Xtext 2.3 Documentation

June 27, 2012

Contents

Part I.

Getting Started

1. 5 Minutes Tutorial

In this chapter you will learn how to create a new Xtext project, generate a fully working language infrastructure and how to start a new Eclipse instance in order to test the editor. But before we get started, make sure you have Eclipse Xtext properly installed.

1.1. Creating A New Xtext Project

The first step is to create a new Xtext project by choosing *File -> New -> Project...*. The dialog offers a couple of different project types. Select *New Xtext Project* from the category *Xtext* and finish the wizard with the default settings. You will find 3 new projects in your workspace which are mostly empty. The Xtext grammar editor will be opened and show the definition of a very simple *Hello World* language.

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

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

Model:
  greetings+=Greeting*;

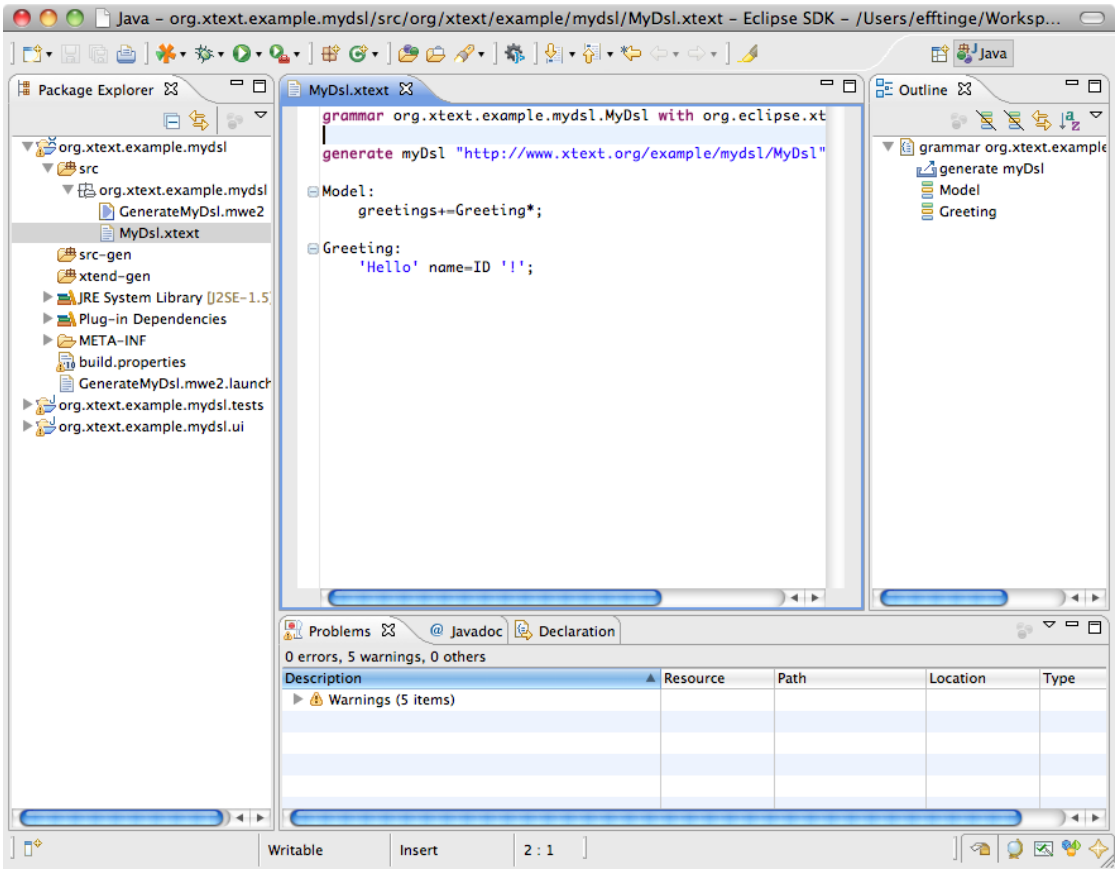
Greeting:
  'Hello' name=ID '!';
```

The only thing this language does, is to allow to write down a list of greetings. The following would be proper input:

