider.class).to(MyConcreteScopeProvider.class);
    }
}
```

With plain Guice modules one implements a method called configure and gets a so called *Binder* passed in. That binder provides a fluent API to define the mentioned mappings. This was just a very brief and simplified description. We highly recommend to have a look at the website Google Guice to learn more.

3.2.1 The Module API

Xtext comes with a slightly enhanced module API. For your language you get two different modules: One for the runtime bundle which is used when executing your language infrastructure outside of Eclipse such as on the build server. The other is located in the UI bundle and adds or overrides bindings when Xtext is used within an Eclipse environment.

The enhancement we added to Guice's Module API is that we provide an abstract base class, which reflectively looks for certain methods in order to find declared bindings. The most common kind of method is :

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

which would do the same as the code snippet above. It simply declares a binding from *IScopeProvider* to *MyConcreteScopeProvider*. That binding will make Guice instantiate and inject a new instance of *MyConcreteScopeProvider* whenever a dependency to *IScopeProvider* is declared.

Having a method per binding allows to deactivate individual bindings by overriding the corresponding methods and either change the binding by returning a different target type or removing that binding completely by returning null.

There are two additional kinds of binding-methods supported. The first one allows to configure a provider. A *Provider* is an interface with just one method :

```
public interface Provider<T> {

  /**
   * Provides an instance of {@code T}. Must never return {@code null}.
   */
  T get();
}
```

This one can be used if you need a hook whenever an instance of a certain type is created. For instance if you want to provide lazy access to a singleton or you need to do some computation each time an instance is created (i.e. factory). If you want to point to a provider rather than to a concrete class you can use the following binding method.

```
public Class<? extends Provider<IScopeProvider>> provideIScopeProvider() {
    return MyConcreteScopeProviderFactory.class;
}
```

(Please forgive us the overuse of the term provider. *The IScopeProvider is not a Guice* provider *.)*

That binding tells Guice to instantiate *MyConcreteScopeProviderFactory* and invoke get() in order to obtain an instance of *IScopeProvider* for clients having declared a dependency to that type. Both mentioned methods are allowed to return an instance instead of a type. This may be useful if some global state should be shared in the application:

```
public Provider<IScopeProvider> provideIScopeProvider() {
    return new MyConcreteScopeProviderFactory();
}
```

or

```
public IScopeProvider bindIScopeProvider() {
    return new MyConcreteScopeProvider();
}
```

respectively.

The last binding method provided by Xtext allows to do anything you can do with Guice's binding API, since it allows you to use it directly. If your method's name starts with the name 'configure', has a return type *void* and accepts one argument of type *Binder*

```
public void configureIScopeProvider(Binder binder) {
    binder.bind(IScopeProvider.class).to(MyConcreteScopeProvider.class);
}
```

3.2.2 Obtaining an Injector

In every application wired up with Guice there is usually one point where you initialize a so called *Injector* using the modules declared and after that using that injector to create

the root instance of the whole application. In plain Java environments this is something that's done in the main method. It could look like this:

```
public static void main(String[] args) {
    Injector injector = Guice.createInjector(new MyDslRuntimeModule());
    MyApplication application = injector.getInstance(MyApplication.class);
    application.run();
}
```

Xtext uses EMF which makes use of a couple of global registries, which have to be configured on startup. Because we of course want to leverage Guice also for all factories, etc. that we put into those registries, we have introduced a so called *ISetup* which provides a method called *Injector createInjectorAndDoEMFRegistration()*. So instead of using the plain Guice code shown above you rather use the *ISetup* class generated for your language, which, as the method name suggests, creates an *Injector* and uses it to initialize a couple of EMF objects and register them in the corresponding registries.

```
Injector injector =
    new MyStandaloneSetup().createInjectorAndDoEMFRegistration();
```

These are the basic ideas around Guice and the small extension Xtext provides on top. For more information we strongly encourage you to read through the documentation on the website of Google Guice

4 Runtime Concepts

Xtext itself and every language infrastructure developed with Xtext is configured and wired-up using dependency injection (§3.2). Xtext may be used in different environments which introduce different constraints. Especially important is the difference between OSGi managed containers and plain vanilla Java programs. To honor these differences Xtext uses the concept of org.eclipse.xtext.ISetup-implementations in normal mode and uses Eclipse's extension mechanism when it should be configured in an OSGi environment.

4.1 Runtime Setup (ISetup)

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

The setup method returns an *Injector*, which can further be used to obtain a parser, etc. It also registers the org.eclipse.emf.ecore.resource.Resource\$Factory and generated org.eclipse.emf.ecore.EPackag to the respective global registries provided by EMF. So basically after having run the setup and you can start using EMF API to load and store models of your language.

4.2 Setup within Eclipse-Equinox (OSGi)

Within Eclipse we have a generated *Activator*, which creates a Guice com.google.inject.Injector using the modules (§3.2.1). In addition an org.eclipse.core.runtime.IExecutableExtensionFactory is generated for each language, which is used to create org.eclipse.core.runtime.IExecutableExtensions. 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.editor.XtextEditor"
    contributorClass=
      "org.eclipse.ui.editors.text.TextEditorActionContributor"
    default="true"
    extensions="ecoredsl"
    id="org.eclipse.xtext.example.EcoreDsl"
```

```
    name="EcoreDsl Editor">
  </editor>
</extension>
```

4.3 Logging

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

In OSGi you provide configuration by creating a fragment for *org.apache.log4j*. In this case you need to make sure that there is not any second fragment contributing a *log4j.properties* file.

4.4 Validation

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

4.4.1 Automatic Validation

Some implementation aspects (e.g. the grammar, scoping) of a language have an impact on what is required for a document or semantic model to be valid. Xtext automatically takes care of this.

Lexer/Parser: 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. One can use the org.eclipse.xtext.parser.antlr.ISyntaxErrorMessageProvider-API to customize this messages. Any syntax errors can be retrieved from the Resource using the common EMF API:

revise indentation levels

- _org.eclipse.emf.ecore.resource.Resource.getErrors()_
- _org.eclipse.emf.ecore.resource.Resource.getWarnings()_

Linker: Crosslink Validation

Any broken crosslinks can be checked generically. As crosslink resolution is done lazily (see linking (§4.5)), any broken links are resolved lazily as well. If you want to validate whether all links are valid, you will have to navigate through the model so that all installed EMF proxies get resolved. This is done automatically in the editor.

Similar to syntax errors, any unresolvable crosslinks will be reported and can be obtained through:

revise indentation levels

- _org.eclipse.emf.ecore.resource.Resource.getErrors()_
- _org.eclipse.emf.ecore.resource.Resource.getWarnings()_

Serializer: Concrete Syntax Validation

The org.eclipse.xtext.validation.IConcreteSyntaxValidator validates all constraints that are implied by a grammar. Meeting these constraints for a model is mandatory to be serialized.

Example:

```
MyRule:
  ({MySubRule} "sub")? (strVal+=ID intVal+=INT)*;
```

This implies several constraints:

revise indentation levels

1. Types: only instances of *MyRule* and *MySubRule* are allowed for this rule. Subtypes are prohibited, since the parser never instantiates unknown sub-types.
2. Features: In case the *MyRule* and *MySubRule* have *EStructuralFeatures* besides *strVal* and *intVal*, only *strVal* and *intVal* may have non-transient values (§4.8.6).
3. Quantities: The following condition must be true: *strVal.size() == intVal.size()*.
4. Values: It must be possible to convert all values (§4.7) to valid tokens for terminal rule *STRING*. The same is true for *intVal* and *INT*.

The typical use cases for the concrete syntax validator are validation in non-Xtext-editors that, however, use an *XtextResource*. This is, for example, the case when combining GMF and Xtext. Another use case is when the semantic model is modified "manually" (not by the parser) and then serialized again. Since it is very difficult for the serializer to provide meaningful error messages (§4.8.3), the concrete syntax validator is executed by default before serialization. A textual Xtext editor itself, however, is *not* a valid use case. Here, the parser ensures that all syntactical constraints are met. Therefore, there is no value in additionally running the concrete syntax validator.

There are some limitations to the concrete syntax validator which result from the fact that it treats the grammar as declarative, which is something the parser doesn't always do.

revise indentation levels

- Grammar rules containing assigned actions (e.g. {*MyType.myFeature=current*} are ignored. Unassigned actions (e.g. {*MyType*}), however, are supported.
- Grammar rules that delegate to one or more rules containing assigned actions via unassigned rule calls are ignored.

- Orders within list-features can not be validated. e.g. *Rule: (foo+=R1 foo+=R2)** implies that *foo* is expected to contain instances of *R1* and *R2* in an alternating order.

To use concrete syntax validation you can let Guice inject an instance of org.eclipse.xtext.validation.IConcreteSy and use it directly. Furthermore, there is an adapter (org.eclipse.xtext.validation.impl.ConcreteSyntaxEValidator) allows integrating of the concrete syntax validator as an *EValidator*. You can, for example, enable it in your runtime module, by adding:

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

To customize error messages please see org.eclipse.xtext.validation.IConcreteSyntaxDiagnosticProvider and subclass org.eclipse.xtext.validation.impl.ConcreteSyntaxDiagnosticProvider.

4.4.2 Custom Validation

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

For Xtext we provide a generator fragment (§3.1.2) for the convenient Java-based *EValidator* API. Just add the following fragment to your generator configuration and you are good to go:

```
fragment = org.eclipse.xtext.generator.validation.JavaValidatorFragment {}
```

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

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

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

```
public class DomainmodelJavaValidator
  extends AbstractDomainmodelJavaValidator {
```

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

You can also implement quick fixes for individual validation errors and warnings. See the chapter on quick fixes (§6.3) for details.

4.4.3 Validation with the Check Language

In addition to the Java-based validation code you can use the language Check (from M2T/Xpand) to implement constraint checks against your model. To do so, you have to configure the generator (§3.1) with the

fixRef org.eclipse.xtext.generator.validation.CheckFragment

. Please note, that you can combine both types of validation in your project.

```
fragment = org.eclipse.xtext.generator.validation.CheckFragment {}
```

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

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

Each check works in a specific context (here: *Type*) and can further denote a feature to which a warning or error should be attached to (here: *name*). Each check could either be a *WARNING* or an *ERROR* with a given string to explain the situation. The essential part of each check is an invariant that must hold true for the given context. If it fails the check will produce an issue with the provided explanation.

Please read more about the Check language as well as the underlying expression language in Xpand's reference documentation which is shipped as Eclipse help.

4.4.4 Validating Manually

As noted above, Xtext uses EMF's *EValidator* API to register Java or Check validators. You can run the validators on your model programmatically using EMF's *Diagnostician*, e.g.

```
EObject myModel = myResource.getContents().get(0);
Diagnostic diagnostic = Diagnostician.INSTANCE.validate(myModel);
switch (diagnostic.getSeverity()) {
  case Diagnostic.ERROR:
    System.err.println("Model␣has␣errors:␣",diagnostic);
```

```
    break;
  case Diagnostic.WARNING:
    System.err.println("Model has warnings: ",diagnostic);
}
```

4.4.5 Test Validators

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

Testing validators typically works as follows:

revise indentation levels

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

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

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

Validator:

```
public class MyLanguageValidator extends AbstractDeclarativeValidator {
  @Check
  public void checkFooElement(FooElement element) {
    if(element.getBarAttribute().contains("foo"))
      error("Only Foos allowed", element,
        MyLanguagePackage.FOO_ELEMENT__BAR_ATTRIBUTE, 101);
  }
}
```

JUnit-Test:

```
public class MyLanguageValidatorTest extends AbstractXtextTests {

  private ValidatorTester<MyLanguageValidator> tester;

  @Override
  public void setUp() {
    with(MyLanguageStandaloneSetup.class);
    MyLanguageValidator validator = get(MyLanguageValidator.class);
    tester = new ValidatorTester<TestingValidator>(validator);
```

```
  }

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

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

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

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

This example uses JUnit 3, but since the involved classes from Xtext have no dependency on JUnit whatsoever, JUnit 4 and other testing frameworks will work as well. JUnit runs the *setUp()*-method before each testcase and thereby helps to create some common state. In this example, the validator (_MyLanguageValidator_) is instantiated by means of Google Guice. As we inherit from the *AbstractXtextTests* there are a plenty of useful methods available and the state of the global EMF singletons will be restored in the *tearDown()*. Afterwards, the *ValidatorTester* is created and parameterized with the actual validator. It acts as a wrapper for the validator, ensures that the validator has a valid state and provides convenient access to the validator itself (_tester.validator()_) as well as to the utility classes which assert diagnostics created by the validator (_tester.diagnose()_). Please be aware that you have to call *validator()* before you can call *diagnose()*. However, you can call *validator()* multiple times in a row.

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

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

revise indentation levels

- _assertError(int code)_: There must be one diagnostic with severity ERROR and the supplied error code.
- _assertErrorContains(String messageFragment)_: There must be one diagnostic with severity ERROR and its message must contain *messageFragment*.
- _assertError(int code, String messageFragment)_: Verifies severity, error code and messageFragment.

- _assertWarning(...)_: This method is available for the same combination of parameters as *assertError()*.
- _assertOK()_: Expects that no diagnostics (errors, warnings etc.) have been raised.
- _assertDiagnostics(int severity, int code, String messageFragment)_: Verifies severity, error code and messageFragment.
- _assertAll(DiagnosticPredicate... predicates)_: Allows to describe multiple diagnostics at the same time and verifies that all of them are present. Class org.eclipse.xtext.junit.validation.AssertableDiagnostics contains static *error()* and *warning()*-methods which help to create the needed *DiagnosticPredicate*. Example: *assertAll(error(123), warning("some part of the message"))*.
- _assertAny(DiagnosticPredicate predicate)_: Asserts that a diagnostic exists which matches the predicate.

4.5 Linking

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

revise indentation levels

1. declaration of a crosslink in the grammar (at least in the Ecore model)
2. specification of linking semantics (usually provided via the scoping API (§4.6))

4.5.1 Declaration of Crosslinks

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

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

Example:

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

The Ecore model inference (§2.3) would create an *EClass ReferringType* with an *EReference referencedObject* of type *Entity* (_containment=false_). The referenced object would be identified either by a *STRING* and the surrounding information in the current context (see scoping (§4.6)). If you do not use *generate* but *import* an existing Ecore model, the class *ReferringType* (or one of its super types) would need to have an *EReference* of type *Entity* (or one of its super types) declared. Also the *EReference*'s containment and container properties needs to be set to *false*.

4.5.2 Default Runtime Behavior (Lazy Linking)

Xtext uses lazy linking by default and we encourage users to stick to this because it provides many advantages. One of which is improved performance in all scenarios where you don't have to load the whole closure of all transitively referenced resources. Furthermore it automatically solves situations where one link relies on other links. Though cyclic linking dependencies are not supported by Xtext at all.

When parsing a given input string, say

ref Entity01

the org.eclipse.xtext.linking.lazy.LazyLinker first creates an EMF proxy and assigns it to the corresponding *EReference*. In EMF a proxy is described by a URI, which points to the real *EObject*. In the case of lazy linking the stored URI comprises of the context information given at parse time, which is the *EObject* containing the cross-reference, the actual *EReference*, the index (in case it's a multi-valued cross-reference) and the string which represented the crosslink in the concrete syntax. The latter usually corresponds to the name of the referenced *EObject*. In EMF a URI consists of information about the resource the *EObject* is contained in as well as a so called fragment part, which is used to find the *EObject* within that resource. When an EMF proxy is resolved, the current *ResourceSet* is asked. The resource set uses the first part to obtain (i.e. load if it is not already loaded) the resource. Then the resource is asked to return the *EObject* based on the fragment in the URI. The actual cross-reference resolution is done by *LazyLinkingResource.getEObject(String)* which receives the fragment and delegates to the implementation of the *ILinkingService*. The default implementation in turn delegates to the scoping API (§4.6).

A simple implementation of the linking service (the org.eclipse.xtext.linking.impl.DefaultLinkingService) is shipped with Xtext and used for any grammar per default. Usually any necessary customization of the linking behavior can best be described using the scoping API (§4.6).

4.6 Scoping

Using the scoping API one defines which elements are referable by a certain reference. For instance, using the introductory example (fowler's state machine language) a transition contains two cross-references: One to a declared event and one to a declared state.

Example:

```
events
  nothingImportant MYEV
end

state idle
  nothingImportant => idle
end
```

The grammar rule for transitions looks like this:

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

The grammar states that for the reference *event* only instances of the type *Event* are allowed and that for the EReference *state* only instances of type *State* can be referenced. However, this simple declaration doesn't say anything about where to find the states or events. That is the duty of scopes.

An org.eclipse.xtext.scoping.IScopeProvider is responsible for providing an org.eclipse.xtext.scoping.IScope for a given context *EObject* and *EReference*. The returned *IScope* should contain all target candidates for the given object and cross-reference.

```
public interface IScopeProvider {

    /**
     * Returns a scope for the given context. The scope
     * provides access to the compatible visible EObjects
     * for a given reference.
     *
     * @param context the element from which 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. Note for implementors: The
     * result may not be <code>null</code>. Return
     * <code>IScope.NULLSCOPE</code> instead.
     */
    IScope getScope(EObject context, EReference reference);

}
```

A single *IScope* represents an element of a linked list of scopes. That means that a scope can be nested within an outer scope. Each scope works like a symbol table or a map where the keys are strings and the values are so called org.eclipse.xtext.resource.IEObjectDescription, which is effectively an abstract description of a real *EObject*.

4.6.1 Global Scopes and IResourceDescriptions

In the state machine example we don't have references across model files. Also there is no concept like a namespace which would make scoping a bit more complicated. Basically, every *State* and every *Event* declared in the same resource is visible by their name. However in the real world things are most likely not that simple: What if you want to reuse certain declared states and events across different state machines and you want to share those as library between different users? You would want to introduce some kind of cross resource reference.

Defining what is visible from outside the current resource is the responsibility of global scopes. As the name suggests, global scopes are provided by instances of the org.eclipse.xtext.scoping.IGlobalScopeProvider. The data structures used to store its elements are described in the next section.

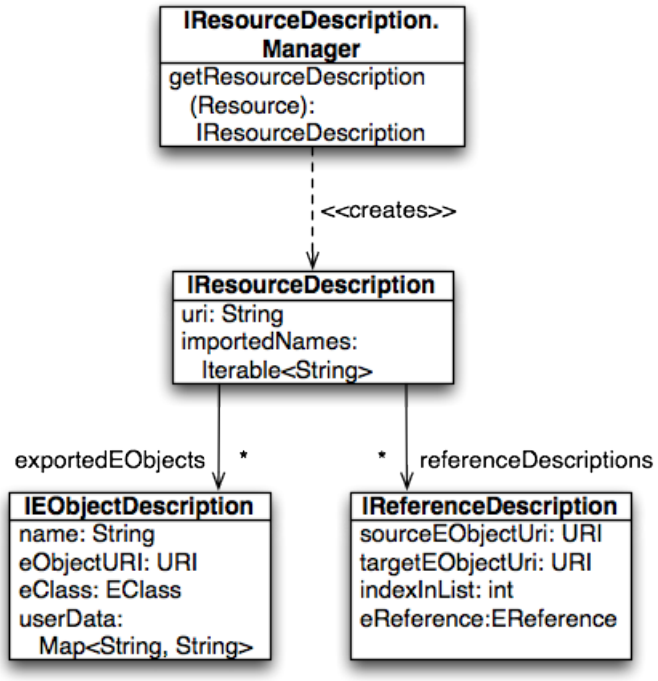
Resource and EObject Descriptions (_IResourceDescription_, IEObjectDescription)

In order to make states and events of one file referable from another file you need to export them as part of a so called org.eclipse.xtext.resource.IResourceDescription.

A *IResourceDescription* contains information about the resource itself (primarily its *URI*), a list of exported *EObjects* (in the form of org.eclipse.xtext.resource.IEObjectDescription) as well as information about outgoing cross-references and qualified names it references. The cross references contain only resolved references, while the list of imported qualified names also contain those names, which couldn't be resolved. This information is important in order to compute the transitive hull of dependent resources, which the shipped index infrastructure automatically does for you.

For users and especially in the context of scoping the most important information is the list of exported *EObjects*. An org.eclipse.xtext.resource.IEObjectDescription contains information about the *URI* to the actual *EObject* and the qualified name of that element as well as the corresponding *EClass*. In addition one can export arbitrary information using the *user data* map. The following diagram gives an overview on the description classes and their relationships.

fix image



A language is configured with a default implementation of *IResourceDescription.Manager* which computes the list of exported *IEObjectDescriptions* by iterating the whole EMF model and applying the *getQualifiedName(EObject obj)* from org.eclipse.xtext.naming.IQualifiedNameProvider on each *EObject*. If the object has a qualified name an *IEObjectDescription* is created and exported (i.e. added to the list). If an *EObject* doesn't have a qualified name, the

element is considered to be not referable from outside the resource and consequently not indexed. If you don't like this behavior, you can implement and bind your own implementation of *IResourceDescription.Manager*.

There are also two different default implementations of *IQualifiedNameProvider*. Both work by looking up an *EAttribute* 'name'. The org.eclipse.xtext.naming.SimpleNameProvider simply returns the plain value, while the org.eclipse.xtext.naming.DefaultDeclarativeQualifiedNameProvider concatenates the simple name with the qualified name of its parent exported *EObject*. This effectively simulates the qualified name computation of most namespace-based languages (like e.g. Java).

As mentioned above, in order to calculate an *IResourceDescription* for a resource the framework asks the *IResourceDescription.Manager*. Here's some Java code showing how to do that:

```
Manager manager = // obtain an instance of IResourceDescription.Manager
IResourceDescription description = manager.getResourceDescription(resource);
for (IEObjectDescription objDescription : description.getExportedObjects()) {
  System.out.println(objDescription.getQualifiedName());
}
```

In order to obtain an *IResourceDescription.Manager* it is best to ask the corresponding org.eclipse.xtext.resource.IResourceServiceProvider. That is because each language might have a totally different implementation and as you might refer from your language to a different language you can't reuse your language's *IResourceDescription.Manager*. One basically asks the *IResourceServiceProvider.Registry* (there is usually one global instance) for an *IResourceServiceProvider*, which in turn provides an *IResourceDescription.Manager* along other useful services.

If you're running in a Guice enabled scenario, the code looks like this:

```
@Inject
private IResourceServiceProvider.Registry resourceServiceProviderRegistry;

private IResourceDescription.Manager getManager(Resource res) {
  IResourceServiceProvider resourceServiceProvider =
    resourceServiceProviderRegistry.getResourceServiceProvider(res.getURI());
  return resourceServiceProvider.getResourceDescriptionManager();
}
```

If you don't run in a Guice enabled context you will likely have to directly access the singleton:

```
private IResourceServiceProvider.Registry resourceServiceProviderRegistry =
  IResourceServiceProvider.Registry.INSTANCE;
```

However, we strongly encourage you to use dependency injection. Now, that we know how to export elements to be referenceable from other resources, we need to learn how those exported *IEObjectDescriptions* can be made available to the referencing resources. That is the responsibility of global scoping (i.e. org.eclipse.xtext.scoping.IGlobalScopeProvider) which is described in the following chapter.

Global Scopes Based On Explicit Imports (ImportURI Mechanism)

A simple and straight forward solution is to have explicit references to other resources in your file by explicitly listing pathes (or *URIs*) to all referenced resources in your model file. That is for instance what most include mechanisms use. In Xtext we provide a handy implementation of an *IGlobalScopeProvider* which is based on a naming convention and makes this semantics very easy to use. Talking of the introductory example and given you would want to add support for referencing external *States* and *Events* from within your state machine, all you had to do is add something like the following to the grammar definition:

```
Statemachine :
  (imports+=Import)* // allow imports
  'events'
     (events+=Event)+
  'end'
  ('resetEvents'
     (resetEvents+=[Event])+
  'end')?
  'commands'
     (commands+=Command)+
  'end'
  (states+=State)+;

Import :
  'import' importURI=STRING; // feature must be named importURI
```

This effectively allows import statements to be declared before the events section. In addition you'll have to make sure that you have bound the org.eclipse.xtext.scoping.impl.ImportUriGlobalScopeProvi for the type *IGlobalScopeProvider* by the means of Guice (§3.2). That implementation looks up any *EAttributes* named 'importURI' in your model and interprets their values as URIs that point to imported resources. That is it adds the corresponding resources to the current resource's resource set. In addition the scope provider uses the _IResourceDescription.Manager_ (§4.6.1) of that imported resource to compute all the *IEObjectDescriptions* returned by the *IScope*.

Global scopes based on import URIs are available if you use the org.eclipse.xtext.generator.scoping.ImportURISc in the workflow of your language. It will bind an org.eclipse.xtext.scoping.impl.ImportUriGlobalScopeProvider (*org.eclipse.xtext.resource.ignorecase.IgnoreCaseImportUriGlobalScopeProvider*

fixme

if the *caseInsensitve* flag is set) that handles *importURI* features.

Global Scopes Based On External Configuration (e.g. Classpath-Based)

Instead of explicitly referring to imported resources, the other possibility is to have some kind of external configuration in order to define what is visible from outside a resource. Java for instances uses the notion of classpaths to define containers (jars and

class folders) which contain any referenceable elements. In the case of Java also the order of such entries is important.

Since version 1.0.0 Xtext provides support for this kind of global scoping. To enable it, a org.eclipse.xtext.scoping.impl.DefaultGlobalScopeProvider has to be bound to the *IGlobalScopeProvider* interface. For case insensitive names use the *org.eclipse.xtext.resource.ignorecase.IgnoreCas*

fixme

.

By default Xtext leverages the classpath mechanism since it is well designed and already understood by most of our users. The available tooling provided by JDT and PDE to configure the classpath adds even more value. However, it is just a default: You can reuse the infrastructure without using Java and independent from the JDT.

In order to know what is available in the "world" a global scope provider which relies on external configuration needs to read that configuration in and be able to find all candidates for a certain *EReference*. If you don't want to force users to have a folder and file name structure reflecting the actual qualified names of the referenceable *EObjects*, you'll have to load all resources up front and either keep holding them in memory or remembering all information which is needed for the resolution of cross-references. In Xtext that information is provided by a so called _IEObjectDescription_ (§4.6.1).

About the Index, Containers and Their Manager Xtext ships with an index which remembers all *IResourceDescription* and their *IEObjectDescription* objects. In the IDE-context (i.e. when running the editor, etc.) the index is updated by an incremental project builder. As opposed to that, in a non-UI context you typically do not have to deal with changes such that the infrastructure can be much simpler. In both situations the global index state is held by an implementation of org.eclipse.xtext.resource.IResourceDescriptions (Note the plural form!). The bound singleton in the UI scenario is even aware of unsaved editor changes, such that all linking happens to the latest maybe unsaved version of the resources. You will find the Guice configuration of the global index in the UI scenario in org.eclipse.xtext.ui.shared.internal.SharedModule.

The index is basically a flat list of instances of *IResourceDescription*. The index itself doesn't know about visibility constraints due to classpath restriction. Rather than that, they are defined by the referencing language by means of so called *IContainers*: While Java might load a resource via *ClassLoader.loadResource()* (i.e. using the classpath mechanism), another language could load the same resource using the file system paths.

Consequently, the information which container a resource belongs to depends on the referencing context. Therefore an *IResourceServiceProvider* provides another interesting service, which is called *IContainer.Manager*. For a given *IResourceDescription*, the *IContainer.Manager* provides you with the org.eclipse.xtext.resource.IContainer as well as with a list of all *IContainers* which are visible from there. Note that the index (_IResourceDesciptions_) is globally shared between all languages while the *IContainer.Manager* that adds the semantics of containers can be very different depending on the language. The following method lists all resources visible from a given *Resource*:

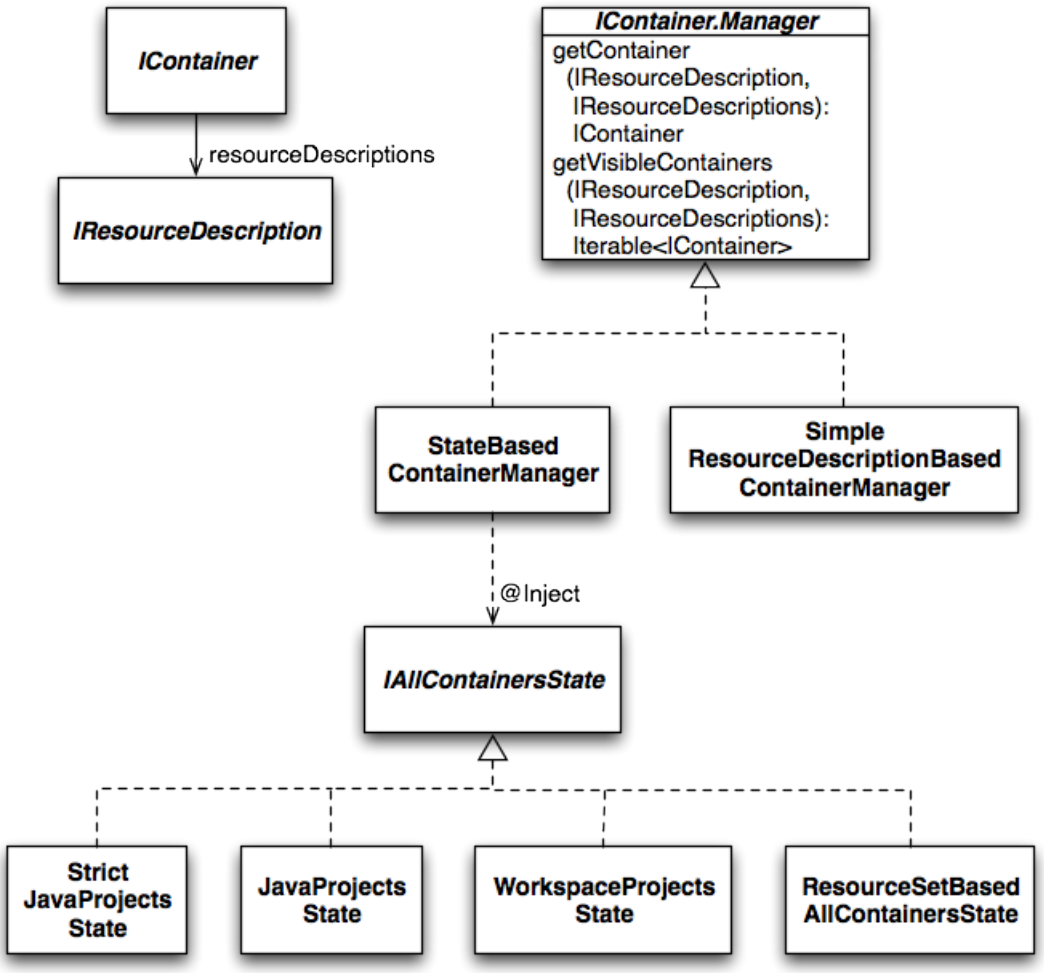
```
@Inject
```

```
IContainer.Manager manager;

public void listVisibleResources(
        Resource myResource, IResourceDescriptions index) {
  IResourceDescription descr =
        index.getResourceDescription(myResource.getURI());
  for(IContainer visibleContainer:
        manager.getVisibleContainers(descr, index)) {
    for(IResourceDescription visibleResourceDesc:
            visibleContainer.getResourceDescription()) {
      System.out.println(visibleResourceDesc.getURI());
    }
  }
}
```

Xtext ships two implementations of *IContainer.Manager* which are as usual bound with Guice: The default binding is to org.eclipse.xtext.resource.impl.SimpleResourceDescriptionsBasedContainerManag which assumes all *IResourceDescription* to be in a single common container. If you don't care about container support, you'll be fine with this one. Alternatively, you can bind org.eclipse.xtext.resource.containers.StateBasedContainerManager and an additional org.eclipse.xtext.resource.containers.IAllContainersState which keeps track of the set of available containers and their visibility relationships.

Xtext offers a couple of strategies for managing containers: If you're running an Eclipse workbench, you can define containers based on Java projects and their classpaths or based on plain Eclipse projects. Outside Eclipse, you can provide a set of file system paths to be scanned for models. All of these only differ in the bound instance of *IAllContainerState* of the referring language. These will be described in detail in the following sections.



JDT-Based Container Manager As JDT is an Eclipse feature, this JDT-based container management is only available in the UI scenario. It assumes so called *IPackageFragmentRoots* as containers. An *IPackageFragmentRoot* in JDT is the root of a tree of Java model elements. It usually refers to

revise indentation levels

- a source folder of a Java project,
- a referenced jar,
- a classpath entry of a referenced Java project, or
- the exported packages of a required PDE plug-in.

So for an element to be referable, its resource must be on the classpath of the caller's Java project and it must be exported (as described above).

As this strategy allows to reuse a lot of nice Java things like jars, OSGi, maven, etc. it is part of the default: You should not have to reconfigure anything to make it work. Nevertheless, if you messed something up, make sure you bind

```
public Class<? extends IContainer.Manager> bindIContainer$Manager() {
  return StateBasedContainerManager.class;
}
```

in the runtime module and

```
public Provider<IAllContainersState> provideIAllContainersState() {
  return org.eclipse.xtext.ui.shared.Access.getJavaProjectsState();
  // return org.eclipse.xtext.ui.shared.Access.getStrictJavaProjectsState();
}
```

in the UI module of the referencing language. The latter looks a bit more difficult than a common binding, as we have to bind a global singleton to a Guice provider. The *StrictJavaProjectsState* requires all elements to be on the classpath, while the default *JavaProjectsState* also allows models in non-source folders.

Eclipse Project-Based Containers If the classpath-based mechanism doesn't work for your case, Xtext offers an alternative container manager based on plain Eclipse projects: Each project acts as a container and the project references *Properties->Project References* are the visible containers.

In this case, your runtime module should define

```
public Class<? extends IContainer.Manager> bindIContainer$Manager() {
  return StateBasedContainerManager.class;
}
```

and the UI module should bind

```
public Provider<IAllContainersState> provideIAllContainersState() {
  return org.eclipse.xtext.ui.shared.Access.getWorkspaceProjectsState();
}
```

ResourceSet-Based Containers If you need an *IContainer.Manager* that is independent of Eclipse projects, you can use the org.eclipse.xtext.resource.containers.ResourceSetBasedAllContainersState. This one can be configured with a mapping of container handles to resource URIs.

It is unlikely you want to use this strategy directly in your own code, but it is used in the back-end of the MWE2 workflow component org.eclipse.xtext.mwe.Reader. This is responsible for reading in models in a workflow, e.g. for later code generation. The *Reader* allows to either scan the whole classpath or a set of paths for all models therein. When paths are given, each path entry becomes an *IContainer* of its own. In the following snippet,

```
component = org.eclipse.xtext.mwe.Reader {
  // lookup all resources on the classpath
  // useJavaClassPath = true
```

```
  // or define search scope explicitly
  path = "src/models"
  path = "src/further-models"

  ...
}
```

4.6.2 Local Scoping

We now know how the outer world of referenceable elements can be defined in Xtext. Nevertheless, not everything is available in any context and with a global name. Rather than that, each context can usually have a different scope. As already stated, scopes can be nested, i.e. a scope can in addition to its own elements contain elements of a parent scope. When parent and child scope contain different elements with the same name, the parent scope's element will usually be *shadowed* by the element from the child scope.

To illustrate that, let's have a look at Java: Java defines 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 shadowed)
      // fileScope{STATIC}->
      // classpathScope{
      // 'all qualified names of accessible static fields'} ->
      // NULLSCOPE{}
      //
    }
    field.add(localVar);
  }
}
```

In fact the classpath scope should also reflect the order of classpath 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 .

Declarative Scoping

If you have to define scopes for certain contexts, the base class org.eclipse.xtext.scoping.impl.AbstractDeclarativeSco allows to do that in a declarative way. It looks up methods which have either of the following two signatures:

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

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

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

The latter method signature is used when computing the scope for a given element type and is applicable to all cross-references of that type. Here *TypeToReturn* is the name of that type.

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. Again it will first attempt to match the context object. Thus in the example the signature first matched would be:

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

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

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

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

4.6.3 Imported Namespace-Aware Scoping

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

org.eclipse.xtext.naming.IQualifiedNameProvider

The org.eclipse.xtext.scoping.impl.ImportedNamespaceAwareLocalScopeProvider makes use of the so called org.eclipse.xtext.naming.IQualifiedNameProvider service. It computes qualified names for EObjects. The default implementation (org.eclipse.xtext.naming.DefaultDeclarativeQualifiedNameProvider) uses a simple name look up and concatenates the result to the qualified name of its parent object using a dot as separator.

It also allows to override the name computation declaratively. The following snippet shows how you could make *Transitions* in the state machine example referable by giving them a name. Don't forget to bind your implementation in your runtime module.

```
FowlerDslQualifiedNameProvider
        extends DefaultDeclarativeQualifiedNameProvider {
  public String qualifiedName(Transition t) {
    if(t.getEvent() == null || !(t.eContainer() instanceof State))
      return null;
    else
      return ((State)t.eContainer()).getName() + "." t.getEvent().getName();
  }
}
```

Importing Namespaces

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

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

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

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

```
package bar {
  entity Foo {}
}
```

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

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

The following would as well be ok:

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

See the JavaDocs and this blog post for details.

4.7 Value Converter

Value converters are registered to convert the parsed text into a certain data type instance and vice versa. The primary hook is called org.eclipse.xtext.conversion.IValueConverterService and the concrete implementation can be registered via the runtime Guice module (§3.2.1). To do so override the corresponding binding in your runtime module like shown in this example:

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

4.7.1 Annotation-Based Value Converters

The most simple way to register additional value converters is to make use of org.eclipse.xtext.conversion.impl.Abs which allows to declaratively register an org.eclipse.xtext.conversion.IValueConverter by means of an annotated method.

If you use the common terminals grammar *org.eclipse.xtext.common.Terminals* you should subclass org.eclipse.xtext.common.services.DefaultTerminalConverters and override or add additional value converters by adding the respective methods. In addition to the explicitly defined converters in the default implementation, a delegating converter is registered for each available *EDataType* that reuses the functionality of the corresponding EMF *EFactory*.

As qualified names - i.e. names composed of namespaces separated by a delimiter - are expected to occur often, we've added a org.eclipse.xtext.conversion.impl.QualifiedNameValueConverter. This one removes comments and whitespaces and delegates to another value converter

for each segment, allowing individually quoted segments. The domainmodel example shows how to use it.

The protocol of an *IValueConverter* allows to throw a *ValueConverterException* if something went wrong. The exception is propagated as a syntax error by the parser or as a validation problem by the *ConcreteSyntaxValidator* if the value cannot be converted to a valid string. The org.eclipse.xtext.conversion.impl.AbstractLexerBasedConverter is useful when implementing a custom value converter. If the converter needs to know about the rule that it currently works with, it may implement the interface *IValueConverter.RuleSpecific*. The framework will set the rule such as the implementation may use it afterwards.

4.8 Serialization

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

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

revise indentation levels

1. Validating the semantic model. This is optional, enabled by default, done by the concrete syntax validator (§4.4.1) and can be turned off in the save options (§4.8.4).
2. Matching the model elements with the grammar rules and creating a stream of tokens. This is done by the parse tree constructor (§4.8.3).
3. Associating comments with semantic objects. This is done by the comment associator (§4.8.5).
4. Associating existing nodes from the node model with tokens from the token stream.
5. Merging existing whitespace (§4.8.9) and line-wraps into to token stream.
6. Adding further needed whitespace or replacing all whitespace using a formatter (§4.9).

Serialization is invoked when calling org.eclipse.xtext.resource.XtextResource.save(...). Furthermore, org.eclipse.xtext.parsetree.reconstr.Serializer provides resource-independent support for serialization. Serialization is *not* called when a textual editors contents is saved to disk. Another situation that triggers serialization is applying Quick Fixes (§6.3) with semantic modifications.

4.8.1 The Contract

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

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

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

4.8.2 Roles of the Semantic Model and the Node Model During Serialization

A serialized document represents the state of the semantic model. However, if there is a node model available (i.e. the semantic model has been created by the parser), the serializer

revise indentation levels

- preserves existing whitespaces (§4.8.9) from the node model.
- preserves existing comments (§4.8.5) from the node model.
- preserves the representation of cross-references: If a cross-referenced object can be identified by multiple names (i.e. scoping returns multiple EObjectDescriptions for the same object), the serializer tries to keep the previously used name.
- preserves the representation of values: For values handled by the value converter (§4.7), the serializer checks whether the textual representation converted to a value equals the value from the semantic model. If that is true, the textual representation is kept.