```
Hello Xtext!
Hello World!
```

1.2. Generating The Language Infrastructure

In order to test drive this language, you will have to generate the respective language infrastructure. Therefore, choose *Run As -> Generate Xtext Artifacts* from the context

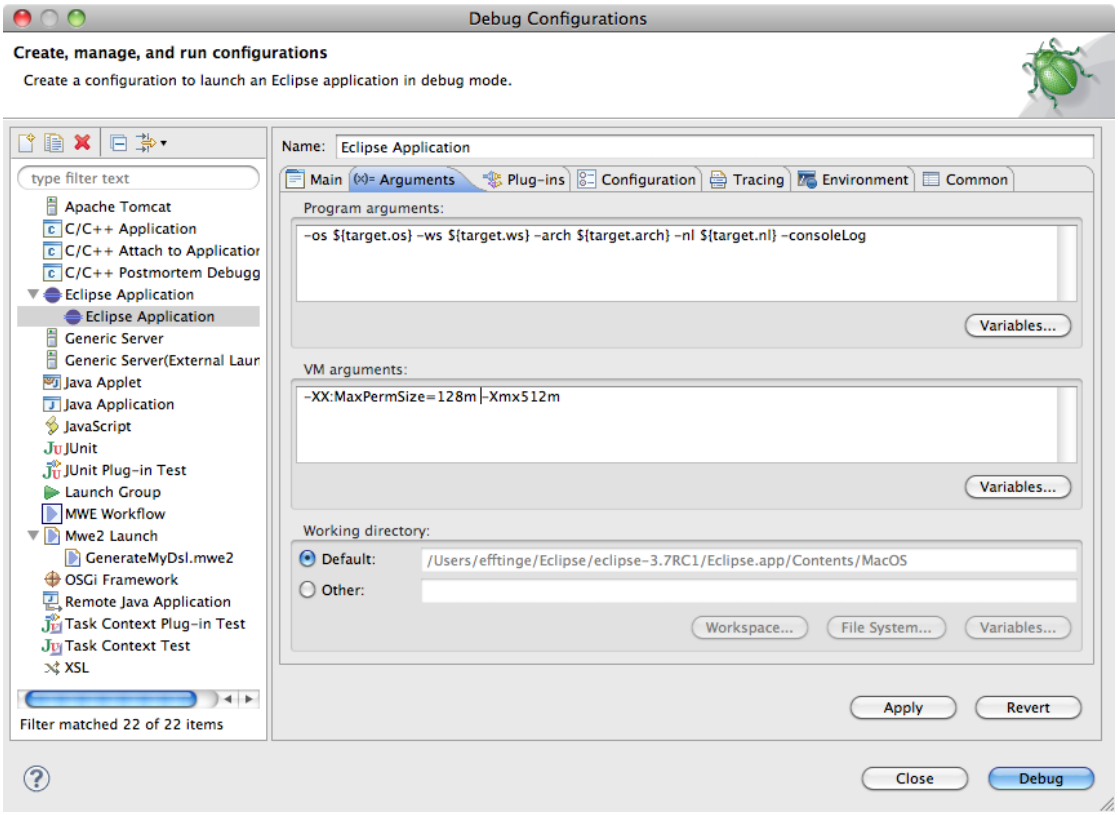


menu of the grammar editor. A new Java process will be spawned and afterwards you will find a couple of new files in the projects that were created in the first step. What you see now is a runnable language infrastructure with a powerful Eclipse editor for a brain-dead language :-).

1.3. Try The Editor

Let's give the editor a try. If you select *Run As -> Eclipse Application* from the project's context menu, you can create a new Eclipse Application. A new Eclipse instance will be launched and allows to test drive the editor.

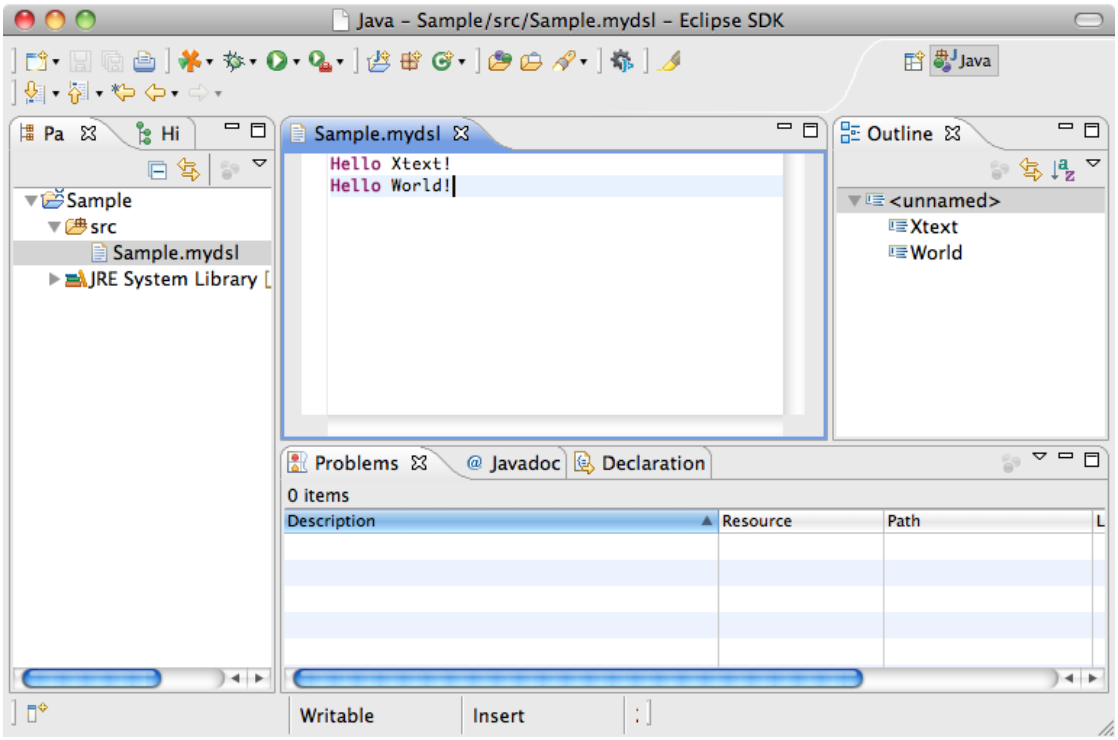
Before you can create a file for the sample language, you will have to create a sample project. Select *File -> New -> Project...* and choose a project type of your choice, e.g. *Java Project*, name it *Sample* and create a new file in the *src* folder of the project: From the context menu of the folder choose *New -> File*, call it *Sample.mydsl* and hit *Finish*. The newly created editor will open for your language and ask you in a dialog, whether you want to add the Xtext nature to your project, which should be confirmed. You can now give the editor a try, e.g. use content assist (*Ctrl+Space*) to insert the keyword Hello and see how the input is validated immediately.



1.4. Conclusion

In your first five minutes with Xtext, you have learned how to create a new set of projects. You have run Xtext's code generation in order to get a fully working language infrastructure, and finally learned how to test the generated editor.

Next up you should go through the more comprehensive Domain Model Example (§2). It explains the different concepts of the Xtext grammar language and illustrates how to customize various aspects of the language.


Java – Sample/src/Sample.mydsl – Eclipse SDK
Java
Sample.mydsl
Hello Xtext!
Hello World!
Outline
<unnamed>
Xtext
World
Sample
src
Sample.mydsl
JRE System Library
Problems
Javadoc
Declaration
0 items
Description
Resource
Path
Writable
Insert

2. 15 Minutes Tutorial

In this tutorial we will implement a small domain-specific language to model entities and properties similar to what you may know from Rails, Grails or Spring Roo. The syntax is very suggestive :

```
datatype String

entity Blog {
  title: String
  many posts: Post
}

entity HasAuthor {
  author: String
}

entity Post extends HasAuthor {
  title: String
  content: String
  many comments: Comment
}

entity Comment extends HasAuthor {
  content: String
}
```

After you have installed Xtext on your machine, start Eclipse and set up a fresh workspace.

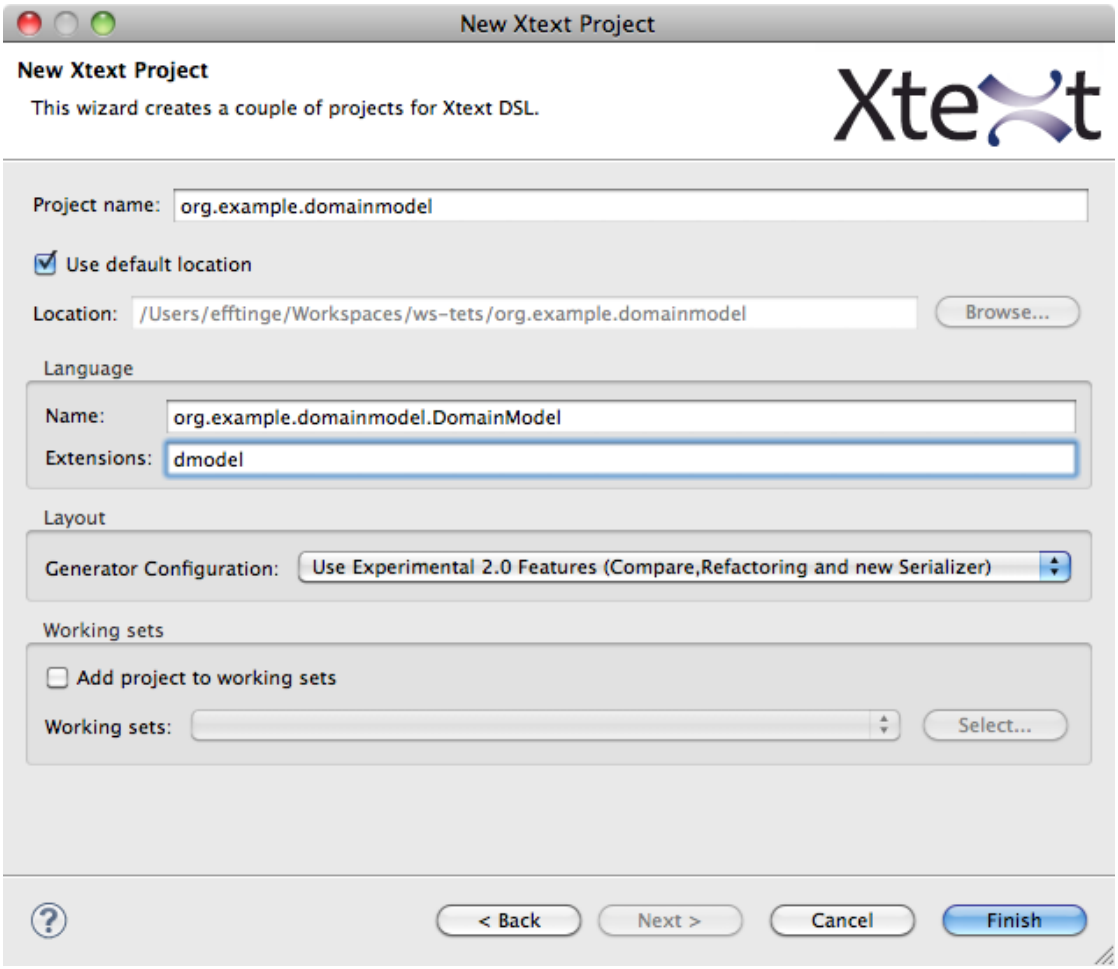
2.1. Create A New Xtext Project

In order to get started we first need to create some Eclipse projects. Use the Eclipse wizard to do so:

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

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

Main project name:	org.example.domainmodel
Language name:	org.example.domainmodel.Domainmodel
DSL-File extension:	dmodel



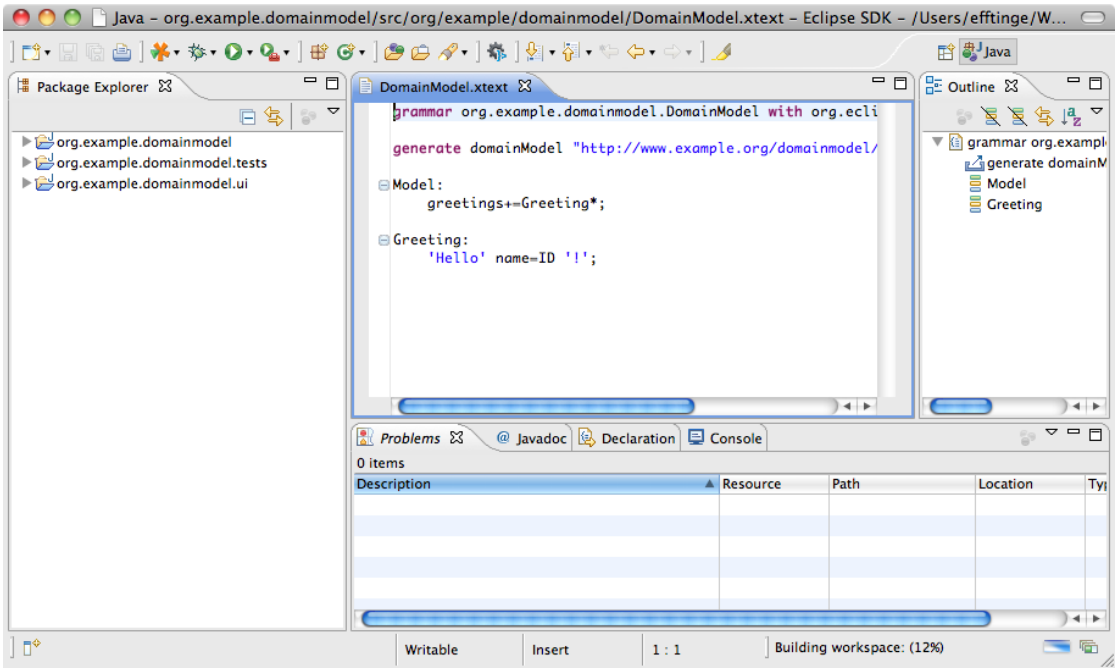
Click on *Finish* to create the projects.

After you have successfully finished the wizard, you will find three new projects in your workspace.

org.example.domainmodel	Contains the grammar definition and all runtime components (parser, lexer, linker, validation, etc.)
org.example.domainmodel.tests	Unit tests go here.
org.example.domainmodel.ui	The Eclipse editor and all the other workbench related functionality.

2.2. Write Your Own Grammar

The wizard will automatically open the grammar file *Domainmodel.xtext* in the editor. As you can see that it already contains a simple *Hello World* grammar:




```
grammar org.example.domainmodel.Domainmodel with
                                org.eclipse.xtext.common.Terminals

generate domainmodel "http://www.example.org/domainmodel/Domainmodel"

Model:
  greetings+=Greeting*;

Greeting:
  'Hello' name=ID '!';
```

Let's now just replace that grammar definition with the one for our domain model language:

```
grammar org.example.domainmodel.Domainmodel with
                                org.eclipse.xtext.common.Terminals

generate domainmodel "http://www.example.org/domainmodel/Domainmodel"

Domainmodel :
  elements += Type*
;
```

```
Type:
  DataType | Entity
;

DataType:
  'datatype' name = ID
;

Entity:
  'entity' name = ID ('extends' superType = [Entity])? '{'
    features += Feature*
  '}'
;

Feature:
  many?='many'? name = ID ':' type = [Type]
;
```

Let's have a more detailed look at what the different grammar rules mean:

1. The first rule in a grammar is always used as the entry or start rule.

   ```
   Domainmodel :
     elements += Type*
   ;
   ```

   It says that a *Domainmodel* contains an arbitrary number (*) of *Type*s which will be added (+=) to a feature called elements.

2. The rule *Type* delegates to either the rule *DataType* or (|) the rule *Entity*.

   ```
   Type:
     DataType | Entity
   ;
   ```

3. The rule *DataType* starts with a keyword 'datatype', followed by an identifier which is parsed by a rule called *ID*. The rule *ID* is defined in the super grammar *org.eclipse.xtext.common.Terminals* and parses a single word, a.k.a identifier. You can navigate to the declaration by using *F3* on the rule call. The value returned by the call to *ID* is assigned (=) to the feature *name*.

```
DataType:
  'datatype' name = ID
;
```

4. The rule *Entity* again starts with the definition of a keyword followed by a name.