4.8.3 Parse Tree Constructor

The parse tree constructor usually does not need to be customized since it is automatically derived from the Xtext Grammar (§2). However, it can be helpful 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 non-transient element of the to-be-serialized EMF model. To *consume* means, in this context, to write the element to the textual representation of the model. This can turn out to be a not-so-easy-to-fulfill requirement, since a grammar usually introduces implicit constraints to the EMF model as explained for the concrete syntax validator (§4.4.1).

If a model can not be serialized, an org.eclipse.xtext.parsetree.reconstr.XtextSerializationException is thrown. Possible reasons are listed below:

revise indentation levels

- 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 (§4.8.6) can be used to indicate that this model element should not be consumed.
- An assignment in the grammar has no corresponding model element. The default transient value service considers a model element to be transient if it is *unset* or equals its default value. However, the parse tree constructor may serialize default values if this is required by a grammar constraint to be able to serialize another model element. The following solution may help to solve such a scenario:
- A model element should be added to the model.
- The assignment in the grammar should be made optional.
- The type of the model element differs from the type in the grammar. The type of the model element must be identical to the return type of the grammar rule or the action's type. Sub-types are not allowed.
- Value conversion (§4.7) fails. The value converter can indicate that a value is not serializeable by throwing a `org.eclipse.xtext.conversion.ValueConverterException`.
- An enum literal is not allowed at this position. This can happen if the referenced enum rule only lists a subset of the literals of the actual enumeration.

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

revise indentation levels

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

4.8.4 Options

org.eclipse.xtext.resource.SaveOptions can be passed to org.eclipse.xtext.resource.XtextResource.save(options) and to org.eclipse.xtext.parsetree.reconstr.Serializer.serialize(..). Available options are:

revise indentation levels

- *Formatting*. Default: *false*. If enabled, it is the formatters (§4.9) job to determine all whitespace information during serialization. If disabled, the formatter only defines whitespace information for the places in which no whitespace information can be preserved from the node model. E.g. When new model elements are inserted or there is no node model.
- *Validating*. Default: *true*: Run the concrete syntax validator (§4.4.1) before serializing the model.

4.8.5 Preserving Comments from the Node Model

The org.eclipse.xtext.parsetree.reconstr.ICommentAssociater associates comments with semantic objects. This is important in case an element in the semantic model is moved to a different position and the model is serialized, one expects the comments to be moved to the new position in the document as well.

Which comment belongs to which semantic object is surely a very subjective issue. The default implementation (org.eclipse.xtext.parsetree.reconstr.impl.DefaultCommentAssociater) behaves as follows, but can be customized:

revise indentation levels

- If there is a semantic token before a comment and in the same line, the comment is associated with this token's semantic object.
- In all other cases, the comment is associated with the semantic object of the next following object.

4.8.6 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 org.eclipse.xtext.parsetree.reconstr.ITransientValueService, or serialization will fail. The default implementation marks all model elements transient, which are *eStructuralFeature.isTransient()* or not *eObject.eIsSet(eStructuralFeature)*. By default, EMF returns *false* for *eIsSet(...)* if the value equals the default value.

4.8.7 Unassigned Text

If there are calls of data type rules or terminal rules that do not reside in an assignment, the serializer by default doesn't know which value to use for serialization.

. Example:

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

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

Valid models for this example are *contents 1 item* or *contents 5 items*. However, it is not stored in the semantic model whether the keyword *item* or *items* has been parsed. This is due to the fact that the rule call *Plural* is unassigned. However, the parse tree constructor (§4.8.3) needs to decide which value to write during serialization. This decision can be be made by customizing the *IValueSerializer.serializeUnassignedValue(EObject, RuleCall, AbstractNode)*.

4.8.8 Cross-Reference Serializer

The cross-reference serializer specifies which values are to be written to the textual representation for cross-references. This behavior can be customized by implementing *ICrossReferenceSerializer*. The default implementation delegates to various other services such as the *IScopeProvider* or the *LinkingHelper* each of which may be the better place for customization.

4.8.9 Merge Whitespaces

After the parse tree constructor (§4.8.3) has done its job to create a stream of tokens which are to be written to the textual representation, and the comment associator (§4.8.5) has done its work, existing whitespace form the node model is merged into the stream.

The strategy is as follows: If two tokens follow each other in the stream and the corresponding nodes in the node model follow each other as well, then the whitespace information in between is kept. In all other cases it is up to the formatter (§4.9) to calculate new whitespace information.

4.8.10 Token Stream

The parse tree constructor (§4.8.3) and the formatter (§4.9) use an org.eclipse.xtext.parsetree.reconstr.ITokenStrea
for their output, and the latter for its input as well. This makes them chainable. Token streams can be converted to a *String* using the org.eclipse.xtext.parsetree.reconstr.impl.TokenStringBuffer and to a *Writer* using the org.eclipse.xtext.parsetree.reconstr.impl.WriterTokenStream.

public interface ITokenStream {

void flush() throws IOException; void writeHidden(EObject grammarElement, String value) throws IOException; void writeSemantic(EObject grammarElement, String value) throws IOException; }

4.9 Formatting (Pretty Printing)

A formatter can be implemented via the org.eclipse.xtext.formatting.IFormatter service. Technically speaking, a formatter is a Token Stream (§4.8.10) which inserts/removes/modifies hidden tokens (whitespace, line-breaks, comments).

The formatter is invoked during the serialization phase (§4.8) and when the user triggers formatting in the editor (for example, using the CTRL+SHIFT+F shortcut).

Xtext ships with two formatters:

revise indentation levels

- The org.eclipse.xtext.formatting.impl.OneWhitespaceFormatter simply writes one whitespace between all tokens.
- The org.eclipse.xtext.formatting.impl.AbstractDeclarativeFormatter allows advanced configuration using a org.eclipse.xtext.formatting.impl.FormattingConfig. Both are explained in the next chapter (§4.9.1).

4.9.1 Declarative Formatter

A declarative formatter can be implemented by sub-classing org.eclipse.xtext.formatting.impl.AbstractDeclarativeFo as shown in the following example:

```
public class ExampleFormatter extends AbstractDeclarativeFormatter {

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

    c.setAutoLinewrap(120);

    // find common keywords an specify formatting for them
    for (Pair<Keyword, Keyword> pair : f.findKeywordPairs("(", ")")) {
      c.setNoSpace().after(pair.getFirst());
      c.setNoSpace().before(pair.getSecond());
    }
    for (Keyword comma : f.findKeywords(",")) {
      c.setNoSpace().before(comma);
    }

    // formatting for grammar rule Line
    c.setLinewrap(2).after(f.getLineAccess().getSemicolonKeyword_1());
    c.setNoSpace().before(f.getLineAccess().getSemicolonKeyword_1());

    // formatting for grammar rule TestIndentation
    c.setIndentationIncrement().after(
        f.getTestIndentationAccess().getLeftCurlyBracketKeyword_1());
    c.setIndentationDecrement().before(
        f.getTestIndentationAccess().getRightCurlyBracketKeyword_3());
```

```
    c.setLinewrap().after(
        f.getTestIndentationAccess().getLeftCurlyBracketKeyword_1());
    c.setLinewrap().after(
        f.getTestIndentationAccess().getRightCurlyBracketKeyword_3());

    // formatting for grammar rule Param
    c.setNoLinewrap().around(f.getParamAccess().getColonKeyword_1());
    c.setNoSpace().around(f.getParamAccess().getColonKeyword_1());

    // formatting for Comments
    cfg.setLinewrap(0, 1, 2).before(g.getSL_COMMENTRule());
    cfg.setLinewrap(0, 1, 2).before(g.getML_COMMENTRule());
    cfg.setLinewrap(0, 1, 1).after(g.getML_COMMENTRule());
  }
}
```

The formatter has to implement the method *configureFormatting(...)* which declaratively sets up a org.eclipse.xtext.formatting.impl.FormattingConfig.

The org.eclipse.xtext.formatting.impl.FormattingConfig consist of general settings and a set of formatting instructions:

General FormattingConfig Settings

revise indentation levels

- _setAutoLinewrap(int)_ defines the amount of characters after which a line-break should be dynamically inserted between two tokens. The instructions *setNoLinewrap().???()*, *setNoSpace().???()* and *setSpace(space).???()* suppress this behavior locally. The default is 80.

FormattingConfig Instructions

Per default, the declarative formatter (§4.9.1) inserts one whitespace between two tokens. Instructions can be used to specify a different behavior. They consist of two parts: *When* to apply the instruction and *what* to do.

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

revise indentation levels

- _after(ele)_: The instruction is applied after the grammar element *ele* has been matched. For example, if your grammar uses the keyword ";" to end lines, this can instruct the formatter to insert a line break after the semicolon.
- _before(ele)_: The instruction is executed before the matched element. For example, if your grammar contains lists which separate their values with the keyword ",", you can instruct the formatter to suppress the whitespace before the comma.

- _around(ele)_: This is the same as *before(ele)* combined with *after(ele)*.
- _between(ele1, ele2)_: This matches if *ele2* directly follows *ele1* in the document. There may be no other tokens in between *ele1* and *ele2*.
- _bounds(ele1, ele2)_: This is the same as *after(ele1)* combined with *before(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 term *tokens* is used slightly different here compared to the parser/lexer. Here, a token is a keyword or the string that is matched by a terminal rule, data type rule or cross-reference. In the terminology of the lexer a data type rule can match a composition of multiple tokens.

The parameter *ele* can be a grammar's *AbstractElement* or a grammar's *AbstractRule*. All grammar rules and almost all abstract elements can be matched. This includes rule calls, parser rules, groups and alternatives. The semantic of *before(ele)*, *after(ele)*, etc. for rule calls and parser rules is identical to when the parser would "pass" this part of the grammar. The stack of called rules is taken into account. The following abstract elements can *not* have assigned formatting instructions:

revise indentation levels

- Actions. E.g. {*MyAction*} or {*MyAction.myFeature=current*}.
- Grammar elements nested in data type rules. This is due to to the fact that tokens matched by a data type rule are treated as atomic by the serializer. To format these tokens, please implement a ValueConverter (§4.7).
- Grammar elements nested in *CrossReferences*.

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

revise indentation levels

- _setIndentationIncrement()_ increments indentation by one unit at this position. Whether one unit consists of one tab-character or spaces is defined by org.eclipse.xtext.formatting.IIndentationInformation. The default implementation consults Eclipse's *PreferenceStore*.
- _setIndentationDecrement()_ decrements indentation by one unit.
- _setLinewrap()_: Inserts a line-wrap at this position.
- _setLinewrap(int count)_: Inserts *count* numbers of line-wrap at this position.

- _setLinewrap(int min, int default, int max)_: If the amount of line-wraps that have been at this position before formatting can be determined (i.e. when a node model is present), then the amount of of line-wraps is adjusted to be within the interval [_min_, *max*] and is then reused. In all other cases *default* line-wraps are inserted. Example: *setLinewrap(0, 0, 1)* will preserve existing line-wraps, but won't allow more than one line-wrap between two tokens.
- _setNoLinewrap()_: Suppresses automatic line wrap, which may occur when the line's length exceeds the defined limit.
- _setSpace(String space)_: Inserts the string *space* at this position. If you use this to insert something else than whitespace, tabs or newlines, a small puppy will die somewhere in this world.
- _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.

Grammar Element Finders

Sometimes, if a grammar contains many similar elements for which the same formatting instructions ought to apply, it can be tedious to specify them for each grammar element individually. The org.eclipse.xtext.IGrammarAccess provides convenience methods for this. The find methods are available for the grammar and for each parser rule.

revise indentation levels

- _findKeywords(String... keywords)_ returns all keywords that equal one of the parameters.
- _findKeywordPairs(String leftKw, String rightKw)_: returns tuples of keywords from the same grammar rule. Pairs are matched nested and sequentially. Example: for *Rule: '(' name=ID ('(' foo=ID ')') ')' | '(' bar=ID ')' findKeywordPairs("(", ")")* returns three pairs.

4.10 Fragment Provider (Referencing Xtext Models From Other EMF Artifacts)

Revert to section2 when we allow Section2Refs

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 and a second part that points to an object. 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 resource shipped with EMF provides a generic path-like computation of fragments. These fragment paths are unique by default and do not have to be serialized. On the other hand, they can be easily broken by reordering the elements in a resource.

With an XMI or other binary-like serialization it is also common and possible to use UUIDs. UUIDs are usually binary and technical, so you don't want them in human readable representations.

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

Therefore one can contribute a so called org.eclipse.xtext.resource.IFragmentProvider per language. It has two methods: *getFragment(EObject, Fallback)* to calculate the fragment of an *EObject* and *getEObject(Resource, String, Fallback)* to go the opposite direction. The *Fallback* interface allows to delegate to the default strategy - usually the fragment paths described above.

The following snippet from the GMF Example (§9.4) shows how to use qualified names as fragments:

```
public QualifiedNameFragmentProvider implements IFragmentProvider {

  @Inject
  private IQualifiedNameProvider qualifiedNameProvider;

  public String getFragment(EObject obj, Fallback fallback) {
    String qualifiedName = qualifiedNameProvider.getQualifiedName(obj);
    return qualifiedName != null ? qualifiedName : fallback.getFragment(obj);
  }

  public EObject getEObject(Resource resource,
                            String fragment,
                            Fallback fallback) {
    if (fragment != null) {
      Iterator<EObject> i = EcoreUtil.getAllContents(resource, false);
      while(i.hasNext()) {
        EObject eObject = i.next();
        String candidateFragment = (eObject.eIsProxy())
            ? ((InternalEObject) eObject).eProxyURI().fragment()
            : getFragment(eObject, fallback);
        if (fragment.equals(candidateFragment))
          return eObject;
      }
    }
    return fallback.getEObject(fragment);
  }
}
```

For performance reasons it is usually a good idea to navigate the resource based on the fragment information instead of traversing it completely. If you know that your fragment is computed from qualified names and your model contains something like *NamedElements*, you should split your fragment into those parts and query the root elements, the children of the best match and so on.

Furthermore it's a good idea to have some kind of conflict resolution strategy to be able

to distinguish between equally named elements that actually are different, e.g. properties may have the very same qualified name as entities.

4.11 Encoding in Xtext

Encoding, AKA *character set*, describes the way characters are encoded into bytes and vice versa. Famous standard encodings are "UTF-8" or "ISO-8859-1". The list of available encodings can be determined by calling *java.nio.Charset.availableCharsets()*. There is also a list of encodings and their canonical Java names in the API docs

Unfortunately, each platform and/or spoken language tends to define its own native encoding, e.g. *Cp1258* on Windows in Vietnamese or *MacIceland* on Mac OS X in Icelandic.

In an Eclipse workspace, files, folders, projects can have individual encodings, which are stored in the hidden file *.settings/org.eclipse.core.resources.prefs* in each project. If a resource does not have an explicit encoding, it inherits the one from its parent recursively. Eclipse chooses the native platform encoding as the default for the workspace root. You can change the default workspace encoding in the Eclipse preferences *Preferences->Workspace->Default text encoding*. If you develop on different platforms, you should consider choosing an explicit common encoding for your text or code files, especially if you use special characters.

While Eclipse allows to define and inspect the encoding of a file, your file system usually doesn't. Given an arbitrary text file there is no general strategy to tell how it was encoded. If you deploy an Eclipse project as a jar (even a plug-in), any encoding information not stored in the file itself is lost, too. Some languages define the encoding of a file explicitly, as in the first processing instruction of an XML file. Most languages don't. Others imply a fixed encoding or offer enhanced syntax for character literals, e.g. $\backslash uXXXX$ in Java.

As Xtext is about textual modeling, it allows to tweak the encoding in various places.

4.11.1 Encoding at Language Design Time

The plug-ins created by the *New Xtext Project* wizard are by default encoded in the workspace's standard encoding. The same holds for all files that Xtext generates in there. If you want to change that, e.g. because your grammar uses/allows special characters, you should manually set the encoding in the properties of these projects after their creation. Do this before adding special characters to your grammar or at least make sure the grammar reads correctly after the encoding change. To tell the Xtext generator to generate files in the same encoding, set the encoding property in the workflow next to your grammar, e.g.

```
Generator {
    encoding ="UTF-8"
    ...
```

4.11.2 Encoding at Language Runtime

As each language could handle the encoding problem differently, Xtext offers a service here. The org.eclipse.xtext.parser.IEncodingProvider has a single method *getEncoding(URI)* to define the encoding of the resource with the given URI. Users can implement their own strategy but keep in mind that this is not intended to be a long running method. If the encoding is stored within the model file itself, it should be extractable in an easy way, like from the first line in an XML file. The default implementation returns the default Java character set in the runtime scenario.

In the UI scenario, when there is a workspace, users will expect the encoding of the model files to be settable the same way as for other files in the workspace. The default implementation of the *IEncodingProvider* in the UI scenario therefore returns the file's workspace encoding for files in the workspace and delegates to the runtime implementation for all other resources, e.g. models in a jar or from a deployed plug-in. Keep in mind that you are going to loose the workspace encoding information as soon as you leave this workspace, e.g. deploy your project.

Unless you want to enforce a uniform encoding for all models of your language, we advise to override the runtime service only. It is bound in the runtime module using the binding annotation *@Runtime*:

```
@Override
public void configureRuntimeEncodingProvider(Binder binder) {
    binder.bind(IEncodingProvider.class)
          .annotatedWith(DispatchingProvider.Runtime.class)
          .to(MyEncodingProvider.class);
}
```

For the uniform encoding, bind the plain *IEncodingProvider* to the same implementation in both modules:

```
@Override
public Class<? extends IEncodingProvider> bindIEncodingProvider() {
    return MyEncodingProvider.class;
}
```

4.11.3 Encoding of an XtextResource

An org.eclipse.xtext.resource.XtextResource uses the *IEncodingProvider* of your language by default. You can override that by passing an option on load and save, e.g.

```
Map<?,?> options = new HashMap();
options.put(XtextResource.OPTION_ENCODING, "UTF-8");
myXtextResource.load(options);
options.put(XtextResource.OPTION_ENCODING, "ISO-8859-1");
myXtextResource.save(options);
```

4.11.4 Encoding in New Model Projects

The org.eclipse.xtext.ui.generator.projectWizard.SimpleProjectWizardFragment generates a wizard that clients of your language can use to create model projects. This wizard expects

its templates to be in the encoding of the Generator that created it (see above). As for every new project wizard, its output will be encoded in the default encoding of the target workspace.

4.11.5 Encoding of Xtext Source Code

The source code of the Xtext framework itself is completely encoded in "ISO 8859-1", which is necessary to make the Xpand templates work everywhere (they use french quotation markup). That encoding is hard coded into the Xtext generator code. You are likely never going to change that.

5 MWE2

The Modeling Workflow Engine 2 (MWE2) is a rewritten backwards compatible implementation of the Modeling Workflow Engine (MWE). It is a declarative, externally configurable generator engine. Users can describe arbitrary object compositions by means of a simple, concise syntax that allows to declare object instances, attribute values and references. One use case - that's where the name had its origins - is the definition of workflows. Such a workflow consists usually of a number of components that interact with each other. There are components to read EMF resources, to perform operations (transformations) on them and to write them back or to generate any number of other artifacts out of the information. Workflows are typically executed in a single JVM. However there are no constraints the prevent implementors to provide components that spawn multiple threads or new processes.

5.1 Examples

Let's start with a couple of examples to demonstrate some usage scenarios for MWE2. The first examples is a simple *HelloWorld* module that does nothing but print a message to standard out. The second module is assembled of three components that read an Ecore file, transform the contained classifier-names to upper-case and serialize the resource back to a new file. The last examples uses the life-cycle methods to print the execution time of the workflow.