```
Entity :
  'entity' name = ID ('extends' superType = [Entity])? '{'
    features += Feature*
  '}'
;
```

   Next up there is the extends clause which is parenthesized and optional (?). Since the feature named *superType* is a cross reference (note the square brackets), the parser rule *Entity* is not called here, but only a single identifier (the *ID*-rule) is parsed. The actual *Entity* will be resolved during the linking phase. Finally between curly braces there can be any number of *Features*, which invokes the next rule.

5. Last but not least, the rule *Feature* is defined as follows:

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

   The keyword many is used to model a multi valued feature in the domain model DSL. The assignment operator (?=) implies that the feature *many* is of type *boolean*. You are already familiar with the other syntax elements in this parser rule.

This domain model grammar already uses the most important concepts of Xtext's grammar language. you have learned that keywords are written as string literals and a simple assignment uses a plain equal sign (=) where the multi value assignment used a plus-equals (+=). We have also seen the boolean assignment operator (?=). Furthermore

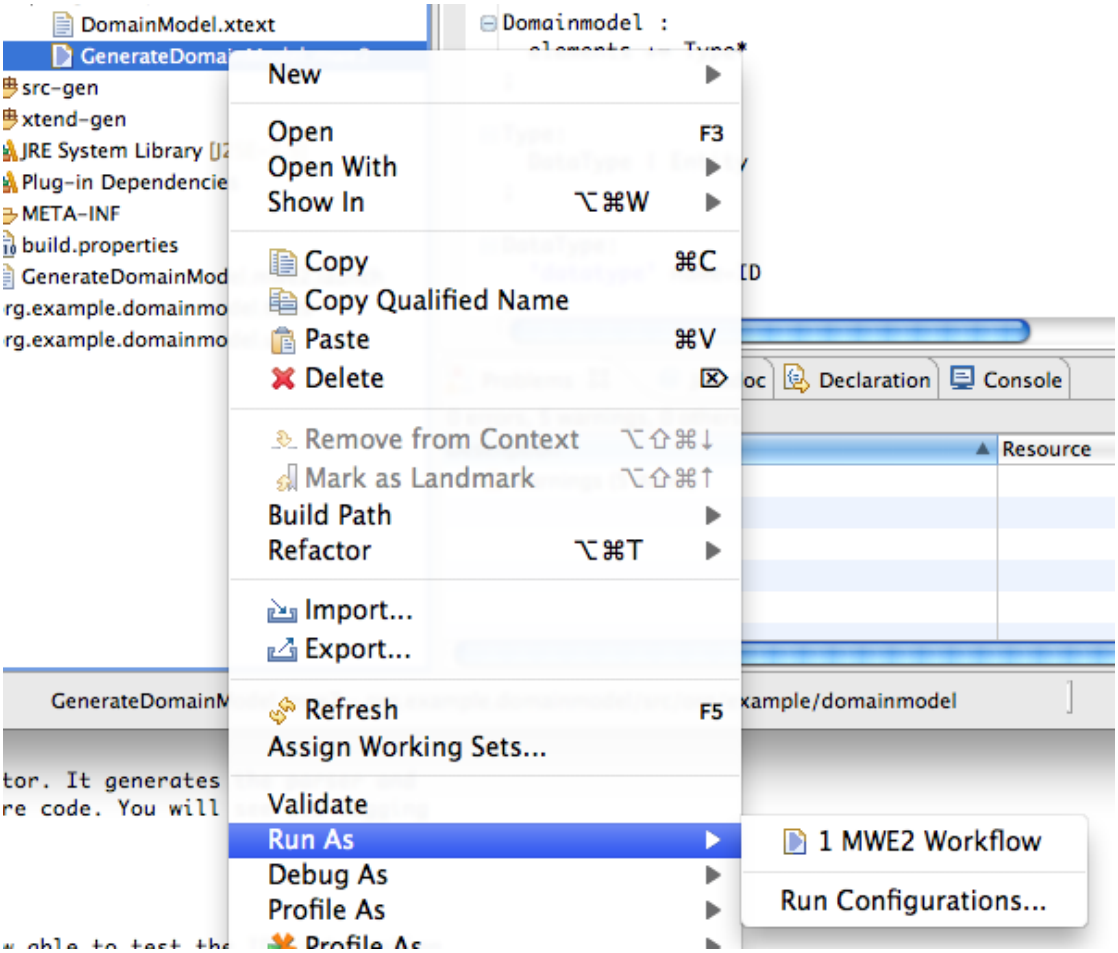
we saw how a cross reference can be declared and learned about different cardinalities (? = optional, * = any number, + = at least once). Please consult the Grammar Language Reference (§6) for more details. Let's now have a look what you can do with such a language description.

2.3. Generate Language Artifacts

Now that we have the grammar in place and defined we need to execute the code generator that will derive the various language components. To do so, locate the file *GenerateDomainmodel.mwe2* file next to the grammar file in the package explorer view. From its context menu, choose

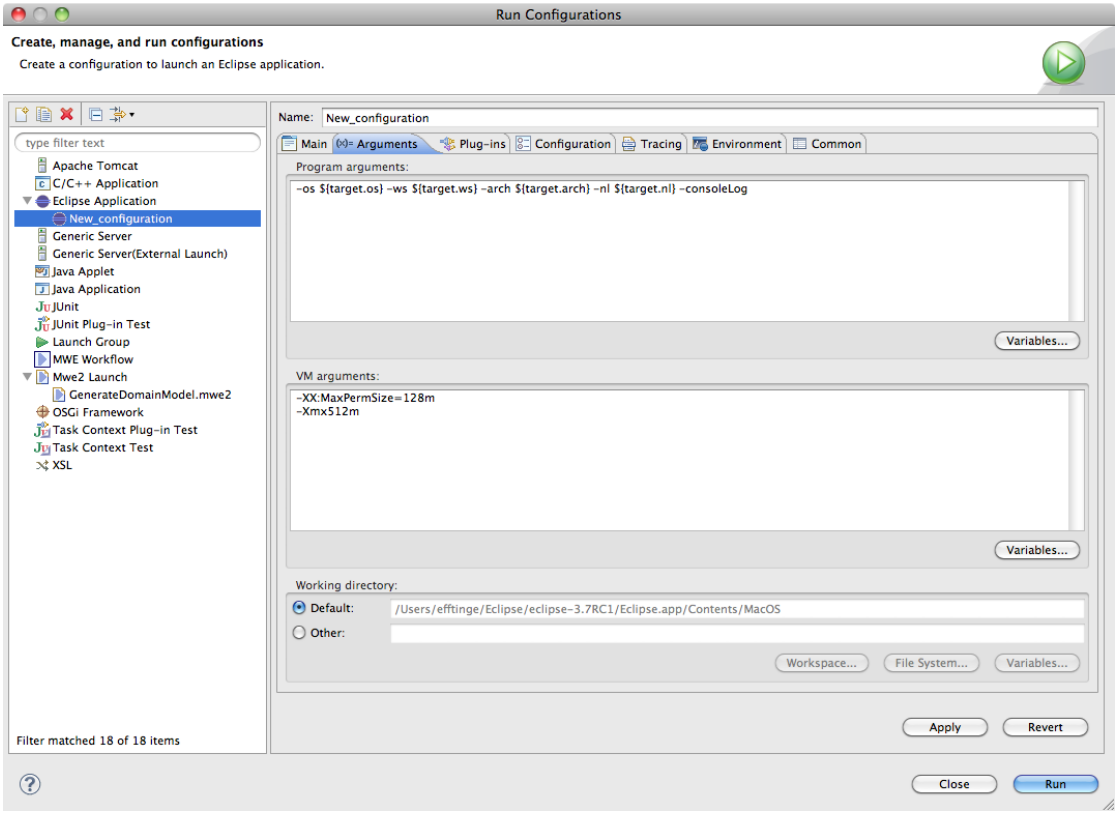
Run As -> MWE2 Workflow.

This will trigger the Xtext language generator. It generates the parser and serializer and some additional infrastructure code. You will see its logging messages in the Console View.



2.4. Run the Generated IDE Plug-in

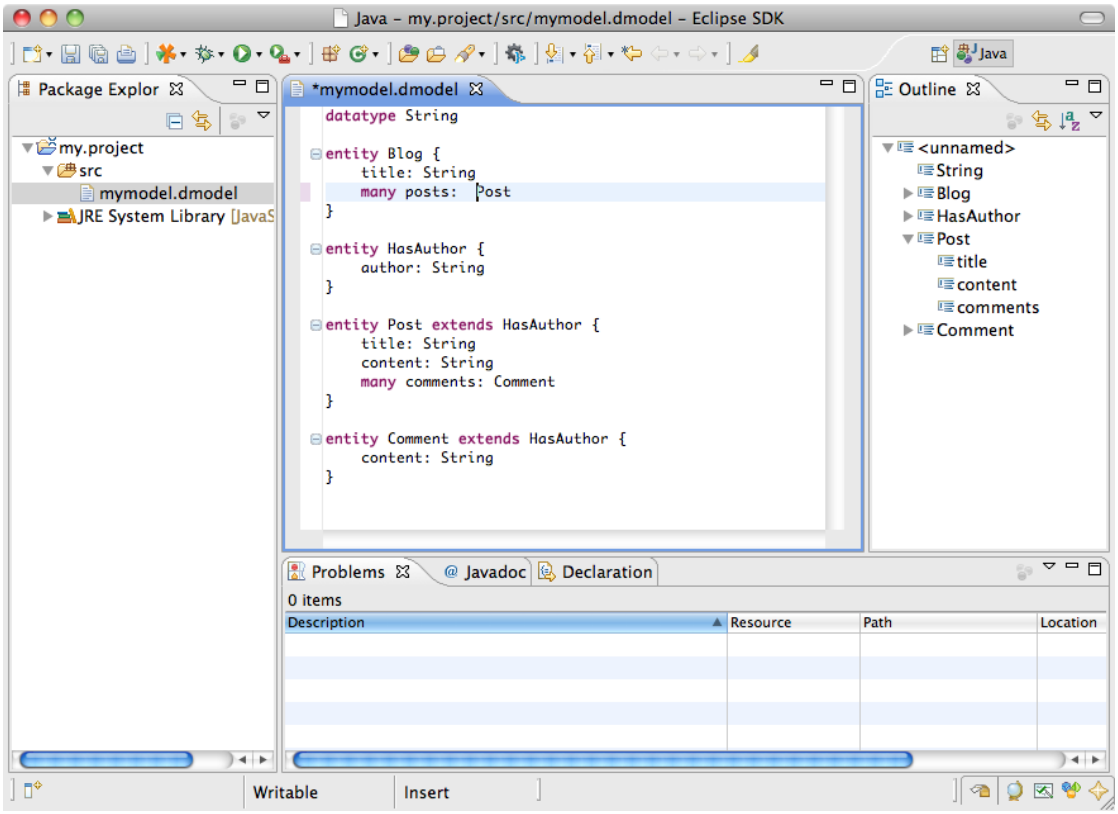
We are now able to test the IDE integration. If you select *Run -> Run Configurations...* from the Eclipse menu, you can choose *Eclipse Application -> Launch Runtime Eclipse*. This preconfigured launch shortcut already has appropriate memory settings and parameters set. Now you can hit *Run* to start a new Eclipse.



This will spawn a new Eclipse workbench with your newly developed plug-ins installed. In the new workbench, create a new project of your choice, e.g. *File -> New -> Project... -> Java Project* and therein a new file with the file extension you chose in the beginning (**.dmodel*). This will open the generated entity editor. Try it and discover the default functionality for code completion, syntax highlighting, syntactic validation, linking errors, the outline view, find references etc.

2.5. Second Iteration: Adding Packages and Imports

After you have created the your first DSL and had a look at the editor, the language should be refined and incrementally enhanced. The Domain Model language should support the notion of *Package*s in order to avoid name clashes and to better fit with the target environment (Java). A *Package* may contain *Types* and other packages. In order to allow fort names in references, we will also add a way to declare imports.



In the end we want to be able to split the previously used model into to distinct files :

```
// datatypes.dmodel

datatype String
```

```
// commons.dmodel

package my.company.common {

  entity HasAuthor {
    author: String
  }
}
```

```
// blogs.dmodel
```

```
package my.company.blog {

  import my.company.common.*

  entity Blog {
    title: String
    many posts: Post
  }

  entity Post extends my.company.common.HasAuthor {
    title: String
    content: String
    many comments: Comment
  }

  entity Comment extends HasAuthor {
    content: String
  }
}
```

Let's start enhancing the grammar.

1. Since a *Domainmodel* no longer contains types but packages, too, the entry rule has to be modified. Furthermore, a common super type for *Packages* and *Types* should be introduced: the *AbstractElement*.

   ```
   Domainmodel:
     (elements += AbstractElement)*
   ;

   AbstractElement:
     PackageDeclaration | Type
   ;
   ```

2. A PackageDeclaration in turn looks pretty much as expected. It contains a number of *Imports* and *AbstractElements*. Since *Imports* should be allowed for the root-Domainmodel, too, we add them as an alternative to the rule AbstractElement.

```
PackageDeclaration:
  'package' name = QualifiedName '{'
    (elements += AbstractElement)*
  '}'
;

AbstractElement:
  PackageDeclaration | Type | Import
;

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

The QualifiedName is a little special. It does not contain any assignments. Therefore, it serves as a data type rule, which returns a String. So the feature *name* of a *Package* is still of type String.

3. Imports can be defined in a very convenient way with Xtext. If you use the name *importedNamespace* in a parser rule, the framework will treat the value as an import. It even supports wildcard and handles them as expected:

```
Import:
  'import' importedNamespace = QualifiedNameWithWildcard
;
   QualifiedNameWithWildcard:
  QualifiedName '.*'?
;
```

Similar to the rule QualifiedName, QualifiedNameWithWildcard returns a plain string.

4. The last step is to allow fully qualified names in cross references, too. Otherwise one could not refer to an entity without adding an import statement.

```
Entity:
  'entity' name = ID
           ('extends' superType = [Entity | QualifiedName])?
  '{'
    (features += Feature)*
  '}'
;

Feature:
  (many ?= 'many')? name = ID ':' type = [Type | QualifiedName]
;
```

Please note that the bar (|) is not an alternative in the context of a cross reference, but used to specify the syntax of the parsed string.