5.1.1 The Simplest Workflow

The arguably shortest MWE2 module may look like the following snippet.

```
module HelloWorld

SayHello {
  message = "Hello World!"
}
```

It configures a very simple workflow component with a message that should be printed to *System.out* when the workflow is executed. The module begins with a declaration of its name. It must fulfill the Java conventions for fully qualified class-names. That's why the module *HelloWorld* has to be placed into the default package of a Java source folder. The second element in the module is the class-name *SayHello* which introduces the root element of the module. The interpreter will create an instance of the given type and configure it as declared between the curly braces. E.g. the assignment *message = "Hello World!"* in the module will be interpreted as an invocation of the *setMessage(String)*

on the instantiated object. As one can easily imagine, the implementation of the class *SayHello* looks straight forward:

```
import org.eclipse.emf.mwe2.runtime.workflow.IWorkflowComponent;
import org.eclipse.emf.mwe2.runtime.workflow.IWorkflowContext;

public class SayHello implements IWorkflowComponent {

  private String message = "Hello␣World!";
  public void setMessage(String message) {
    this.message = message;
  }
  public String getMessage() {
    return message;
  }

  public void invoke(IWorkflowContext ctx) {
    System.out.println(getMessage());
  }

  public void postInvoke() {}
  public void preInvoke() {}
}
```

It looks like a simple POJO and that's the philosophy behind MWE2. It is easily possible to assemble completely independent objects in a declarative manner. To make the workflow executable with the *WorkflowRunner*, the component *SayHello* must be nested in a root workflow:

```
module HelloWorld

Workflow {
    component = SayHello {
        message = "Hello␣World!"
    }
}
```

The class *Workflow* is actually *org.eclipse.emf.mwe2.runtime.workflow.Workflow* but its package is implicitly imported in MWE2 modules to make the the modules more concise. The execution result of this workflow will be revealed after a quick *Run As .. -> MWE2 Workflow* in the console as

```
Hello World!
```

5.1.2 A Simple Transformation

The following workflow solves the exemplary task to rename every *EClassifier* in an **.ecore* file. It consists of three components that read, modify and write the model file:

```
module Renamer
Workflow {
  component = ResourceReader {
```

```
        uri = "model.ecore"
  }
  component = RenamingTransformer {}
  component = ResourceWriter {
        uri = "uppercaseModel.ecore"
  }
}
```

The implementation of these components is surprisingly simple. It is easily possible to create own components even for minor operations to automate a process.

The *ResourceReader* simply reads the file with the given *URI* and stores it in a so called *slot* of the workflow context. A slot can be understood as a dictionary or map-entry.

```
public class ResourceReader extends WorkflowComponentWithSlot {
  private String uri;
  public void invoke(IWorkflowContext ctx) {
    ResourceSet resourceSet = new ResourceSetImpl();
    URI fileURI = URI.createFileURI(uri);
    Resource resource = resourceSet.getResource(fileURI, true);
    ctx.put(getSlot(), resource);
  }

  public void setUri(String uri) {
    this.uri = uri;
  }
  public String getUri() {
    return uri;
  }
}
```

The actual transformer takes the model from the slot and modifies it. It simply iterates the content of the resource, identifies each *EClassifier* and sets its name.

```
public class RenamingTransformer extends WorkflowComponentWithSlot {
  private boolean toLowerCase = false;
  public void invoke(IWorkflowContext ctx) {
    Resource resource = (Resource) ctx.get(getSlot());
    EcoreUtil.resolveAll(resource);
    Iterator<Object> contents = EcoreUtil.getAllContents(resource, true);
    Iterator<EClassifier> iter =
        Iterators.filter(contents, EClassifier.class);
    while(iter.hasNext()) {
      EClassifier classifier = (EClassifier) iter.next();
      classifier.setName(isToLowerCase()
          ? classifier.getName().toLowerCase()
          : classifier.getName().toUpperCase());
    }
  }

  public void setToLowerCase(boolean toLowerCase) {
```

```
    this.toLowerCase = toLowerCase;
  }
  public boolean isToLowerCase() {
    return toLowerCase;
  }
}
```

After the model has been modified it should be written to a new file. That's what the *ResourceWriter* does. It actually takes the resource from the given *slot* and saves it with the configured *URI*:

```
public class ResourceWriter extends WorkflowComponentWithSlot {
  private String uri;
  public void invoke(IWorkflowContext ctx) {
    Resource resource = (Resource) ctx.get(getSlot());
    URI uri = URI.createFileURI(getUri());
    uri = resource.getResourceSet().getURIConverter().normalize(uri);
    resource.setURI(uri);
    try {
      resource.save(null);
    } catch (IOException e) {
      throw new WrappedException(e);
    }
  }

  public void setUri(String uri) {
    this.uri = uri;
  }
  public String getUri() {
    return uri;
  }
}
```

Last but not least, the common super-type for those components looks like this:

```
public abstract class WorkflowComponentWithSlot
      implements IWorkflowComponent {
  private String slot = "model";
  public void setSlot(String slot) {
    this.slot = slot;
  }
  public String getSlot() {
    return slot;
  }

  public void postInvoke() {}
  public void preInvoke() {}
}
```

Each of the mentioned implementations is rather simple and can be done in a couple of minutes. This is true for many tedious tasks that developers face in their daily work. MWE2 can be used to automize these tasks with minimum effort.

5.1.3 A Stopwatch

The last example demonstrates how to combine the MWE2 concepts to create a simple stopwatch that allows to measure the execution time of a set of components. The idea is to add the very same stopwatch twice as a component to a workflow. It will measure the time from the first pre-invoke to the last post-invoke event and print the elapsed milliseconds to the console.

```
public class StopWatch implements IWorkflowComponent {
  private long start;
  private boolean shouldStop = false;
  public void invoke(IWorkflowContext ctx) {}

  public void postInvoke() {
    if (shouldStop) {
      long elapsed = System.currentTimeMillis() - start;
      System.out.println("Time elapsed: " + elapsed + " ms");
    }
    shouldStop = true;
  }

  public void preInvoke() {
    start = System.currentTimeMillis();
  }
}
```

Clients who want to leverage this kind of stopwatch may use the following pattern. The stopwatch-instance will be added as the first component and the last component to a workflow. Everything in between will be measured. In this case, it is another workflow that does not need know about this decoration. The idea is to use a local identifier for the instantiated *StopWatch* and reuse this one at the end to receive the post-invoke life-cycle event twice.

```
module MeasuredWorkflow

Workflow {
  component = StopWatch: stopWatch {}
  component = @OtherWorkflow {}
  component = stopWatch
}
```

5.2 Language Reference

MWE2 has a few well defined concepts which can be combined to assemble arbitrary object graphs in a compact and declarative manner.

revise indentation levels

- A MWE2 file defines a *module* which exposes its root *component* as reusable artifact.

- _Properties_ can be used to extract reusable, configurable parts of the workflow.
- Components are mapped to plain vanilla *Java objects*. Arbitrary *set- and add-methods* are used to configure them.

Let's consider the follow short example module and *SampleClass* to explain these concepts.

```
module com.mycompany.Example

import java.util.*

SampleClass {
  singleValue = 'a string'
  multiValue = ArrayList {}
  child = {}
}
```

```
package com.mycompany;

import java.util.List;

public class SampleClass {
  public void setSingleValue(String value) {..}
  public void addMultiValue(List<?> value) {..}
  public void addChild(SampleClass value) {..}
}
```

5.2.1 Mapping to Java Classes

The module *com.mycompany.Example* defines a root component of type *com.mycompany.SampleClass*. It is possible to use the simple class-name because MWE2 uses the very same visibility rules as the Java compiler. Classes that are in the same package as the module can be referenced by their simple name. The same rule applies for classes from the *java.lang* package. For convenience reasons is the package *org.eclipse.emf.mwe2.runtime.workflow* implicitly imported as well as it exposes some library workflow components. However, the imports are more flexible then in Java since MWE2-imports can be relative, e.g. the *import java.** resolves the reference *util.ArrayList* to *java.util.ArrayList*.

The root instance of type *SampleClass* has to be configured after it has been created. Therefore the method *setSingleValue* will be called at first. The given parameter is *'a string'*. The method is identified by its name which starts with *set*. To allow to assign multi-value properties, MWE provides access to methods called *add** as well.

If the right side of the assignment in the workflow file does not define a class explicitly, its type is inferred from the method parameter. The line *child* = {} is equivalent to *child* = *SampleClass* {} and creates a new instance of *SampleClass*.

MWE2 ships with nice tool support. The editor will provide content assist for the allowed types and highlight incompatible assignments. The available properties for Java classes will be proposed as well.

5.2.2 Module

As MWE2 modules have a fully qualified name, it is possible to refer to them from other modules. The type of the module is derived from the type of its root component. The *com.mycompany.Example* can be assigned at any place where a *com.mycompany.SampleClass* is expected.

Let's create a second module *com.mycompany.Second* like this:

```
module com.mycompany.sub.Second

import com.mycompany.*

SampleClass {
  child = @Example {}
}
```

The *child* value will be assigned to an instance of *SampleClass* that is configured as in the first example workflow. This enables nice composition and a very focused, reusable component design.

As the same rules apply in MWE2 like in Java, the module *com.mycompany.sub.Second* has to be defined in a file called *Second.mwe2* in the package *com.mycompany.sub*. The import semantic for other modules is the same as for classes. The import statement allows to refer to *com.mycompany.Example* with a shortened name.

5.2.3 Properties

MWE2 allows to extract arbitrary information into properties to ensure that these pieces are not cluttered around the workflow and to allow for easier external customization. The exemplary component definition was only changed slightly by introducing a property *value*.

```
module com.mycompany.Example

var value = 'a string'

SampleClass {
  singleValue = value
}
```

The type of the property will be derived from the default value similar to the mechanism that is already known from *set-* and *add*-methods. If no default value is given, *java.lang.String* will be assumed. However, properties are not limited to strings. The second built in type is boolean via the familiar literals *true* and *false*. More flexibility is available via actual component literals.

```
module com.mycompany.Example

var childInstance = SampleClass {
                      singleValue = "child"
                    }
```

```
SampleClass {
  child = childInstance
}
```

If one wants to define string properties that are actual reusable parts for other properties, she may use defined variables inside other literals like this:

```
var aString = "part"
var anotherString = "reuse␣the␣${part}␣here"
```

This is especially useful for file paths in workflows as one would usually want to define some common root directories only ones in the workflow and reuse this fragment across certain other file locations.

5.2.4 Mandatory Properties

It is not always feasible to define default values for properties. That is where mandatory properties come into play. Modules define their interface not only via their fully qualified name and the type of the root component but also by means of the defined properties.

```
module com.mycompany.Example

var optional = 'a␣string'
var mandatory

SampleClass {
  singleValue = optional
  child = {
    singleValue = mandatory
  }
}
```

This version of the example module exposes two externally assignable properties. The second one has no default value assigned and is thereby considered to be mandatory. The mandatory value must be assigned if we reuse *org.mycompany.Example* in another module like this:

```
module com.mycompany.Second

var newMandatory

@Example {
  mandatory = "mandatoryValue"
  optional = newMandatory
}
```

Note that it is even possible to reuse another module as the root component of a new module. In this case we set the mandatory property of *Example* to a specific constant value while the previously optional value is now redefined as mandatory by means of a new property without a default value.

It is not only possible to define mandatory properties for MWE2 modules but for classes as well. Therefore MWE2 ships with the *@Mandatory* annotation. If a *set-* or *add-*method is marked as *mandatory*, the module validation will fail if no value was assigned to that feature.

5.2.5 Named Components

Properties are not the only way to define something that can be reused. It is possible to assign a name to any instantiated component whether it's created from a class literal or from another component. This allows to refer to previously created and configured instances. Named instances can come handy for notification and call-back mechanisms or more general in terms of defined life-cycle events.

If we wanted to assign the created instance to a property of itself, we could use the following syntax:

```
module com.mycompany.Example

SampleClass : self {
  child = self
}
```

A named component can be referenced immediately after its creation but it is not possible to define forward references in a MWE2 file.

5.2.6 Auto Injection

Existing modules or classes often expose a set of properties that will be assigned to features of its root component or set- and add- methods respectively. In many cases its quite hard to come up with yet another name for the very same concept which leads to the situation where the properties itself have the very same name as the component's feature. To avoid the overall repetition of assignments, MWE2 offers the possibility to use the *auto-inject* modifier on the component literal:

```
module com.mycompany.Example

var child = SampleClass {}

SampleClass auto-inject {
}
```

This example will implicitly assign the value of the property *child* to the feature *child* of the root component. This is especially useful for highly configurable workflows that expose dozens of optional parameters each of which can be assigned to one or more components.

The *auto-inject* modifier can be used for a subset of the available features as well. It will suppressed for the explicitly set values of a component.

5.3 Syntax Reference

The following chapter serves as a reference for the concrete syntax of MWE2. The building blocks of a module will be described in a few words.

MWE2 is not sensitive to white-space and allows to define line-comments and block comments everywhere. The syntax is the same as one is used to from the Java language:

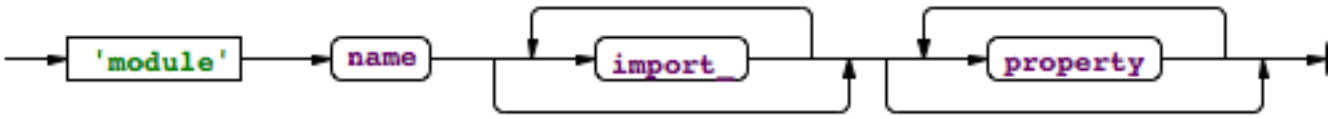
```
// This is a comment
/*
  This is another one.
*/
```

Every name in MWE2 can be a fully qualified identifier and must follow the Java conventions. However, in contrast to Java identifiers it is not allowed to use German umlauts or Unicode escape sequences in identifiers. A valid ID-segment in MWE2 starts with a letter or an underscore and is followed by any number of letters, numbers or underscores. An identifier is composed from one or more segments which are delimited by a '.' dot.

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

MWE2 does not use a semicolon as statement delimiter at any place.

5.3.1 Module



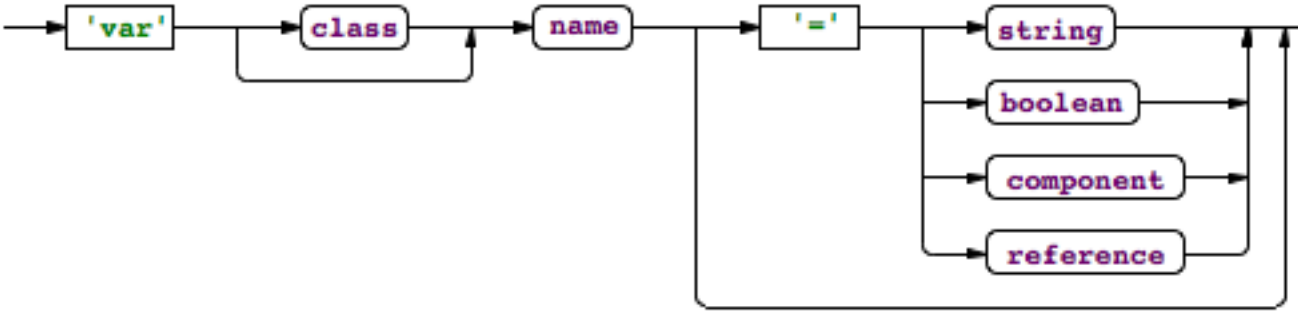
A *module* consists of four parts. The very first statement in a **.mwe2* file is the module declaration. The name of the module must follow the naming convention for Java classes. That MWE2 file's name must therefore be the same as the last segment of the module-name and it has to be placed in the appropriate package of a Java source path.

It is allowed to define any number of import statements in a module. Imports are either suffixed by a wild-card or concrete for a class or module. MWE2 can handle relative imports in case one uses the wild-card notation:

```
'import'name '.*'?
```

5.3.2 Property

The list of declared properties follows the optional import section. It is allowed to define modules without any properties.

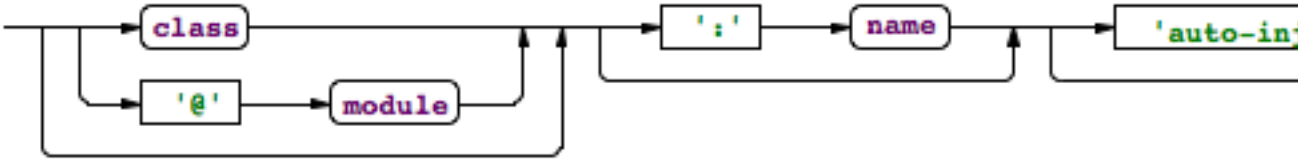


Each declared property is locally visible in the module. It furthermore defines an assignable feature of the module in case one refers to it from another module. Properties may either have a default value or they are considered to be mandatory. If the type of property is omitted it will be inferred from the default value. The default type of a property is *java.lang.String* so if no default value is available, the property is mandatory and of type *String*.

There are four types of values available in MWE2. One may either define a string, boolean or component literal or a reference to a previously defined property.

5.3.3 Component

The last part of a module is the root component. It defines the externally visible type of the module and may either be created from a Java type or from another module.



The type of the component can be derived in many cases except for the root component. That's why it's optional in the component literal. If no type is given, it will be inferred from the left side of the assignment. The assigned feature can either be a declared property of the module or a *set*- or *add*-method of a Java class.

Components can be named to make them referable in subsequent assignments. Following the *':'* keyword, one can define an identifier for the instantiated component. The identifier is locally visible in the module and any assignment that is defined after the named component can refer to this identifier and thereby point to exactly the instantiated object.

The next option for a component is *auto-inject*. If this modifier is set on a component, any available feature of the component that has the same name as a property or previously created named component will be automatically assigned.

The core of a component is the list of assignments between the curly braces. An

arbitrary number of values can be set on the component by means of feature-to-value pairs.

!{width:50%}images/mwe2/assignment.png!

The available constructs on the right hand side of the assignment are the same as for default values for properties.

5.3.4 String Literals

String values are likely to be the most used literals in MWE2. There is a convenient syntax for string concatenation available due to the high relevance in a description object composition and configuration language. MWE2 strings are multi-line strings and can be composed of several parts.

```
var aString = 'a␣value'
var anotherString = 'It␣is␣possible␣to␣embed␣${aString}␣into
␣a␣multi−line␣string'
```

This is especially convenient for path-substitution if one defines e.g. a common root directory and wants to specify other paths relative to the base.

There are two different delimiters available for strings. Users are free to either use single- or double-quotes to start and end strings. If a certain string contains a lot of single-quotes one would better choose double-quotes as delimiter and vice versa. There is no semantic difference between both notations.

The escape character in MWE2 is the back-slash "\". It can be used to write line-breaks or tabular characters explicitly and to escape the beginning of substitution variables *${* and the quotes itself. Allowed escape sequences are:

```
\n .. line break
\r .. carriage return
\t .. tabular character
\'␣␣..␣single−quote␣(can␣be␣omitted␣in␣double−quoted␣strings)
\"␣␣..␣double−quote␣(can␣be␣omitted␣in␣single−quoted␣strings)
\${␣..␣escape␣the␣substitution␣variable␣start␣${
\\␣␣..␣the␣back−slash␣itself
```

Other escape sequence are illegal in MWE2 strings.

5.3.5 Boolean Literals

MWE2 has native support for the boolean type. The literals are *true* and *false*.

5.3.6 References

Each assigned value in MWE2 either as default for properties or in a component assignment can be a reference to a previously declared property or named component. The can be referenced intuitively by their name.

6 IDE Concepts

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

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

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

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

AbstractRule:
  RuleA | RuleB
;

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

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

6.1 Label Provider

There are various places in the UI in which model elements have to be presented to the user: In the outline view (§6.5), in hyper links (§6.6), in content proposals (§6.2), find dialogs etc. Xtext allows to customize each of these appearances by individual *ILabelProviders*.

An *ILabelProvider* has two methods: *getText(Object)* returns the text in an object's label, while *getImage(Object)* returns the icon. In addition, the Eclipse UI framework offers the *IStyledLabelProvider*, which returns a styled string (i.e. with custom fonts, colors etc.) in the *getStyledText(Object)* method.

Almost all label providers in the Xtext framework inherit from the base class org.eclipse.xtext.ui.label.AbstractI which unifies both approaches. Subclasses can either return a styled string or a string in the *doGetText(Object)* method. The framework will automatically convert it to a styled text (with default styles) or to a plain text in the respective methods.

Dealing with images can be cumbersome, too, as image handles tend to be scarce system resources. The org.eclipse.xtext.ui.label.AbstractLabelProvider helps you managing the images: In your implementation of *doGetImage(Object)* you can as well return an *Image* or a string, representing a path in the *icons/* folder of the containing plug-in. This path is actually configurable by Google Guice. Have a look at the org.eclipse.xtext.ui.PluginImageHelper to learn about the customizing possibilities.

If you have the org.eclipse.xtext.ui.generator.labeling.LabelProviderFragment in the list of generator fragments in the MWE2 workflow for your language, it will automatically create stubs and bindings for an _{MyLang}EObjectLabelProvider_ (§6.1.1) and an _{MyLang}DescriptionLabelProvider_ (§6.1.2) which you can implement by manually.