That's all for the grammar. It should now read as

```
grammar org.example.domainmodel.Domainmodel with
                                org.eclipse.xtext.common.Terminals

generate domainmodel "http://www.example.org/domainmodel/Domainmodel"

Domainmodel:
  (elements += AbstractElement)*
;

PackageDeclaration:
  'package' name = QualifiedName '{'
    (elements += AbstractElement)*
  '}'
;

AbstractElement:
  PackageDeclaration | Type | Import
;

QualifiedName:
  ID ('.' ID)*
;

Import:
  'import' importedNamespace = QualifiedNameWithWildcard
;
```

```
QualifiedNameWithWildcard:
  QualifiedName '.*'?
;

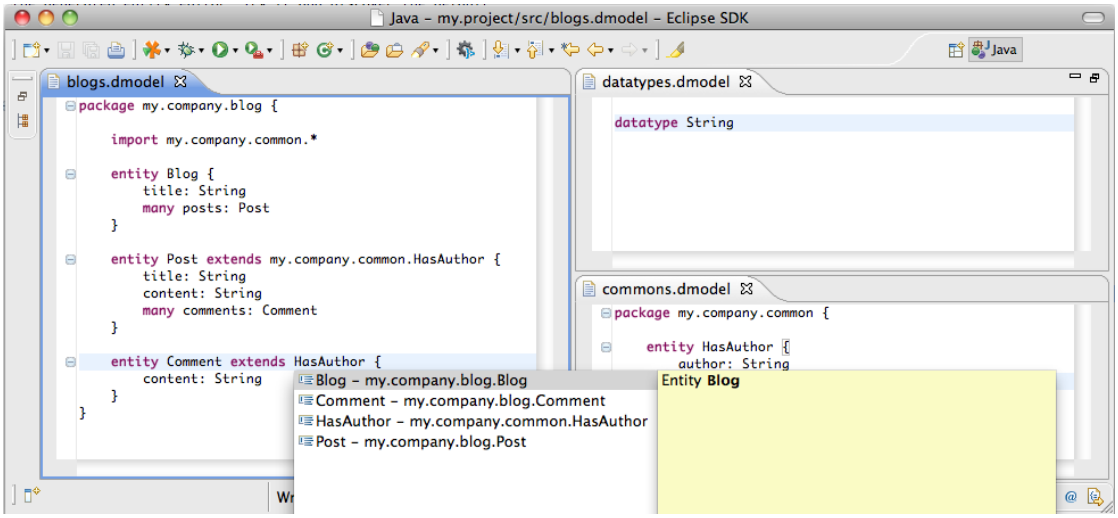
Type:
  DataType | Entity
;

DataType:
  'datatype' name=ID
;

Entity:
  'entity' name = ID
           ('extends' superType = [Entity | QualifiedName])?
  '{'
    (features += Feature)*
  '}'
;

Feature:
  (many ?= 'many')? name = ID ':' type = [Type | QualifiedName]
;
```

You should regenerate the language infrastructure as described in the previous section, and give the editor another try. You can even split up your model into smaller parts and have cross-references across file boundaries.



3. 15 Minutes Tutorial - Extended

After you have developed you first own DSL, the question arises, how the behavior and the semantics of the language can be customized. Therefore a couple of mini-tutorials are available, that illustrate common use cases when crafting an own DSL.

These lessons are independent from each other. Each of them will be based on the language that was build in the first domain model tutorial (§2).

That is, the syntax and the grammar for the language look like this:

```
package java.lang {
  datatype String
}

package my.company.blog {
  import java.lang.*
  import my.company.common.*

  entity Blog {
    title: String
    many posts: Post
  }

  entity HasAuthor {
    author: String
  }

  entity Post extends HasAuthor {
    title: String
    content: String
    many comments: Comment
  }

  entity Comment extends HasAuthor {
    content: String
  }
}
```

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

```
generate domainmodel "http://www.eclipse.org/xtext/example/Domainmodel"

Domainmodel:
  (elements += AbstractElement)*
;

PackageDeclaration:
  'package' name = QualifiedName '{'
    (elements += AbstractElement)*
  '}'
;

AbstractElement:
  PackageDeclaration | Type | Import
;

QualifiedName:
  ID ('.' ID)*
;

Import:
  'import' importedNamespace = QualifiedNameWithWildcard
;

QualifiedNameWithWildcard:
  QualifiedName '.*'?
;

Type:
  DataType | Entity
;

DataType:
  'datatype' name=ID
;

Entity:
  'entity' name = ID
            ('extends' superType = [Entity | QualifiedName])?
  '{'
    (features += Feature)*
  '}'
;

Feature:
  (many ?= 'many')? name = ID ':' type = [Type | QualifiedName]
;
```

3.1. Writing a Code Generator With Xtend

As soon as you generate the Xtext artifacts for a grammar, a code generator stub will be put into the runtime project of your language. Let's dive into Xtend and see how you can integrate your own code generator with Eclipse.

In this lesson you will generate Java Beans for entities that are defined in the domain model DSL. For each *Entity*, a Java class is generated and each *Feature* will lead to a private field in that class and public getters and setters. For the sake of simplicity, we will use fully qualified names all over the generated code.

```
package my.company.blog;

public class HasAuthor {
  private java.lang.String author;

  public java.lang.String getAuthor() {
    return author;
  }

  public void setAuthor(java.lang.String author) {
    this.author = author;
  }
}
```

First of all, locate the file *DomainmodelGenerator.xtend* in the package *org.eclipse.xtext.example.generator* This Xtend class is used to generate code for your models in the standalone scenario and in the interactive Eclipse environment.

```
package org.eclipse.xtext.example.generator

import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.IGenerator
import org.eclipse.xtext.generator.IFileSystemAccess

class DomainmodelGenerator implements IGenerator {
  override void doGenerate(Resource resource, IFileSystemAccess fsa) {
  }
}
```

Let's make the implementation more meaningful and start writing the code generator. The strategy is to find all entities within a resource and trigger code generation for each one.

1. First of all, you will have to filter the contents of the resource down to the defined entities. Therefore we need to iterate a resource with all its deeply nested elements. This can be achieved with the method getAllContents(). To use the resulting TreeIterator in a for loop, we use the extension method toIterable() from the built-in library class IteratorExtensions.

```
class DomainmodelGenerator implements IGenerator {
  override void doGenerate(Resource resource, IFileSystemAccess fsa) {
    for(e: resource.allContents.toIterable.filter(typeof(Entity))) {
      ...
    }
  }
}
```

2. Now let's answer the question, how we determine the file name of the Java class, that each *Entity* should yield. This information should be derived from the qualified name of the *Entity* since Java enforces this pattern. The qualified name itself has to be obtained from a special service that is available for each language. Fortunately, Xtend allows to reuse that one easily. We simply inject the IQualifiedNameProvider into the generator.

```
@Inject extension IQualifiedNameProvider
```

This allows to ask for the name of an entity. It is straightforward to convert the name into a file name:

```
override void doGenerate(Resource resource, IFileSystemAccess fsa) {
  for(e: resource.allContents.toIterable.filter(typeof(Entity))) {
    fsa.generateFile(
      e.fullyQualifiedName.toString("/") + ".java",
      e.compile)
  }
}
```

3. The next step is to write the actual template code for an entity. For now, the function Entity.compile does not exist, but it is easy to create it:

```
def compile(Entity e) '''
  package «e.eContainer.fullyQualifiedName»;

  public class «e.name» {
  }
'''
```

4. This small template is basically the first shot at a Java-Beans generator. However, it is currently rather incomplete and will fail, if the *Entity* is not contained in a package. A small modification fixes this. The package-declaration has to be wrapped in an IF expression:

```
def compile(Entity e) '''
  «IF e.eContainer != null»
    package «e.eContainer.fullyQualifiedName»;
  «ENDIF»

  public class «e.name» {
  }
'''
```

Let's handle the *superType* of an *Entity* gracefully, too by using another IF expression:

```
def compile(Entity e) '''
  «IF e.eContainer != null»
    package «e.eContainer.fullyQualifiedName»;
  «ENDIF»

  public class «e.name» «IF e.superType != null
        »extends «e.superType.fullyQualifiedName» «ENDIF»{
  }
'''
```