6.1.1 Label Providers For EObjects

The first set of label providers refers to actually loaded and thereby available model elements. By default, Xtext binds the *DefaultEObjectLabelProvider* to all use cases, but you can change the binding individually for the Outline, Content Assist or other places. For that purpose, there is a so called *binding annotation* for each use case. For example, to use a custom *MyContentAssistLabelProvider* to display elements in the content assist, you have to override

```
@Override
public void configureContentProposalLabelProvider(Binder binder) {
  binder.bind(ILabelProvider.class)
    .annotatedWith(ContentProposalLabelProvider.class)
    .to(MyContentAssistLabelProvider.class);
}
```

p.in your language's␣UI␣module.

If your grammar uses an imported *EPackage*, there may be an existing *.edit* plug-in generated by EMF that also provides label providers for model elements. To use this as a fallback, your label provider should call the constructor with the delegate parameter and use DI for the initialization, e.g.

```
public class MyLabelProvider {
    @Inject
    public MyLabelProvider(AdapterFactoryLabelProvider delegate) {
        super(delegate);
    }
    ...
}
```

DefaultEObjectLabelProvider

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 *DefaultEObjectLabelProvider*.

```
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";
  }
}
```

What is especially nice about the default implementation is the actual reason for its class name: It provides very reasonable defaults. To compute the label for a certain model element, it will at first have a look for an EAttribute *name* and try to use this one. If it cannot find such a feature, 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.

You can a also customize error handling by overriding the methods *handleTextError()* or *handleImageError()*.

6.1.2 Label Providers For Index Entries

Xtext maintains an index of all model elements to allow quick searching and linking without loading the referenced resource (See the chapter on index-based scopes (§4.6.1) for details). The elements from this index also appear in some UI contexts, e.g. in the *Find model elements* dialog or in the *Find references* view. For reasons of scalability, the UI should not automatically load resources, so we need another implementation of a label provider that works with the elements from the index, i.e. *IResourceDescription*, *IEObjectDescription*, and *IReferenceDescription*.

The default implementation of this service is the org.eclipse.xtext.ui.label.DefaultDescriptionLabelProvider. It employs the same polymorphic dispatch mechanism as the DefaultEObjectLabelProvider (§6.1.1). The default text of an *EObjectDescription* is its indexed name. The image is resolved by dispatching to *image(EClass)* with the *EClass* of the described object. This is likely the only method you want to override. *IResourceDescriptions* will be represented with their path and the icon registered for your language's editor.

To have a custom description label provider, make sure it is bound in your UI module:

```
public void configureResourceUIServiceLabelProvider(Binder binder) {
binder.bind(ILabelProvider.class) .annotatedWith(ResourceServiceDescriptionLabelProvider.class)
.to(MyCustomDefaultDescriptionLabelProvider.class); }
```

6.2 Content Assist

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

revise indentation levels

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

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

6.2.1 ProposalProvider

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

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

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

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

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

6.2.2 Sample Implementation

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

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

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

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

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

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

6.3 Quick Fixes

For validations written using the AbstractDeclarativeValidator (§4.4.2) it is possible to provide corresponding quick fixes in the editor. To be able to implement a quick fix for a given diagnostic (a warning or error) the underlying *cause* of the diagnostic must be known (i.e. what actual problem does the diagnostic represent?), otherwise the fix doesn't know what needs to be done. As we don't want to deduce this from the diagnostic's error message we associate a problem specific *code* with the diagnostic.

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

```
warning("Name should start with a capital",
```

```
    DomainmodelPackage.TYPE__NAME, INVALID_TYPE_NAME, type.getName());
```

Now that the validation has a unique code identifying the problem we can register quick fixes for it. We start by adding the org.eclipse.xtext.ui.generator.quickfix.QuickfixProviderFragment to our workflow and after regenerating the code we should find an empty class *MyDslQuickfixProvider* in our DSL's UI project and new entries in the *plugin.xml_gen* file.

Continuing with the *INVALID_TYPE_NAME* problem from the Domainmodel example we add a method with which the problem can be fixed (have a look at the *DomainmodelQuickfixProvider* for details):

```
@Fix(DomainmodelJavaValidator.INVALID_TYPE_NAME)
public void fixName(final Issue issue, IssueResolutionAcceptor acceptor) {
  acceptor.accept(issue,
    "Capitalize␣name", // quick fix label
    "Capitalize␣name␣␣of␣'" + issue.getData()[0] + "'", // description
    "upcase.png", // quick fix icon
    new IModification() {
      public void apply(IModificationContext context)
                                              throws BadLocationException {
        IXtextDocument xtextDocument = context.getXtextDocument();
        String firstLetter = xtextDocument.get(issue.getOffset(), 1);
        xtextDocument.replace(issue.getOffset(), 1,
                              Strings.toFirstUpper(firstLetter));
      }
    }
  );
}
```

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

The first three parameters given to the org.eclipse.xtext.ui.editor.quickfix.IssueResolutionAcceptor define the UI representation of the quick fix. As the document is not necessarily loaded when the quick fix is offered, we need to provide any additional data from the model that we want to refer to in the UI when creating the issue in the validator above. In this case, we provided the existing type name. The additional data is available as *Issue.getData()*. As it is persisted in markers, only strings are allowed.

The actual model modification is implemented in the org.eclipse.xtext.ui.editor.model.edit.IModification. The org.eclipse.xtext.ui.editor.model.edit.IModificationContext provides access to the erroneous document. In this case, we're using Eclipse's *IDocument* API to replace a text region.

If you prefer to implement the quick fix in terms of the semantic model use a org.eclipse.xtext.ui.editor.model.ed instead. Its *apply(EObject, IModificationContext)* method will be invoked inside a modify-transaction and the first argument will be the erroneous semantic element. This makes it very easy for the fix method to modify the model as necessary. After the method returns the model as well as the Xtext editor's content will be updated accordingly. If the method fails (throws an exception) the change will not be committed. The following snippet shows a semantic quick fix for a similar problem.

```
@Fix(DomainmodelJavaValidator.INVALID_FEATURE_NAME)
public void fixFeatureName(final Issue issue,
                           IssueResolutionAcceptor acceptor) {
  acceptor.accept(issue,
    "Uncapitalize name", // label
    "Uncapitalize name of '" + issue.getData()[0] + "'", // description
    "upcase.png", // icon
    new ISemanticModification() {
      public void apply(EObject element, IModificationContext context) {
        ((Feature) element).setName(
            Strings.toFirstLower(issue.getData()[0]));
      }
    }
  );
}
```

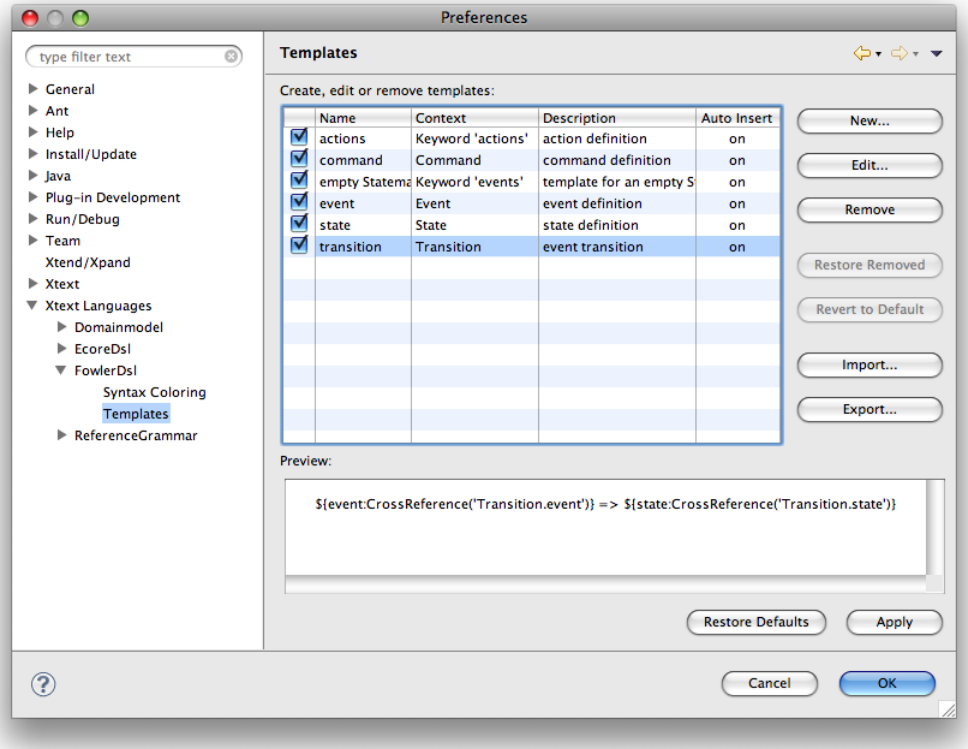
6.3.1 Quickfixes for Linking Errors and Syntax Errors

You can even define quick fixes for linking errors. The issue codes are assigned by the org.eclipse.xtext.linking.ILinkingDiagnosticMessageProvider. Have a look at the domain model example how to add quick fixes for these errors.

Analogously, there is the org.eclipse.xtext.parser.antlr.ISyntaxErrorMessageProvider to assign issue codes to syntactical errors.

6.4 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 named *templates.xml* inside the *templates* directory of the generated UI-plug-in. This file contains templates in a format as described in the Eclipse online help .



By default Xtext registers *ContextTypes* for each rule (*[languageName].[RuleName]*) and for each keyword (*[languageName].kw_[keyword]*). If you don't like these defaults you'll have to subclass org.eclipse.xtext.ui.editor.templates.XtextTemplateContextTypeRegistry and configure it via Guice (§3.2.1).

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

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

6.4.1 CrossReference TemplateVariableResolver

Xtext comes with a specific template variable resolver *TemplateVariableResolver* called org.eclipse.xtext.ui.editor.templates.CrossReferenceTemplateVariableResolver, which can be used to place cross-references within a template.

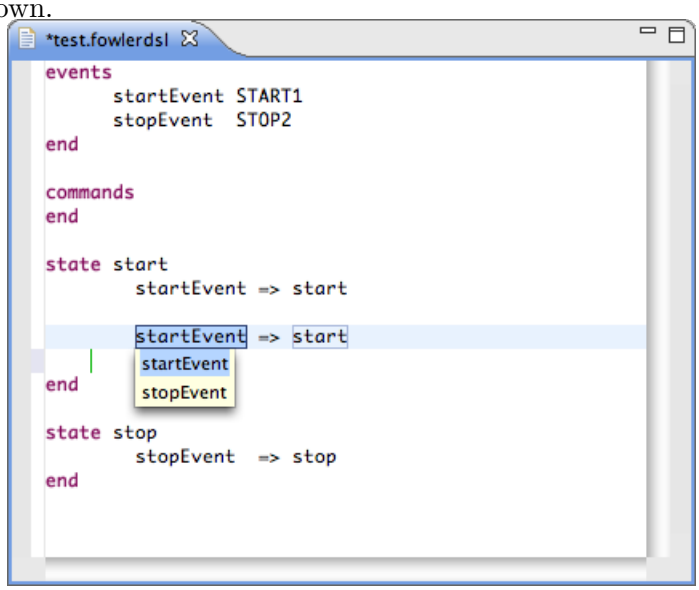
The syntax is as follows:

${<displayText>:CrossReference([<MyPackageName>.]<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.



6.4.2 Enumeration TemplateVariableResolver

The org.eclipse.xtext.ui.editor.templates.EnumTemplateVariableResolver resolves a template variable to *EEnumLiteral* literals which are assignment-compatible to the enumeration type declared as the first parameter of the the *Enum TemplateVariable*.

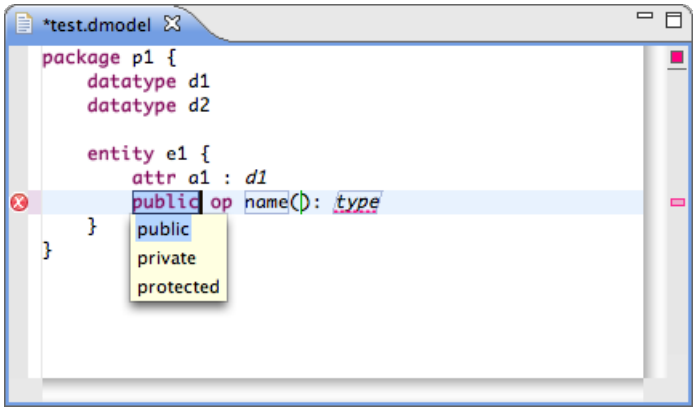
The syntax is as follows:

${<displayText>:Enum([<MyPackage>.]<EnumType>)

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

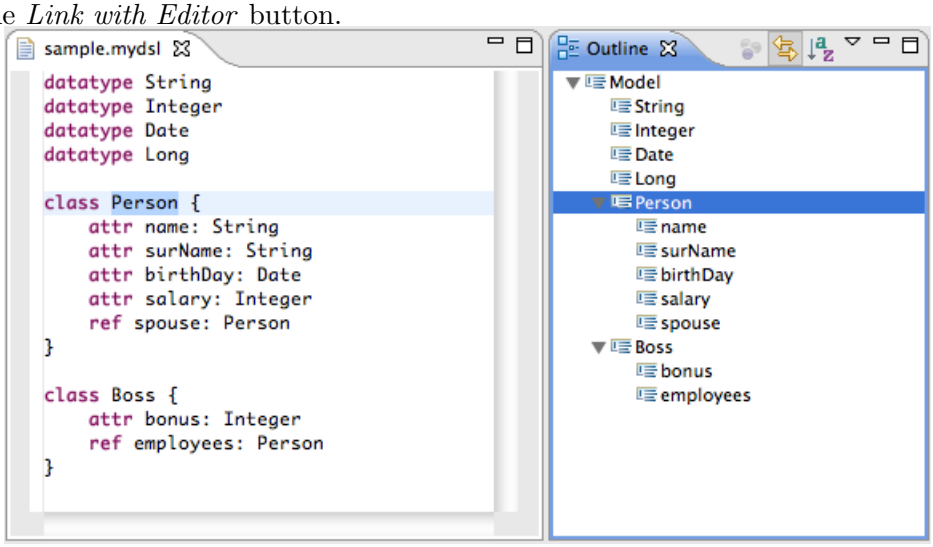
```
<template name="Operation" description="template␣for␣an␣Operation"
  id="org.eclipse.xtext.example.Domainmodel.Operation"
  context="org.eclipse.xtext.example.Domainmodel.Operation"
  enabled="true">
    ${visibility:Enum('Visibility')} op ${name}(${cursor}):
        ${type:CrossReference('Operation.type')}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

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



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



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 nodes, you can customize the outline by implementing your own org.eclipse.xtext.ui.editor.outline.IOutlineTreeProvider.

If your workflow defines the org.eclipse.xtext.ui.generator.outline.OutlineTreeProviderFragment, Xtext generates a stub for your own org.eclipse.xtext.ui.editor.outline.IOutlineTreeProvider that allows you to customize every aspect of the outline by inheriting the powerful customization methods of org.eclipse.xtext.ui.editor.outline.impl.DefaultOutlineTreeProvider. The following sections show how to do fill this stub with life.

6.5.1 Influencing the outline structure

Each node the outline tree is an instance of org.eclipse.xtext.ui.editor.outline.IOutlineNode. The outline tree is always rooted in a org.eclipse.xtext.ui.editor.outline.impl.DocumentRootNode. This node is automatically created for you. Its children are the root nodes in the displayed view.

An org.eclipse.xtext.ui.editor.outline.impl.EObjectNode represents a model element. By default, Xtext creates an org.eclipse.xtext.ui.editor.outline.impl.EObjectNode for each model element in the node of its container. Nodes are created by calling the method *createNode(parentNode, modelElement)* which delegates to *createEObjectNode(...)* if not specified differently.

To change the children of specific nodes, you have to implement the method *_createChildren(parentNode, parentModelElement)* with the appropriate types. The following snippet shows you how to skip the root model element of type *DomainModel* in the outline of our domain model example:

```
protected void _createChildren(DocumentRootNode parentNode,
                               DomainModel domainModel) {
    for (AbstractElement element : domainModel.getElements()) {
        createNode(parentNode, element);
    }
}
```

You can choose not to create any node in the *_createChildren()* method. Because the outline nodes are calculated on demand, the UI will show you an expandable node that doesn't reveal any children if expanded. This might be confuse your users a bit. To overcome this shortcoming, you have to implement the method *_isLeaf(modelElement)* with the appropriate argument type, e.g.

```
// feature nodes are leafs and not expandable
protected boolean _isLeaf(Feature feature) {
    return true;
}
```

Xtext provides a third type of node: org.eclipse.xtext.ui.editor.outline.impl.EStructuralFeatureNode. It is used to represent a feature of a model element rather than element itself. The following simplified snippet from Xtend2 illustrates how to use it:

```
protected void _createChildren(DocumentRootNode parentNode,
                               XtendFile xtendFile) {
    // show a node for the attribute XtendFile.package
    createEStructuralFeatureNode(parentNode,
        xtendFile,
        Xtend2Package.Literals.XTEND_FILE__PACKAGE,
        getImageForPackage(),
        xtendFile.getPackage(),
        true);
    // show a container node for the list reference XtendFile.imports
    // the imports will be shown as individual child nodes automatically
    createEStructuralFeatureNode(parentNode,
```

```
        xtendFile,
        Xtend2Package.Literals.XTEND_FILE__IMPORTS,
        getImageForImportContainer(),
        "import declarations",
        false);
    createEObjectNode(parentNode, xtendFile.getXtendClass());
}
```

Of course you can add further custom types of nodes. For consistency, make sure to inherit from org.eclipse.xtext.ui.editor.outline.impl.AbstractOutlineNode. To instantiate these, you have to implement *_createNode(parentNode, semanticElement)* with the appropriate parameter types.

6.5.2 Styling the outline

You can also customize the icons and texts for an outline node. By default, Xtext uses the label provider (§6.1) of your language. If you want the labels to be specific to the outline, you can override the methods *_text(modelElement)* and *_image(modelElement)* in your org.eclipse.xtext.ui.editor.outline.impl.DefaultOutlineTreeProvider.

Note that the method *_text(modelElement)* can return a java.lang.String or a org.eclipse.jface.viewers.StyledStrin
The org.eclipse.xtext.ui.label.StylerFactory can be used to create org.eclipse.jface.viewers.StyledStrings, like in the follwoing example:

```
@Inject
private StylerFactory stylerFactory;

public Object _text(Entity entity) {
    if(entity.isAbstract()) {
        return new StyledString(entity.getName(),
            stylerFactory
                .createXtextStyleAdapterStyler(getTypeTextStyle())));
    else
        return entity.getName();
}

protected TextStyle getTypeTextStyle() {
    TextStyle textStyle = new TextStyle();
    textStyle.setColor(new RGB(149, 125, 71));
    textStyle.setStyle(SWT.ITALIC);
    return textStyle;
}
```

To access images we recommend to use the org.eclipse.xtext.ui.PluginImageHelper.

6.5.3 Filtering actions

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 a filter action to your outline. Filter actions must extend org.eclipse.xtext.ui.editor.outline.actions.AbstractFilterOutlineContribution to ensure that

the action toggle state is handled correctly. Here is an example form our domain model example:

```
public class FilterOperationsContribution
        extends AbstractFilterOutlineContribution {

    public static final String PREFERENCE_KEY =
        "ui.outline.filterOperations";

    @Inject
    private PluginImageHelper imageHelper;

    @Override
    protected boolean apply(IOutlineNode node) {
        return !(node instanceof EObjectNode)
                || !((EObjectNode) node).getEClass()
                    .equals(DomainmodelPackage.Literals.OPERATION);
    }

    @Override
    public String getPreferenceKey() {
        return PREFERENCE_KEY;
    }

    @Override
    protected void configureAction(Action action) {
        action.setText("Hide operations");
        action.setDescription("Hide operations");
        action.setToolTipText("Hide operations");
        action.setImageDescriptor(getImageDescriptor());
    }

    protected ImageDescriptor getImageDescriptor(String imagePath) {
        return ImageDescriptor.createFromImage(
            imageHelper.getImage("Operation.gif"));
    }


}
```

The contribution must be bound in the *MyDslUiModule* like this

```
public void configureFilterOperationsContribution(Binder binder) {
    binder
        .bind(IOutlineContribution.class).annotatedWith(
            Names.named("FilterOperationsContribution"))
        .to(FilterOperationsContribution.class);
}
```

6.5.4 Sorting actions

Xtext already adds a sorting action to your outline. By default, nodes are sorted lexically by their text. You can change this behavior by binding your own org.eclipse.xtext.ui.editor.outline.impl.OutlineFilte

A very common use case is to group the children by categories first, e.g. show the imports before the types in a package declaration, and sort the categories separately. That is why the org.eclipse.xtext.ui.editor.outline.actions.SortOutlineContribution\$DefaultComparator has a method *getCategory(IOutlineNode)* that allows to specify such categories. The example shows how to use such categories:

```
public class MydslOutlineNodeComparator extends DefaultComparator {
    @Override
    public int getCategory(IOutlineNode node) {
        if (node instanceof EObjectNode)
            switch((EObjectNode) node).getEClass().getClassifierID())) {
            case MydslPackage.TYPE0:
                return −10;
            case MydslPackage.TYPE1:
                return −20;
            }
        return Integer.MIN_VALUE;
    }
}
```

As always, you have to declare a binding for your custom implementation in your *MydslUiModule*:

```
@Override
public Class<? extends IComparator>
    bindOutlineFilterAndSorter$IComparator() {
        return MydslOutlineNodeComparator.class;
}
```

6.5.5 Quick Outline

Xtext also provides a quick outline: If you press CTRL-O in an Xtext editor, the outline of the model is shown in a popup window. The quick outline also supports drill-down search with wildcards. To enable the quick outline, you have to put the org.eclipse.xtext.ui.generator.outline.QuickOutlineFragment into your workflow.

6.6 Hyperlinking

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

6.6.1 Location Provider

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

revise indentation levels

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

Customized Location Provider

As the default strategy is a best effort it may not always result in the selection you want. If that's the case you can override (§3.2.1) the *ILocationInFileProvider* binding in the UI module as in the following example:

```
public class MyDslUiModule extends AbstractMyDslUiModule {

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

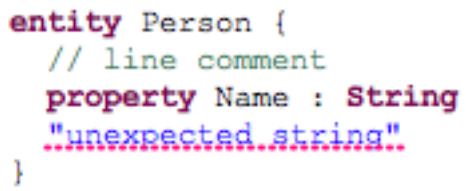
Often the default strategy only needs some guidance (e.g. selecting the text corresponding to another feature than *name*). In that case you can simply subclass *DefaultLocationInFileProvider* and override the methods *getIdentifierFeature()* and / or *useKeyword()* to guide the first and last steps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

6.6.2 Customizing Available Hyperlinks

The hyperlinks are provided by the org.eclipse.xtext.ui.editor.hyperlinking.HyperlinkHelper which will create links for cross-referenced objects by default. Clients may want to override *createHyperlinksByOffset(XtextResource, int, IHyperlinkAcceptor)* to provide additional links or supersede the default implementation.