5. Even though the template will compile the *Entities* without any complains, it still lacks support for the Java properties, that each of the declared features should yield.
   For that purpose, you have to create another Xtend function that compiles a single feature to the respective Java code.

```
def compile(Feature f) '''
  private «f.type.fullyQualifiedName» «f.name»;

  public «f.type.fullyQualifiedName» get«f.name.toFirstUpper»() {
    return «f.name»;
  }

  public void set«f.name.toFirstUpper»(«f.type.fullyQualifiedName» «f.name») {
    this.«f.name» = «f.name»;
  }
'''
```

   As you can see, there is nothing fancy about this one. Last but not least, we have to make sure that the function is actually used.

```
def compile(Entity e) '''
  «IF e.eContainer != null»
    package «e.eContainer.fullyQualifiedName»;
  «ENDIF»

  public class «e.name» «IF e.superType != null
          »extends «e.superType.fullyQualifiedName» «ENDIF»{
    «FOR f:e.features»
      «f.compile»
    «ENDFOR»
  }
'''
```

The final code generator looks pretty much like the following code snippet. Now you can give it a try! Launch a new Eclipse Application (*Run As -> Eclipse Application* on the Xtext project) and create a *dmodel* file in a Java Project. Now simply create a new source folder *src-gen* in the that project and see how the compiler will pick up your sample *Entities* and generate Java code for them.

```
package org.eclipse.xtext.example.generator
```

```
import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.IGenerator
import org.eclipse.xtext.generator.IFileSystemAccess
import org.eclipse.xtext.example.domainmodel.Entity
import org.eclipse.xtext.example.domainmodel.Feature
import org.eclipse.xtext.naming.IQualifiedNameProvider

import com.google.inject.Inject

class DomainmodelGenerator implements IGenerator {

  @Inject extension IQualifiedNameProvider

  override void doGenerate(Resource resource, IFileSystemAccess fsa) {
    for(e: resource.allContents.toIterable.filter(typeof(Entity))) {
      fsa.generateFile(
        e.fullyQualifiedName.toString("/") + ".java",
        e.compile)
    }
  }

  def compile(Entity e) '''
    «IF e.eContainer != null»
      package «e.eContainer.fullyQualifiedName»;
    «ENDIF»

    public class «e.name» «IF e.superType != null
          »extends «e.superType.fullyQualifiedName» «ENDIF»{
      «FOR f:e.features»
        «f.compile»
      «ENDFOR»
    }
  '''


  def compile(Feature f) '''
    private «f.type.fullyQualifiedName» «f.name»;

    public «f.type.fullyQualifiedName» get«f.name.toFirstUpper»() {
      return «f.name»;
    }

    public void set«f.name.toFirstUpper»(«f.type.fullyQualifiedName» «f.name») {
      this.«f.name» = «f.name»;
    }
  '''
}
```

If you want to play around with Xtend, you can try to use the Xtend tutorial which

can be materialized into your workspace. Simply choose *New -> Example -> Xtend Examples -> Xtend Introductory Examples* and have a look at Xtend's features. As a small exercise, you could implement support for the *many* attribute of a *Feature* or enforce naming conventions, e.g. field names should start with an underscore.

3.2. Unit Testing the Language

Automated tests are crucial for the maintainability and the quality of a software product. That is why it is strongly recommended to write unit tests for your language, too. The Xtext project wizard creates a test project for that purpose. It simplifies the setup procedure both for the Eclipse agnostic tests and the UI tests for Junit4.

This tutorial is about testing the parser and the linker for the *Domainmodel*. It leverages Xtend to write the testcase.

1. First of all, a new Xtend class has to be created. Therefore, choose the src folder of the test plugin, and select *New -> Xtend Class* from the context menu. Provide a meaningful name and enter the package before you hit finish.
   The core of the test infrastructure is the XtextRunner and the language specific IInjectorProvider. Both have to be provided by means of class annotations:

```
import org.eclipse.xtext.junit4.XtextRunner
import org.eclipse.xtext.example.domainmodel.DomainmodelInjectorProvider

@InjectWith(typeof(DomainmodelInjectorProvider))
@RunWith(typeof(XtextRunner))
class ParserTest {
}
```

2. The actual test case is pretty straight forward with Xtend. The utility class *org.eclipse.xtext.junit4.util.ParseHelper* allows to parse an arbitrary string into a *Domainmodel*. The model itself can be traversed and checked afterwards. A static import of Assert leads to concise and readable test cases.

```
import org.eclipse.xtext.junit4.util.ParseHelper
import static org.junit.Assert.*

...
    @Inject
ParseHelper<Domainmodel> parser

@Test
def void parseDomainmodel() {
  val model = parser.parse(
    "entity MyEntity {      parent: MyEntity    }")
  val entity = model.elements.head as Entity
  assertSame(entity, entity.features.head.type)
}
```

3. After saving the Xtend file, it is time to run the test. Please locate the generated java class in the *xtend-gen* folder and select *Run As -> JUnit Test* from the context menu.

3.3. Creating Custom Validation Rules

One of the main advantages of DSLs is the possibility to statically validate domain specific constraints. This can be achieved by means of static analysis. Because this is a common use case, Xtext provides a dedicated hook for this kind of validation rules. In this lesson, we want to ensure that the name of an *Entity* starts with an upper-case letter and that all features have distinct names across the inheritance relationship of an *Entity*.

Try to locate the class *DomainmodelJavaValidator* in the package *org.eclipse.xtext.example.validation*. It can be found in the language plug-in. Defining the constraint itself is only a matter of a few lines of code:

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

Any name for the method will do. The important thing is the Check annotation that advises the framework to use the method as a validation rule. If the name starts with a lower case letter, a warning will be attached to the name of the *Entity*.

The second validation rule is straight-forward, too. We traverse the inheritance hierarchy of the *Entity* and look for features with equal names.

```
@Check
public void checkFeatureNameIsUnique(Feature f) {
  Entity superEntity = ((Entity) f.eContainer()).getSuperType();
  while(superEntity != null) {
    for(Feature other: superEntity.getFeatures()) {
      if (f.getName().equals(other.getName())) {
        error("Feature names have to be unique",
            DomainmodelPackage.Literals.FEATURE__NAME);
        return;
      }
    }
    superEntity = superEntity.getSuperType();
  }
}
```

The sibling features, that are defined in the same entity, are automatically validated by the Xtext framework. Therefore, they do not have to be checked twice.

4. Five simple steps to your JVM language

In this tutorial we will basically implement the domain model language again, but this time we will make use of the special JVM support shipped with Xtext 2.x. This kind of language really is a sweet spot for DSLs, so feel free to use this as a blueprint and add your project specific features later on.

The revised domain model language supports expressions and cross links to Java types. It is directly translated to Java source code. The syntax should look very familiar. Here is an example:

```
import java.util.List

package my.model {

  entity Person {
    name: String
    firstName: String
    friends: List<Person>
    address : Address
    op getFullName() : String {
      return firstName + " " + name;
    }

    op getFriendsSortedByFullName() : List<Person> {
      return friends.sortBy[ f | f.fullName ]
    }
  }

  entity Address {
    street: String
    zip: String
    city: String
  }
}
```

As you can see, it supports all kinds of advanced features such as Java generics and full expressions even including lambda expressions. Don't panic you will not have to implement these concepts on your own but will reuse a lot of helpful infrastructure to build the language.