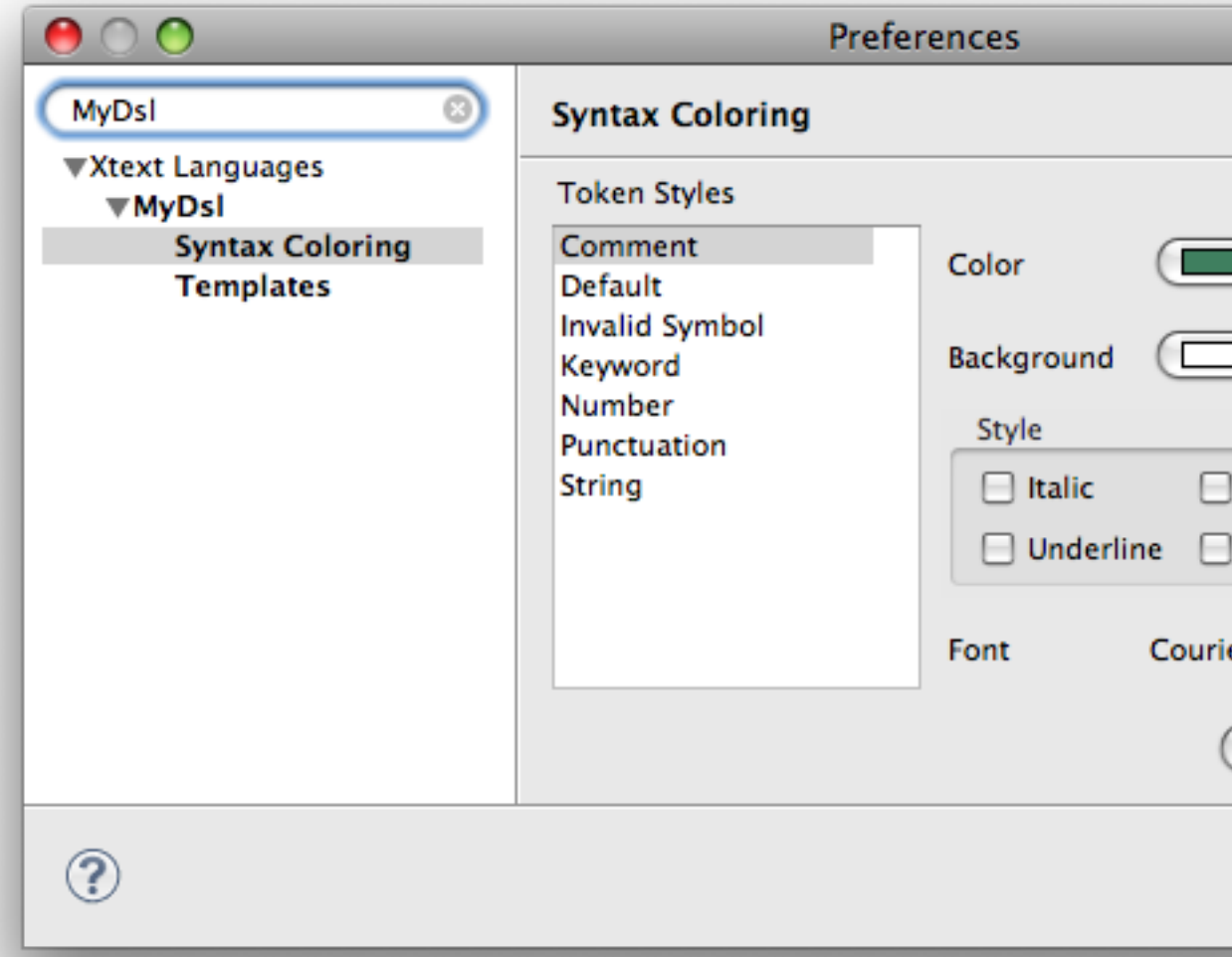
6.7 Syntax Coloring

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



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

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



6.7.1 Lexical Highlighting

The lexical highlighting can be customized by providing implementations of the interface org.eclipse.xtext.ui.editor.syntaxcoloring.IHighlightingConfiguration and the abstract class org.eclipse.xtext.ui.editor.syntaxcoloring.AbstractTokenScanner. The latter fulfills the interface *ITokenScanner* from the underlying JFace Framework, which may be implemented by clients directly.

The org.eclipse.xtext.ui.editor.syntaxcoloring.IHighlightingConfiguration is used to register any default style without a specific binding to a pattern in the model file. It is used to populate the preferences page and to initialize the *ITextAttributeProvider*, which in turn is the component that is used to obtain the actual settings for a style's id. An implementation

will usually be very similar to the org.eclipse.xtext.ui.editor.syntaxcoloring.DefaultHighlightingConfiguration and read like this:

```
public class DefaultHighlightingConfiguration
    implements IHighlightingConfiguration {

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

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

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

  // ...
}
```

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

6.7.2 Semantic Highlighting

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

As for the lexical highlighting the interface to register the available styles is the org.eclipse.xtext.ui.editor.syntaxcoloring.IHighlightingConfiguration. The org.eclipse.xtext.ui.editor.syntaxcoloring.ISemant is the primary hook to implement the logic that will compute to-be-highlighted ranges based on the model elements.

The framework will pass the current XtextResource and an org.eclipse.xtext.ui.editor.syntaxcoloring.IHighlighted to the calculator. It is ensured, that the resource will not be altered externally until the

called method *provideHighlightingFor()* returns. However, the resource may be null. The implementor's task is to navigate your semantic model and compute various ranges based on the attached node information and associate styles with them. This may read similar to the following snippet:

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

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

This example refers to an implementation of the *IHighlightingConfiguration* that registers a style for a cross-reference. It is pretty much the same implementation as for the previously mentioned sample of a lexical *IHighlightingConfiguration*.

```
public class HighlightingConfiguration
    implements IHighlightingConfiguration {

  // lexical stuff goes here
  // ..
  public final static String CROSS_REF = "CrossReference";

  public void configure(IHighlightingConfigurationAcceptor acceptor) {
    // lexical stuff goes here
    // ..
    acceptor.acceptDefaultHighlighting(CROSS_REF,
      "Cross-References", crossReferenceTextStyle());
  }

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

The implementor of an *IHighlightingCalculator* should be aware of performance to ensure a good user experience. It is probably not a good idea to traverse everything of your model when you will only register a few highlighted ranges that can be found easier with some typed method calls. It is strongly advised to use purposeful ways to navigate your model. The parts of Xtext's core that are responsible for the semantic highlighting

are pretty optimized in this regard as well. The framework will only update the ranges that actually have been altered, for example. This speeds up the redraw process. It will even move, shrink or enlarge previously announced regions based on a best guess before the next semantic highlighting pass has been triggered after the user has changed the document.

6.8 Project Wizard

Optionally, Xtext can generate a *New Project Wizard* for your DSL. Using this wizard a user can create a new project with only a few clicks that is configured with the right dependencies and natures. By default it contains a sample model file and workflow that serve as a scaffold for the user. Furthermore it has the Xtext nature and thereby the builder assigned. You enable the generation of the project wizard by adding the *SimpleProjectWizardFragment* fragment to the workflow (§3.1.2):

```
import org.eclipse.xtext.ui.generator.*

// project wizard fragment
fragment = projectWizard.SimpleProjectWizardFragment {
  generatorProjectName = "${projectName}.generator"
  modelFileExtension = file.extensions
}
```

Here

revise indentation levels

- the *generatorProjectName* is used to specify the project that contains the workflows and templates used to generate artifacts from your DSL. The generated project wizard uses this to add a corresponding dependency to the created project.
- and the *modelFileExtension* specifies the default file extension associated with your DSL. When the user clicks the *Finish* button the project wizard will look for a file with the given extension in the created project and open a editor.

After running the Xtext generator, the DSL's UI project will contain a new Xpand template *MyDslNewProject.xpt* in the *src* folder in the package *[base-package].ui.wizard*. Note: It may be necessary to manually merge the new entry in the *plugin.xml_gen* into your *plugin.xml* of the UI project to enable the wizard contribution.

The generated Xpand template will be expanded by the project wizard when the user clicks the *Finish* button and it is responsible for initializing the project with some sample content. When finishing the wizard the template will be used to create a sample model file and a simple workflow to run the model through the generator project's *MyDslGenerator.mwe* workflow. However, this is only a pragmatic default. As the Xpand template is in the *src* source folder you may of course modify it to generate whatever initial content you want for a new project. Just make sure to leave the top-level *main* definition in place, as that is the interface for the project wizard.

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

6.8.1 Customizing the Project Wizard

To further customize the creation of the project you can implement your own *project creator*. The default project creator is represented by the generated class *MyDslProjectCreator* in the *src-gen* folder. It is highly extensible. Without any changes it will simply create a new plug-in project with the Xtext nature assigned. Afterwards it will execute the *main* definition of the Xpand template as described above.

To add more pages or input fields to the project wizard you should subclass the class *MyDslNewProjectWizard* as appropriate. Don't forget to register the subclass in the UI project's *MyDslUiModule*. You may also want to make additionally entered user data available to the Xpand template. In this case you should enhance the *MyDslProjectInfo* to allow that one to hold the information. This is the context object which gets passed to the template when it's executed. Don't forget that your specialized *MyDslNewProjectWizard* has to populate the data fields of your *MyDslProjectInfo*.

7 Referring to Java Types

A common use case when developing languages is the requirement to refer to existing concepts of other languages. Xtext makes this very easy for other self defined languages. However, it is often very useful to have access to the available types of the Java Virtual Machine. The JVM types Ecore model enables clients to do exactly this. It is possible to create cross-references to classes, interfaces, and their fields and methods. Basically every information about the structural concepts of the Java type system is available via the JVM types. This includes annotations and their specific values and enumeration literals as well.

The implementation will be selected transparently to the client code depending on where the code is executed. If the environment is a plain stand-alone Java or OSGi environment, the *java.lang.reflect* API will be used to deduce the necessary data. On the contrary, the type-model will be created from the live data of the JDT in an interactive Eclipse environment. All this happens behind the scenes via different implementations that are bound to specific interfaces with Google Guice.

7.1 How to Use Java Types

Using the JVM types model is very simple. First of all, the grammar has to import the *JavaVMTypes* Ecore model. Thanks to content assist this is easy to spot in the list of proposals.

```
import "http://www.eclipse.org/xtext/common/JavaVMTypes" as types
```

The next step is to actually refer to an imported concept. Let's define a mapping to actually available Java types for the simple data types in the self defined language. By means of cross-references this works as one got already used to when dealing with references in Xtext.

```
Type:
  'type' name=ID 'mapped-to' javaType=[types::JvmType|FQN];
```

Last but not least, the org.eclipse.xtext.generator.types.TypesGeneratorFragment has to be added to the workflow. The safest way is to add it after the actually used scoping fragments as a specialized version of the *IGlobalScopeProvider* will be configured. Don't forget to refer to the *genmodel* of the Java VM types. The shortest possible URI is a classpath-URI.

```
fragment = ecore.EcoreGeneratorFragment {
  referencedGenModels="classpath:/model/JavaVMTypes.genmodel"
}
```

```
...
// scoping and exporting API
...

fragment = types.TypesGeneratorFragment {}
```

After regenerating your language, it will be allowed to define a type *Date* that maps to the *java.util.Date* like this:

```
type Date mapped-to java.util.Date
```

These two steps will provide a nice integration into the Eclipse JDT. There is *Find References* on Java methods, fields and types that will reveal results in your language files. *Go To Declaration* works as expected and content assist will propose the list of available types.

7.2 Customization Points

There are several customization hooks in the runtime layer and on the editor side as well:

The org.eclipse.xtext.common.types.xtext.AbstractTypeScopeProvider can be used to create scopes for members with respect to the override semantics of the Java language. Of course it is possible to use this implementation to create scopes for types as well.

As the Java VM types expose a lot of information about visibility, parameter- and return types, generic, available annotations or enumeration literals, it is very easy to define constraints for the referred types.

The org.eclipse.xtext.common.types.xtext.ui.ITypesProposalProvider can be used to provide optimized proposals based on various filter criteria. The most common selector can be used directly via *createSubTypeProposals*. The implementation is optimized and uses the JDT Index directly to minimize the effort for object instantiation. The class org.eclipse.xtext.common.types.xtext.ui.TypeMatchFilters provides a comprehensive set of reusable filters that can be easily combined to reduce the list of proposals to a smaller number of valid entries.

```
import static org.eclipse.xtext.common.types.xtext.ui.TypeMatchFilters.*;
..
..

  proposalProvider.createSubTypeProposals(factory, context,
      and(not(canInstantiate()), isPublic()), acceptor);
```

Since Xtext 1.0.1:

Proposals can be created by means of a scoped proposal provider that is actually aware of existing import statements and scoping rules while still leveraging the JDT index for faster proposal creation. Use the *ITypesProposalProvider.IScopableTypesProposalProvider* to create a scoped proposal provider and use this one instead of the plain implementation.

```
@Override
public void complete<TypeName>_<ReferenceName>(
```

```
    EObject model,
    Assignment assignment,
    ContentAssistContext context,
    ICompletionProposalAcceptor acceptor) {
  IScope scope = getScopeProvider().getScope(
      model,
      <MyDsl>Package.Literals.<POINTER_TO_REFERENCE>);
  ITypesProposalProvider scopedProposalProvider =
      scopableProposalProvider.getScopedProposalProvider(model, scope);
  scopedProposalProvider.create...
}
```

8 Typical Language Configurations

8.1 Case Insensitive Languages

In some cases, e.g. if your *SHIFT* key is broken, you might want to design a case insensitive language. Xtext offers separate generator fragments (§3.1.2) for this purpose.

For case insensitive keywords, open your MWE workflow and replace the Antlr related fragments:

```
// The antlr parser generator fragment.
fragment = parser.antlr.XtextAntlrGeneratorFragment {
// options = {
//  backtrack = true
// }
}
...

// generates a more lightweight Antlr parser and lexer tailored ...
fragment = parser.antlr.XtextAntlrUiGeneratorFragment {
}
```

with

```
// The antlr parser generator fragment.
fragment = parser.antlr.ex.rt.AntlrGeneratorFragment {
    options = {
        ignoreCase = true
    }
}
...

// generates a more lightweight Antlr parser and lexer tailored ...
fragment = parser.antlr.ex.ca.ContentAssistParserGeneratorFragment {
    options = {
        ignoreCase = true
    }
}
```

For case insensitve element names, use the *ignoreCase* option in your scope fragment, i.e.

```
fragment = scoping.ImportNamespacesScopingFragment {
    ignoreCase = true
}
```

or if you are using importURI based global scopes (§4.6.1)

```
fragment = scoping.ImportURIScopingFragment {
    ignoreCase = true
}
```

8.2 Languages Independent of JDT

The following section describes how you make your language independent of Eclipse's Java Development Toolkit (JDT).

In the *UIModule* of your language you have to overwrite some bindings. First, remove the bindings to components with support for the '_classpath:_' URI protocol, i.e.

```
@Override
public Class<? extends IResourceForEditorInputFactory>
    bindIResourceForEditorInputFactory() {
        return ResourceForIEditorInputFactory.class;
}

@Override
public Class<? extends IResourceSetProvider> bindIResourceSetProvider() {
    return SimpleResourceSetProvider.class;
}
```

Second, configure the global scope provider to scan project root folders instead of the classpath of Java projects.

```
@Override
public com.google.inject.Provider
    <org.eclipse.xtext.resource.containers.IAllContainersState>
    provideIAllContainersState() {
    return org.eclipse.xtext.ui.shared.Access.getWorkspaceProjectsState();
}
```

The remaining steps show you how to adapt the project wizard (§6.8) for your language, if you have generated one. The best way to do this is to create a new subclass of the generated *IProjectCreator* in the *src/* folder of the *ui* project and apply the necessary changes there. First, remove the JDT project configuration by overriding *configureProject* with an empty body.

The next thing is to redefine the project natures and builders that should be applied to you language projects.

In in this case just remove the JDT stuff in this way:

```
protected String[] getProjectNatures() {
    return new String[] {
        "org.eclipse.pde.PluginNature",
        "org.eclipse.xtext.ui.shared.xtextNature"
    };
}

protected String[] getBuilders() {
```

```
    return new String[] {
        "org.eclipse.pde.ManifestBuilder",
        "org.eclipse.pde.SchemaBuilder"
    };
}
```

After that you have to bind the new *IProjectCreator*

```
@Override
public Class<? extends IProjectCreator> bindIProjectCreator() {
    return JDTFreeMyDslProjectCreator.class;
}
```

That's all. Your language and its IDE should now no longer depend on JDT.

8.3 Parsing Expressions with Xtext

Parsing simple XML-like, structural languages with Xtext is a no-brainer. However, parsing nested expressions is often considered a bit more complicated. This is because they are more complicated due to their recursive nature and also because with Xtext you have to avoid left recursive parser rules. As the underlying parser (generated by Antlr) uses a top-down approach it would recurse endlessly if you had a left recursive grammar.

Let's have a look at parsing a simple arithmetic expression:

$2 + 20 * 2$

If you know EBNF a bit and wouldn't think about avoiding left recursion, operator precedence or associativity, you'ld probably write a grammar like this:

```
Expression :
  Expression '+' Expression |
  Expression '*' Expression |
  INT;
```

This grammar would be left recursive because the parser reads the grammar top down and left to right and would endlessly call the Expression rule without consuming any characters, i.e. altering the underlying state of the parser. While this kind of grammar can be written for bottom-up parsers, you'ld still have to deal with operator precedence in addition. That is define that a multiplication has higher precedence than an addition for example.

In Xtext you define the precedence implicitly when left-factoring such a grammar. Left-factoring means you get rid of left recursion by applying a certain technique, which we will show in the following.

So here is a left-factored grammar (not yet working with Xtext) for the expression language above :

```
Addition :
  Multiplication ('+' Multiplication)*;

Multiplication:
```

```
  NumberLiteral ('*' NumberLiteral)*;

NumberLiteral:
  INT;
```

As you can see the main difference is that we have three rules instead of one and if you look a bit closer you see, that there's a certain delegation pattern involved. The rule Addition doesn't call itself but calls Multiplication instead. The operator precedence is defined by the order of delegation. The later the rule is called the higher is its precedence. This is at least the case for the first two rules which are of a left recursive nature (but we've left-factored them now). The last rule is not left recursive which is why you can write them down without applying this pattern.

We should allow users to explicitly adjust precedence by adding parenthesis, e.g. write something like (2 + 20) * 2. So let's add support for that (note that the grammar is still not working with Xtext):

```
Addition :
  Multiplication ('+' Multiplication)*;

Multiplication:
  Primary ('*' Primary)*;

Primary :
  NumberLiteral |
  '(' Addition ')';

NumberLiteral:
  INT;
```

So once again: if you have some construct that recurses on the left hand side, you need to put it into the delegation chain according to their operator precedence. The pattern is always the same, the thing that recurses delegates to the rule with the next higher precedence.

8.3.1 Construction of an AST

Now that we know how to avoid left recursion, let's have a look at what the parser produces. In Xtext each rule returns some value. Parser rules return AST nodes (i.e. EObject instances), enum rules return enum literals and datatype rules as well as terminal rules return simple values like strings and the like (EDatatype in EMF jargon). Xtext can automatically infer whether some rule is a parser rule, i.e. constructs and returns an AST node, or if it is a datatype rule that returns a value. The grammar above only consists of datatype rules all of them returning plain strings. In order to construct an AST we need to add Assignments and Actions. But before we do that we need to talk about return types.

The return type of a rule can be specified explicitly using the 'returns' keyword but can be inferred if the type's name is the same as the rule's name. That is

```
  NumberLiteral : ... ;
```

is a short form of

NumberLiteral **returns** NumberLiteral : ... ;

However in the case of the expressions grammar above, all rules need to return the same type since they are recursive. So in order to make the grammar functional we need to add a common return type explicitly (but the grammar is still missing some bits):

```
Addition returns Expression:
  Multiplication ('+' Multiplication)*;

Multiplication returns Expression:
  Primary ('*' Primary)*;

Primary returns Expression:
  NumberLiteral |
  '(' Addition ')';

NumberLiteral:
  INT;
```

The AST type inference mechanism of Xtext will infer two types: Expression and NumberLiteral. Now we need to add assignments and Actions in order to store all the important information in the AST and to create reasonable subtypes for the two operations.

In the following you see the final fully working Xtext grammar:

```
Addition returns Expression:
  Multiplication ({Addition.left=current} '+' right=Multiplication)*;

Multiplication returns Expression:
  Primary ({Multiplication.left=current} '*' right=Primary)*;

Primary returns Expression:
  NumberLiteral |
  '(' Addition ')';

NumberLiteral:
  value=INT;
```

Let's go through the grammar as the parser would do it for the expression

(1 + 20) * 2

The parser always starts with the first rule (Addition). Therein the first element is an unassigned rule call to Multiplication which in turn calls Primary. Primary now has two alternatives. The first one is calling NumberLiteral which consists only of one assignment to a feature called 'value'. The type of 'value' has to be compatible to the return type of the INT rule.

But as the first token in the sample expression is an opening parenthesis '(' the parser will take the second alternative in Primary, consume the '(' and call the rule Addition. Now the value '1' is the lookahead token and again Addition calls Multiplication and

Multiplication calls Primary. This time the parser takes the first alternative because '1' was consumed by the INT rule (which btw. is a reused library terminal rule).

As soon as the parser hits an assignment it checks whether an AST node for the current rule was already created. If not it will create one based on the return type of the current rule, which is NumberLiteral. The Xtext generator created an EClass 'NumberLiteral' before which can now be instantiated. That type will also have a property called value of type int (actually of type EInt), which will get the value '1' set. This is what the Java equivalent would look like:

```
// value=INT
if (current == null)
 current = new NumberLiteral();
current.setValue(ruleINT());
...
```

Now that the rule has been completed the created EObject is returned to the calling rule Primary, which in turn returns the object unchanged to its own caller. Within Multiplication the call to Primary has been successfully parsed and returned an instance of NumberLiteral. The second part of the rule (everything within the parenthesis) is a so called group. The asterisk behind the closing parenthesis states that this part can be consumed zero or more times. The first token to consume in this part is the multiplication operator '*'. Unfortunately in the current situation the next token to accept is the plus operator '+', so the group is not consumed at all and the rule returns what they got from the unassigned rule call (the NumberLiteral).

In rule Addition there's a similar group but this time it expects the correct operator so the parser steps into the group. The first element in the group is a so called action. As Xtext grammars are highly declarative and bi-directional it is not a good idea to allow arbitrary expression within actions as it is usually the case with parser generators. Instead we only support two kinds of actions. This one will create a new instance of type Addition and assign what was the to-be-returned object to the feature left. In Java this would have been something like:

```
// Multiplication rule call
current = ruleMultiplication();
// {Addition.left=current}
Addition temp = new Addition();
temp.setLeft(current);
current = temp;
...
```

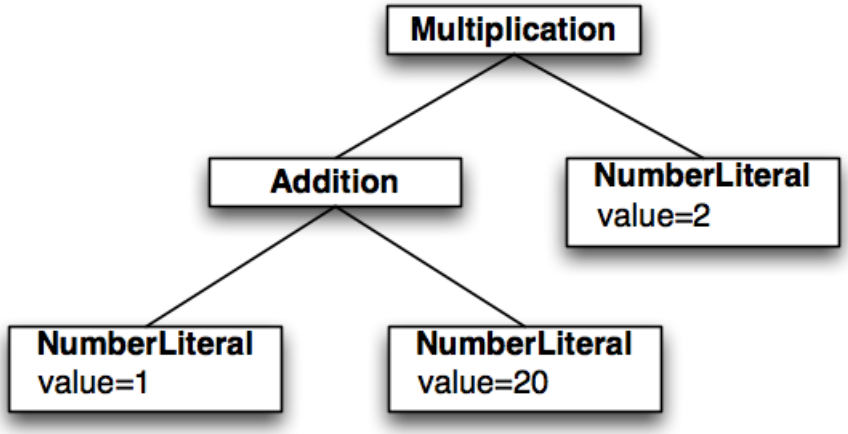
As a result the rule would now return an instance of Addition which has a NumberLiteral set to its property left. Next up the parser consumes the '+' operator. We do not store the operator in the AST because we have an explicit Addition type, which implicitly contains this information. The assignment ('right=Multiplication') calls the rule Multiplication another time and assigns the returned object (a NumberLiteral of value=20) to the property named right.

If we now had an additional plus operation '+' (e.g. 1 + 2 + 3) the group would match another time and create another instance of Addition. But we don't and therefore the

rule is completed and returns the created instance of Addition to its caller which was the second alternative in Primary. Now the closing parenthesis is matched and consumed and the parser stack is reduced once more.

We are now in rule Multiplication and have the multiplication operator '*' on the lookahead. The parser goes into the group and applies the action. Finally it calls the Primary rule, gets another instance of NumberLiteral (value=2), assigns it as the 'right' operand of the Multiplication and returns the Multiplication to the rule Addition which in turn returns the very same object as there's nothing left to parse.

The resulting AST looks like this:



(It's pretty hard to follow what's going on just by reading this text. Therefore we have prepared a small video which visualizes and explains the details. see http://vimeo.com/14358869)

8.3.2 Associativity

There is still one topic we should mention, which is associativity. There is left and right associativity as well as none associativity. In the example we have seen left associativity. Associativity tells the parser how to construct the AST when there are two infix operations with the same precedence. The following example is taken from the corresponding wikipedia entry:

Consider the expression a ˜ b ˜ c. If the operator ˜ has left associativity, this expression would be interpreted as (a ˜ b) ˜ c and evaluated left-to-right. If the operator has right associativity, the expression would be interpreted as a ˜ (b ˜ c) and evaluated right-to-left. If the operator is non-associative, the expression might be a syntax error, or it might have some special meaning. We already know the most important form which is left associativity:

```
Addition returns Expression:
  Multiplication ({Addition.left=current} '+' right=Multiplication)*;
```

Right associativity is done using the following pattern (note the quantity operator and the call to the rule itself at the end):

```
Addition returns Expression:
  Multiplication ({Addition.left=current} '+' right=Addition)?;
```

And if you don't want to allow multiple usages of the same expression in a row (hence non-associativity) you write:

```
Addition returns Expression:
  Multiplication ({Addition.left=current} '+' right=Multiplication)?;
```

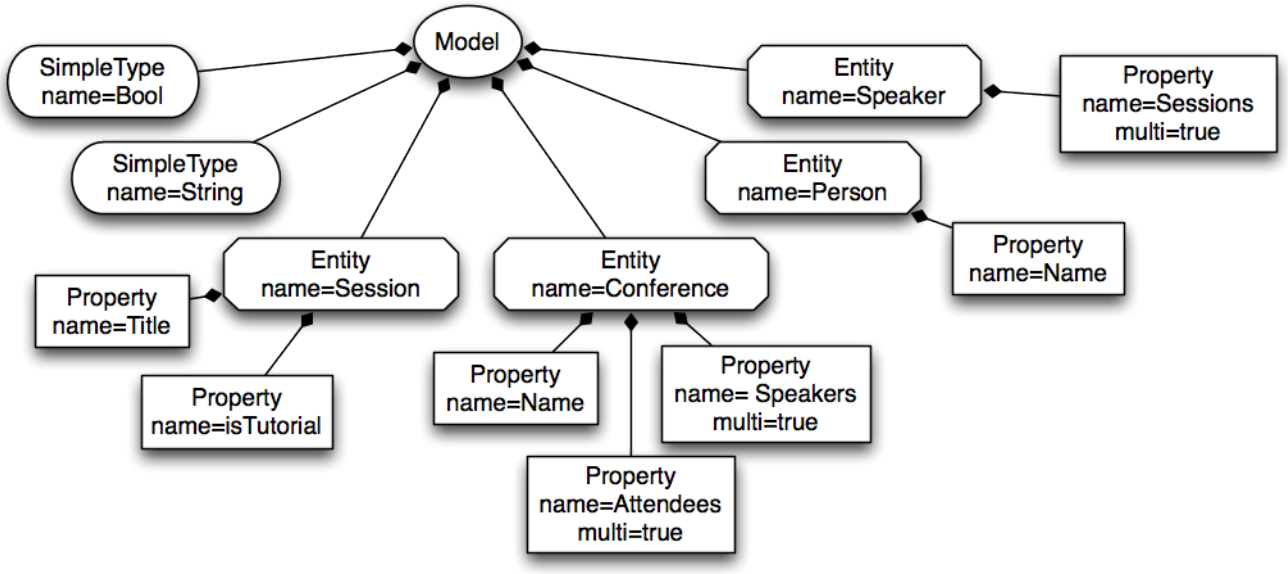
Note that sometimes it's better to allow associativity on parser level, but forbid it later using validation, because you can come up with a better error message. Also the whole parsing process won't be interrupted, so your tooling will generally be more forgiving.

9 Integration with EMF and Other EMF Editors

Xtext relies heavily on EMF internally, but it can also be used as the serialization backend of other EMF-based tools. In this section we introduce the basic concepts of the Eclipse Modeling Framework (EMF) in the context of Xtext. If you want to learn more about EMF, we recommend reading the EMF book

9.1 Model, Ecore Model, and Ecore

When Xtext uses EMF models as the in-memory representation of any parsed text files. This in-memory object graph is called the *Abstract Syntax Tree* (AST). Depending on the community this concepts is also called *document object graph (DOM)*, *semantic model*, or simply *model*. We use *model* and *AST* interchangeably. Given the example model from the introduction (§**??**), the AST looks similar to this



The *AST* should contain the essence of your textual models and abstract over syntactical information. It is used by later processing steps, such as validation, compilation or interpretation. In EMF a model is made up of instances of *EObjects* which are connected and an *EObject* is an instance of an *EClass*. A set of *EClasses* if contained in a so called *EPackage*, which are both concepts of *Ecore* In Xtext, meta models are

either inferred from the grammar or predefined by the user (see the section on package declarations (§2.2.2) for details). The next diagram shows the meta model of our example:

!images/metamodel.png(Sample meta model)!

The language in which the meta model is defined is called *Ecore*. In other words, the meta model is the Ecore model of your language. Ecore an essential part of EMF. Your your models instantiate the meta model, and your meta model instantiates Ecore. To put an end to this recursion, Ecore is defined in itself (an instance of itself).

The meta model defines the types of the semantic nodes as Ecore *EClasses*. EClasses are shown as boxes in the meta mode diagram, so in our example, *Model*, *Type*, *SimpleType*, *Entity*, and *Property* are EClasses. An EClass can inherit from other EClasses. Multiple inheritance is allowed in Ecore, but of course cycles are forbidden.

EClasses can have *EAttributes* for their simple properties. These are shown inside the EClasses nodes. The example contains two EAttributes *name* and one EAttribute *isMulti*. The domain of values for an EAttribute is defined by its *EDataType*. Ecore ships with some predefined *EDataTypes*, which essentially refer to Java primitive types and other immutable classes like String. To make a distinction from the Java types, the EDataTypes are prefixed with an *E*. In our example, that's *EString* and *EBoolean*.

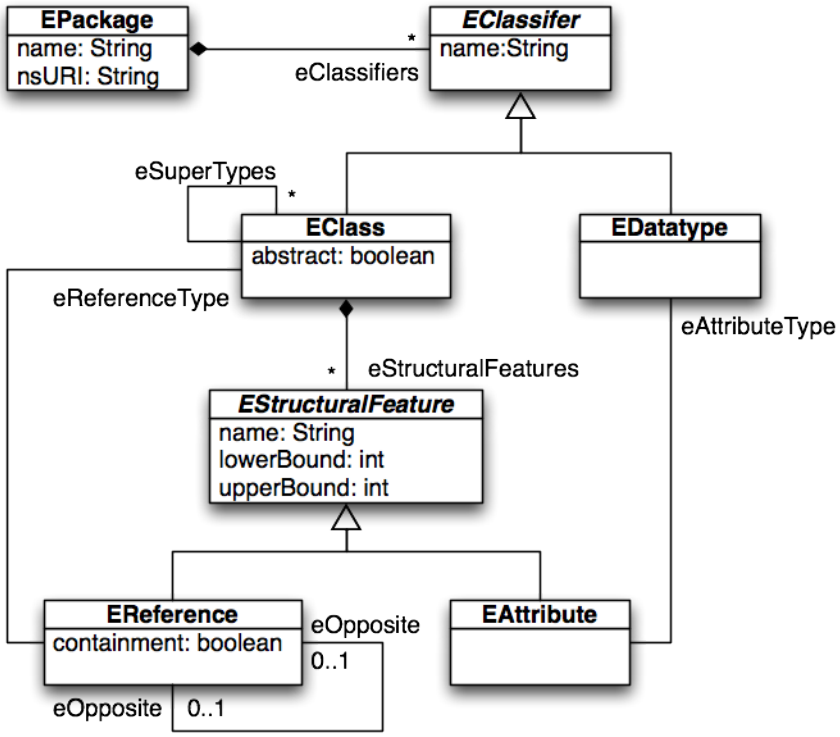
In contrast to EAttributes, *EReferences* point to other EClasses. The *containment* flag indicates whether an EReference is a *containment reference* or a *cross-reference*. In the diagram, references are edges and containment references are marked with a diamond. At the model level, each element can have at most one container, i.e. another element referring to it with a containment reference. This infers a tree structure to the models, as can be seen in the sample model diagram. On the other hand, *cross-references* refer to elements that can be contained anywhere else. In the example, *elements* and *properties* are containment references, while *type* and *extends* are cross-references. For reasons of readability, we skipped the cross-references in the sample model diagram. Note that in contrast to other parser generators, Xtext creates ASTs with linked cross-references.

Other than associations in UML, EReferences in Ecore are always owned by one EClass and only navigable in the direction form the owner to the type. Bi-directional associations must be modeled as two references, being *eOpposite* of each other and owned by either end of the associations.

The superclass of EAttibutes and EReferences is *EStructuralFeature* and allows to define a name and a cardinality by setting *lowerBound* and *upperBound*. Setting the latter to -1 means 'unbounded'.

The common supertype of EDataType and EClass is *EClassifier*. An *EPackage* acts as a namespace and container of EClassifiers.

We have summarized these most relevant concepts of Ecore in the following diagram:



9.2 EMF Code Generation

EMF also ships with a code generator that generates Java classes from your Ecore model. The code generators input is the so called *EMF generator model*. It decorates (references) the Ecore model and adds additional information for the Ecore -> Java transformation. Xtext will automatically generate a generator model with reasonable defaults for all generated metamodels, and run the EMF code generator on them.

The generated classes are based on the EMF runtime library, which offers a lot of infrastructure and tools to work with your models, such as persistence, reflection, referential integrity, lazy loading etc.

Among other things, the code generator will generate

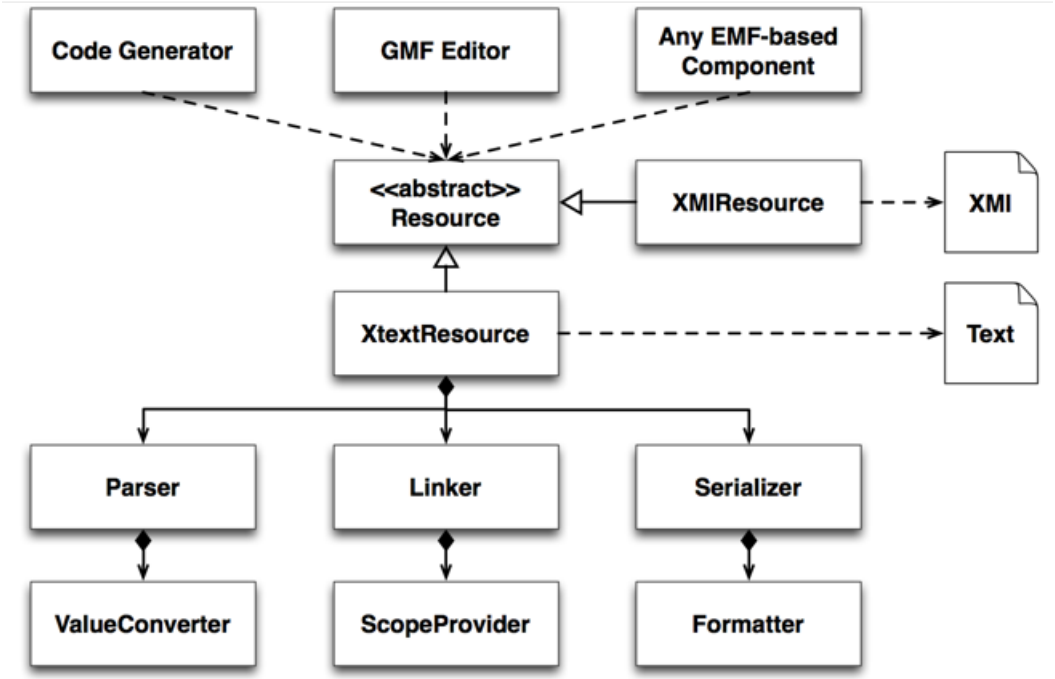
revise indentation levels

- A Java interface and a Java class for each EClassifier in your Ecore model. By default, all classes will implement the interface *org.eclipse.emf.ecore.EObject*, linking a lot of runtime functionality.
- A Java bean property for each EStructuralFeature (member variable, accessor methods)

- A package interface and class, holding singleton objects for all elements of your Ecore model, allowing reflection. EPackages are also registered to the *EPackage.Registry* to be usable at runtime.
- A factory interface and class for creating instances
- An abstract switch class implementing a visitor pattern to avoid if-instanceof cascades in your code.

9.3 XtextResource Implementation

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



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

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

revise indentation levels

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

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

revise indentation levels

- Prefer optional rule calls (cardinality *?* or ***) to mandatory ones (cardinality *+* or default), such that missing references will not obstruct serialization.
- You should not use an Xtext-Editor on the same model instance as a self-synchronizing other editor, e.g. a canonical GMF editor (see subsection 9.4.1 for details). The Xtext parser replaces re-parsed subtrees of the AST rather than modifying it, so elements will become stale. As the Xtext editor continuously re-parses the model on changes, this will happen rather often. It is safer to synchronize editors more loosely, e.g. on file changes.
- Implement an *IFragmentProvider* (§4.10) to make the XtextResource return stable fragments for its contained elements, e.g. based on composite names rather than order of appearance.
- Implement an *IQualifiedNameProvider* and an *IScopeProvider* (§4.6) to make the names of all linkable elements in cross-references unique.
- Provide an *IFormatter* (§4.9) to improve the readability of the generated textual models.
- Register an *IReferableElementsUnloader* to turn deleted/replaced model elements into EMF proxies. Design the rest of your application such that it does never keep references to *EObjects* or to cope with proxies. That will improve the stability of your application drastically.

- Xtext will register an EMF *ResourceFactory*, so resources with the file extension you entered when generating the Xtext plug-ins will be automatically loaded in an *XtextResource* when you use EMF's *ResourceSet* API to load it.

9.4 Integration with GMF Editors

The Graphical Modeling Framework (GMF) allows to create graphical diagram editors for Ecore models. To illustrate how to build a GMF on top of an *XtextResource* we have provided an example. You must have the Helios version 2.3 of GMF Notation, Runtime and Tooling and their dependencies installed in your workbench to run the example. With other versions of GMF it might work to regenerate the diagram code. Choose *New->Examples->Xtext->Xtext GMF Integration* to import it into your workbench. The example consists of a number of plug-ins

Plug-in	Framework	Purpose	Contents
o.e.x.example.gmf	Xtext	Xtext runtime plug-in	Grammar, derived metamodel and language infrastructure
o.e.x.e.g.ui	Xtext	Xtext UI plug-in	Xtext editor and services
o.e.x.e.g.edit	EMF	EMF.edit plug-in	UI services generated from the metamodel
o.e.x.e.g.models	GMF	GMF design models	Input for the GMF code generator
o.e.x.e.g.diagram	GMF	GMF diagram editor	Purely generated from the GMF design models
o.e.x.e.g.d.extensions	GMF and Xtext	GMF diagram editor extensions	Manual extensions to the generated GMF editor for integration with Xtext
o.e.x.gmf.glue	Xtext and GMF	Glue code	Generic code to integrate Xtext and GMF

We will elaborate the example in three stages.

9.4.1 Stage 1: Make GMF Read and Write the Semantic Model As Text

A diagram editor in GMF by default manages two resources: One for the semantic model, that is the model we're actually interested in for further processing. In our example it is a model representing entities and datatypes. The second resource holds the notation model. It represents the shapes you see in the diagram and their graphical properties. Notation elements reference their semantic counterparts. An entity's name would be in the semantic model, while the font to draw it in the diagram would be stored the notation model. Note that in the integration example we're only trying to represent the semantic resource as text.

To keep the semantic model and the diagram model in sync, GMF uses a so called *CanonicalEditPolicy*. This component registers as a listener to the semantic model and automatically updates diagram elements when their semantic counterparts change, are added or are removed. Some notational information can be derived from the semantic model by some default mapping, but usually there is a lot of graphical stuff that the user wants to change to make the diagram look better.

In an Xtext editor, changes in the text are transfered to the underlying *XtextResource* by a call to the method *org.eclipse.xtext.resource.XtextResource.update(int, int, String)*, which will trigger a partial parsing of the dirty text region and a replacement of the corresponding subtree in the AST model (semantic model).

Having an Xtext editor and a canonical GMF editor on the same resource can therefore lead to loss of notational information, as a change in the Xtext editor will remove a subtree in the AST, causing the *CanonicalEditPolicy* to remove all notational elements, even though it was customized by the user. The Xtext rebuilds the AST and the notation model is restored using the default mapping. It is therefore not recommended to let an Xtext editor and a canonical GMF editor work on the same resource.

In this example, we let each editor use its own memory instance of the model and synchronize on file changes only. Both frameworks already synchronize with external changes to the edited files out-of-the-box. In the glue code, a *org.eclipse.xtext.gmf.glue.concurrency.ConcurrentModif*

codeRef

warns the user if she tries to edit the same file with two different model editors concurrently.

In the example, we started with writing an Xtext grammar for an entity language. As explained above, we preferred optional assignments and rather covered mandatory attributes in a validator. Into the bargain, we added some services to improve the EMF integration, namely a formatter, a fragment provider and an unloader. Then we let Xtext generate the language infrastructure. From the derived Ecore model and its generator model, we generated the edit plug-in (needed by GMF) and added some fancier icons.

From the GMF side, we followed the default procedure and created a gmfgraph model, a gmftool model and a gmfmap model referring to the Ecore model derived form the Xtext grammar. We changed some settings in the gmfgen model derived by GMF from the gmfmap model, namely to enable printing and to enable validation and validation decorators. Then we generated the diagram editor.

Voilà, we now have a diagram editor that reads/writes its semantic model as text. Also note that the validator from Xtext is already integrated in the diagram editor via the menu bar.

Stage 2: Calling the Xtext Parser to Parse GMF Labels

GMF's generated parser for the labels is a bit poor: It will work on attributes only, and will fail for cross-references, e.g. an attibute's type. So why not use the Xtext parser to process the user's input?

An *XtextResource* keeps track of it's concrete syntax representation by means of a so called node model (see subsection 2.2.4 for a more detailed description). The node

model represents the parse tree and provides information on the offset, length and text that has been parsed to create a semantic model element. The nodes are attached to their semantic elements by means of a node adapter.

We can use the node adapter to access the text block that represents an attribute, and call the Xtext parser to parse the user input. The example code is contained in *org.eclipse.xtext.gmf.glue.edit.part.AntlrParserWrapper*

codeRef

. *SimplePropertyWrapperEditPartOverride* shows how this is integrated into the generated GMF editor. Use the *EntitiesEditPartFactoryOverride* to instantiate it and the *EntitiesEditPartProviderOverride* to create the overridden factory, and register the latter to the extension point. Note that this is a non-invasive way to extend generated GMF editors.

When you test the editor, you will note that the node model will be corrupt after editing a few labels. This is because the node model is only updated by the Xtext parser and not by the serializer. So we need a way to automatically call the (partial) parser every time the semantic model is changed. You will find the required classes in the package *org.eclipse.xtext.gmf.glue.editingdomain*. To activate node model reconciling, you have to add a line

```
XtextNodeModelReconciler.adapt(editingDomain);
```

in the method *createEditingDomain()* of the generated *EntitiesDocumentProvider*. To avoid changing the generated code, you can modify the code generation template for that class by setting

```
Dynamic Templates -> true
    Template Directory -> "org.eclipse.xtext.example.gmf.models/templates"
```

in the *GenEditorGenerator* and

```
Required Plugins -> "org.eclipse.xtext.gmf.glue"
```

in the *GenPlugin* element of the gmfgen before generating the diagram editor anew.

Stage 3: A Popup Xtext Editor (experimental)

SimplePropertyPopupXtextEditorEditPartOverride demonstrates how to spawn an Xtext editor to edit a model element. The editor pops up in its control and shows only the section of the selected element. It is a fully fledged Xtext editor, with support of validation, code assist and syntax highlighting. The edited text is only transfered back to the model if it does not have any errors.

Note that there still are synchronization issues, that's why we keep this one marked as experimental.

10 Migrating from Xtext 0.7.x to 1.0

Most of the tasks when migrating to Xtext 1.0 can be automated. Some changes will be necessary in the manually written code where you have to carefully verify that your implementation is still working with Xtext 1.0. A reliable test-suite helps a lot.

The grammar language is fully backward compatible. You should not have to apply any changes in the primary artifact. However, we introduced some additional validation rules that mark inconsistencies in your grammar. If you get any warnings in the grammar editor, it should be straight forward to fix them.

Tip: You'll learn something about the new features if you compare a freshly created Xtext project based on 0.7.x with a new Xtext project based on 1.0. Especially the new fragments in the workflow are a good indicator for useful new features.

10.1 Take the Shortcut

If you haven't made too many customizations to the generated defaults and if you're not referencing many classes of your Xtext language from the outside, you might consider starting with a new Xtext project, copying your grammar and then manually restoring your changes step by step. If that does not work for you, go on reading!

10.2 Migrating Step By Step

Before you start the migration to Xtext 1.0, you should make sure that no old plug-ins are in your target platform. Some plug-ins from Xtext 0.7.x have been merged and do no longer exist.

Tip: The following steps try to use the Eclipse compiler to spot any source-incompatible changes while fixing them with only a few well described user actions. Doing these steps in another order causes most likely a higher effort.

10.2.1 Update the Plug-in Dependencies and Import Statements

You should update the constraints from version *0.7.x* to *[1.0.0,2.0.0)* in your manifest files if you specified any concrete versions. Make sure that your *dsl.ui*-projects do not refer to the plug-in *org.eclipse.xtext.ui.common* or *org.eclipse.xtext.ui.core* but to *org.eclipse.xtext.ui* instead. The arguably easiest way is a global text-based search and replace across the manifest files. The bundle *org.eclipse.xtext.log4j* is obsolete as well. The generator will create *import-package* entries in the manifests later on.

The next step is to fix the import statements in your classes to match the refactored naming scheme in Xtext. Perform a global search for "_import org.eclipse.xtext.ui.common._"

and "_import org.eclipse.xtext.ui.core._" and replace the matches with "_import org.eclipse.xtext.ui._". This fixes most of the problems in the manually written code.

10.2.2 Rename the Packages in the dsl.ui-Plug-in

We changed the naming pattern for artifacts in the *dsl.ui*-plug-in to match the OSGI conventions. The easiest way to update your existing projects is to apply a "Rename Package" refactoring on the packages in the *src-* and *src-gen* folder *before* you re-run the workflow that regenerates your language. Make sure you ticked "Rename sub-packages" in the dialog. It is error-prone to enable the search in non-Java files as this will perform incompatible changes in the manifest files. Furthermore, it is important to perform the rename operation in the *src-gen* folder, too. This ensures that the references in your manually written code are properly updated.

10.2.3 Update the Workflow

The *JavaScopingFragment* does no longer exist. It has been superseded by the *ImportURIScopingFragment* in combination with the *SimpleNamesFragment*. Please replace

```
<fragment
    class="org.eclipse.xtext.generator.scoping.JavaScopingFragment"/>
```

with

```
<fragment
    class="org.eclipse.xtext.generator.scoping.ImportURIScopingFragment"/>
<fragment
    class="org.eclipse.xtext.generator.exporting.SimpleNamesFragment"/>
```

The *PackratParserFragment* has been abandoned as well. It is save to remove the reference to that one if it is activated in your workflow. After you've changed your workflow, it should be possible to regenerate your language without any errors in the console. It is ok to have compilation errors prior to executing the workflow.

10.2.4 MANIFEST.MF and plugin.xml

The previous rename package refactoring updated most of the entries in the *MANIFEST.MF* and and some entries in the *plugin.xml*. Others have to be fixed manually. The Eclipse compiler will point to many of the remaining problems in the manifest files but it is unlikely that it will spot the erroneous references in the *plugin.xml*.

revise indentation levels

- In the generated UI plug-in's *MANIFEST.MF*, remove the package exports of no longer existing packages and make sure the bundle activator points to the newly generated one (with *.ui.* in its package name).
- It was already mentioned that the plug-ins *org.eclipse.xtext.ui.core* and *org.eclipse.xtext.ui.common* have been merged into a new single plug-in *org.eclipse.xtext.ui*. The same happened to the respective Java packages. Change eventually remaining bundle-dependencies in all manifests.

- The plug-in *org.eclipse.xtext.log4j* no longer exists. We use a package import of *org.apache.log4j* instead. Also remove the buddy registration.
- Due to renamed packages, you have to fix all references to classes therein in the *plugin.xml*. A diff with the *plugin.xml_gen* will be a great help. If you haven't added a lot manually, consider merging these into the generated version instead of going the other way around. Note that warnings in the *plugin.xml* can be considered to be real errors most of the time. Make sure
- the *[MyDsl]ExecutableExtensionFactory* has the *.ui.* package prefix
- classes from *org.eclipse.xtext.ui.common* and *org.eclipse.xtext.ui.core* are now usually somewhere in *org.eclipse.xtext.ui*. They are also referenced by the *[MyDsl]ExecutableExtensionFactory* and thus not covered by the editor's validation.
- A number of new features are being registered in the *plugin.xml*, e.g. *Find references*, *Quick Outline*, and *Quick Fixes*. You can enable them by manually copying the respective entries from *plugin.xml*_gen_ to *plugin.xml*.
- To run MWE2 workflows later on, you must change the plug-in dependencies from *org.eclipse.emf.mwe.core* to *org.eclipse.emf.mwe2.launch* in your manifest. Optional resolution is fine.

10.2.5 Noteworthy API Changes

The *src* folders are generated once, so existing code will not be overwritten but has to be updated manually. At least one new class has appeared in your *src*-folder of the *ui* plug-in. there will now be a *[MyDsl]StandaloneSetup* inheriting form the generated *[MyDsl]StandaloneSetupGenerated* to allow customization.

You will face a couple of compilation problems due to changes in the API. Here's a list of the most prominent changes. It is usually only necessary to change your code, if you face any compilation problems.

revise indentation levels

- The method *IScopeProvider.getScope(EObject,EClass)* has been removed. Use *IScopeProvider.getScope(EObject,EReference)* instead.
- Renamed *DefaultScopeProvider* to *SimpleNameScopeProvider*. There have been further significant changes in the scoping API that allow for optimized implementations. Consult the section on scoping (§4.6) for details.
- The return type of *AbstractInjectableValidator.getEPackages()* was changed from *List<? extends EPackage>* to *List<EPackage>*.
- The parser interfaces now use *java.io.Reader* instead of *java.io.InputStream* to explicitly address encoding. Have a look at the section on encoding (§4.11) for details.

- The handling of *ILabelProvider* in various contexts has been refactored. The former base class *DefaultLabelProvider* no longer exists. Use the *DefaultEObjectLabelProvider* instead. See the section on label providers (§6.1) for details.
- We have introduced a couple of new packages to better separate concerns. Most classes should be easy to relocate.
- The runtime and UI modules have separate base classes *DefaultRuntimeModule* and *DefaultUiModule* now. We use Guice's module overrides to combine them with the newly introduced *SharedModule*. You have to add a constructor the your *[MyDsl]UiModule* that takes an *AbstractUiPlugin* as argument and pass that one to the super constructor. *Tip: There is an Eclipse quick fix available for that one.*
- The interfaces *ILexicalHighlightigConfiguration* and *ISemanticHighlightingConfiguration* have been merged into *IHighlightingConfiguration*.
- The *DefaultTemplateProposalProvider* takes an additional, injectable constructor parameter of type *ContextTypeIdHelper*.
- The *HyperlinkHelper* uses field injection instead of constructor injection. The method *createHyperlinksByOffset* should be overridden instead of the former *findCrossLinkedEObject*.
- The API to skip a node in the outline has changed. Instead of returning the *HIDDEN_NODE* you'll have to implement *boolean consumeNode(MyType)* and return *false*.
- The *Readonly*Storage* implementations have been removed. There is a new API to open editors for objects with a given URI. Please use the *IURIEditorOpener* to create an editor or the *IStorage2UriMapper* to obtain an *IStorage* for a given URI.
- The interfaces *IStateAccess* and *IEObjectHandle* have been moved along with the *IUnitOfWork* to the package *org.eclipse.xtext.util.concurrent*.
- The *ValidationJobFactory* is gone. Please implement a custom *IResourceValidator* instead.
- The grammar elements *Alternatives* and *Group* have a new common super type *CompoundElement*. The methods *getGroups* and *getTokens* have been refactored to *getElements*.
- Completion proposals take a *StyledString* instead of a plain string as display string.
- The *AbstractLabelProvider* does no longer expose its *IImageHelper*. Use *convertToImage* instead or inject your own *IImageHelper*.
- The implementation-classes from *org.eclipse.xtext.index* were superseded by the builder infrastructure. Use the *QualifiedNamesFragment* and the *ImportNamespacesScopingFragment* instead of the *ImportedNamespacesScopingFragment*. Please refer to the section about the builder infrastructure for details.
- All the Xtend-based fragments were removed.

- _ILinkingService.getLinkText_ was removed. Have a look at the *LinkingHelper* and the *CrossReferenceSerializer* if you relied on this method.
- The *SerializerUtil* was renamed to *Serializer*. There were other heavy refactorings that involved the serializer and its components (like e.g. the *ITransientValueService*) but it should be pretty straight forward to migrate existing client code.
- The method-signatures of the *IFragmentProvider* have changed. The documentation will give a clue on how to update existing implementations.
- Some static methods were removed from utility classes such as *EcoreUtil2* and *ParsetreeUtil* in favor of more sophisticated implementations.

10.3 Now Go For The New Features

After migrating, some of the new features in Xtext 1.0 will be automatically available. Others require further configuration. We recommend reading the sections about

revise indentation levels

- qualified names and namespace imports (§4.6.1)
- the builder infrastructure (§4.6.1)
- quick fixes (§6.3)
- unordered groups (§2.2.4)
- quick outline (§6.5.5)
- MWE2 (§5)
- refering to Java elements (§7)

For an overview over the new features consult our New and Noteworthy online.

11 Deploying DSLs

Deploying a DSL is a straightforward process - all you need to know is how to build Eclipse Feature Projects. For convenience, here is a short overview to get you started:

11.1 Prepare Your DSL Projects

If you use custom icons in your DSL, please make sure to include the */icons* folder (or whatever you called the folder with your custom icons) in the binary build of the UI bundle of your DSL. Otherwise, your icons will not show up in the deployed DSL editor.

11.2 Creating a Feature For Your DSL

11.3 Creating an Update Site For Your DSL

12 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 self defined language grammar.

It is recommended to use the ANTLR -based parser. ANTLR is a very sophisticated parser generator framework which implements 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.

Xtext uses ANTLR 3 which 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 ANTLR parser generator is unfortunately still implemented in an older version of itself (v 2.x), where it was not possible for the Eclipse Foundation to ensure where exactly every line of code originated. Therefore ANTLR 2 didn't get the required IP approval. Eclipse has a strict IP policy, which makes sure that everything provided by Eclipse can be consumed under the terms of the Eclipse Public License. The details are described in this document That is why we are not allowed to ship Xtext with the ANTLR generator but only with the IP approvied runtime components. We have to provide it separately and for your convenience you'll be asked to download the ANTLR generator when you run your language generator for the first time. You can even download it directly or install an additional plug-in into Eclipse:

revise indentation levels

- http://download.itemis.com/antlr-generator-3.0.1.jar
- or use the update site at http://download.itemis.com/updates

The workflow will not bother you with this issue on subsequent executions as the archive will be stored in your project's root directory and can thereby be reused during the next run.

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 it is not needed at runtime

List of External Links

http://help.eclipse.org/help32/topic/org.eclipse.emf.doc/references/javadoc/org/eclipse/emf/common/util/URI.html
http://www.ietf.org/rfc/rfc2396.txt
http://code.google.com/p/google-guice/
http://www.eclipse.org/Xtext/documentation/helios/new_and_noteworthy.php
http://help.eclipse.org/ganymede/topic/org.eclipse.cdt.doc.user/tasks/cdt_t_imp_code_temp.htm
http://help.eclipse.org/stable/index.jsp?topic=/org.eclipse.emf.doc/references/javadoc/org/eclipse/emf/ecore/change/impl/package-summary.html
http://martinfowler.com/bliki/SyntacticNoise.html
http://martinfowler.com/books.html#dsl
http://blog.efftinge.de/2009/07/xtext-scopes-and-emf-index-in-action.html
http://download.itemis.com/updates
http://www.antlr.org
http://www.eclipse.org/emf.
http://java.sun.com/j2se/1.5.0/docs/guide/intl/encoding.doc.html.
http://blog.efftinge.de/2009/01/xtext-scopes-and-emf-index.html
http://download.itemis.com/antlr-generator-3.0.1.jar
http://code.google.com/p/google-guice/.
http://www.eclipse.org/emf
http://xtext.itemis.com
http://git.eclipse.org/c/tmf/org.eclipse.xtext.git/tree/plugins/org.eclipse.xtext/src/org/eclipse/xtext/Xtext.xtext
http://www.eclipse.org/org/documents/Eclipse_IP_Policy.pdf.
http://www.eclipse.org/modeling/gmp/?project=gmf

Todo list