We will now walk through the *five!* little steps needed to get this language fully working including its compiler.

After you have installed Xtext on your machine, start Eclipse and set up a fresh workspace.

4.1. Step One: Create A New Xtext Project

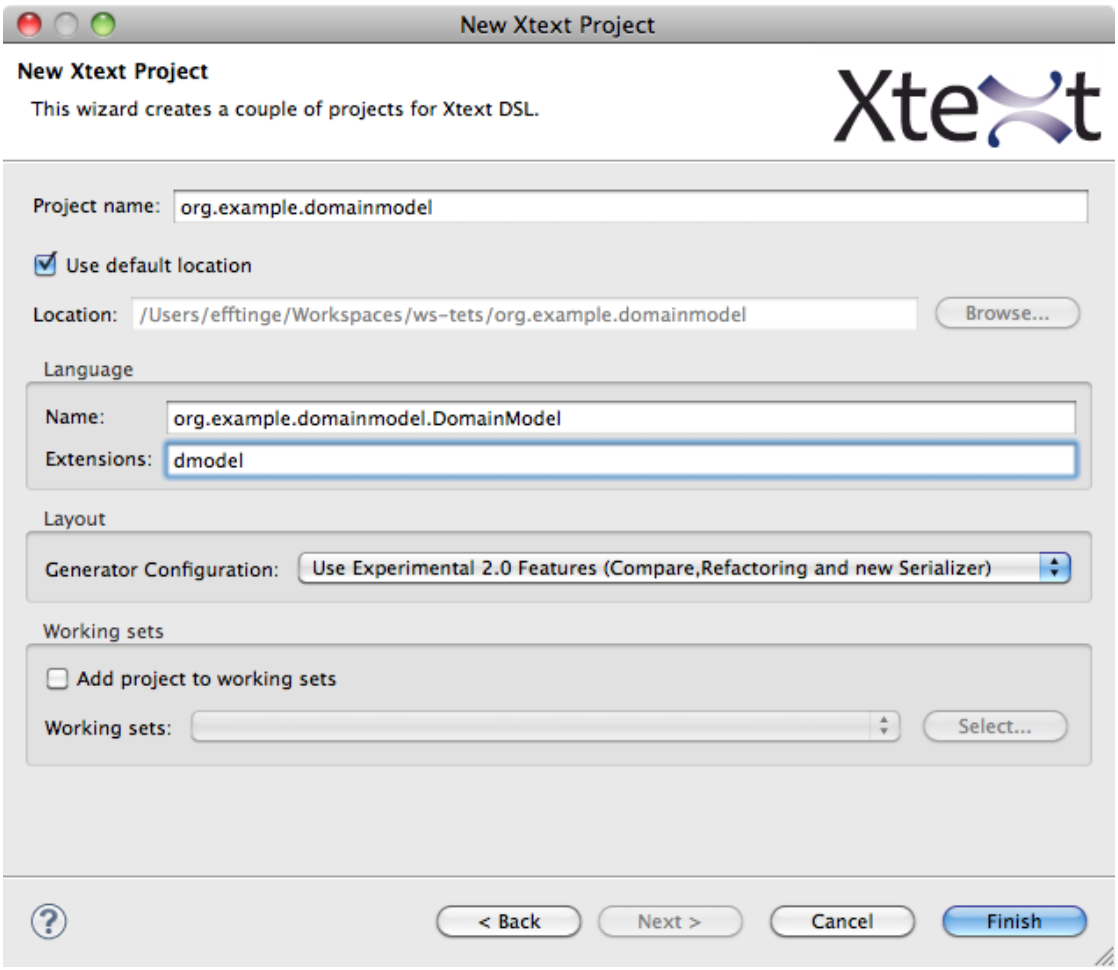
In order to get started we first need to create some Eclipse projects. Use the Eclipse wizard to do so:

File -> New -> Project... -> Xtext -> Xtext Project

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

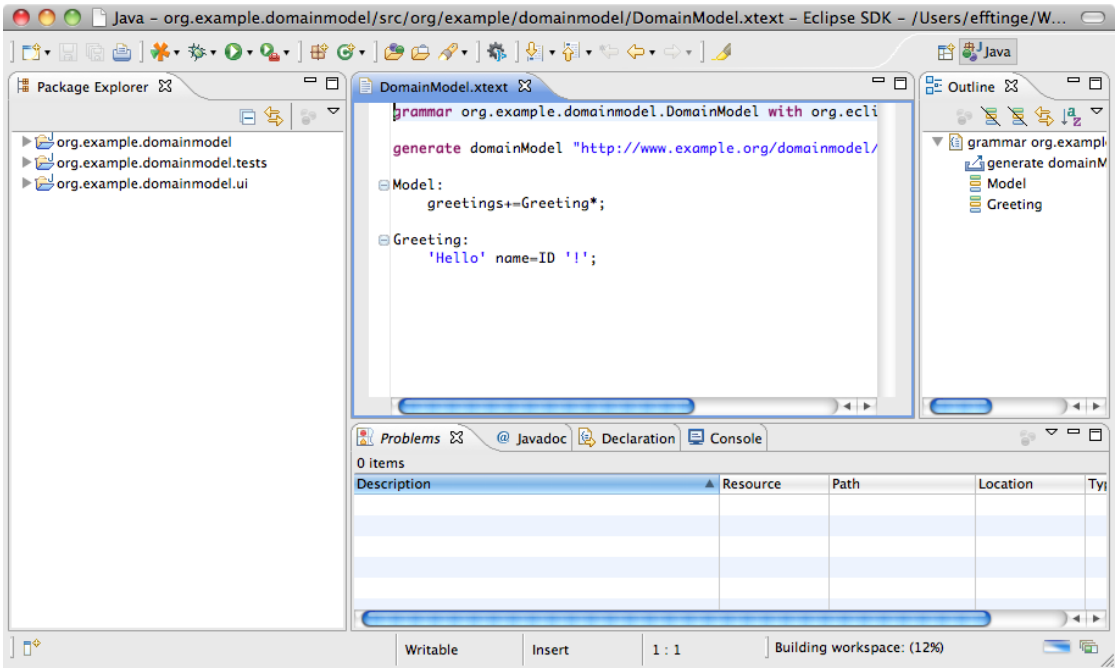
Main project name:	org.example.domainmodel
Language name:	org.example.domainmodel.Domainmodel
DSL-File extension:	dmodel

Click on *Finish* to create the projects.



After you have successfully finished the wizard, you will find three new projects in your workspace.

org.example.domainmodel	Contains the grammar definition and all runtime components (parser, lexer, linker, validation, etc.)
org.example.domainmodel.tests	Unit tests go here.
org.example.domainmodel.ui	The Eclipse editor and all the other workbench related functionality.



4.2. Step Two: Write the Grammar

The wizard will automatically open the grammar file *Domainmodel.xtext* in the editor. As you can see it already contains a simple *Hello World* grammar:

```
grammar org.example.domainmodel.Domainmodel with
                                org.eclipse.xtext.common.Terminals

generate domainmodel "http://www.example.org/domainmodel/Domainmodel"

Model:
  greetings+=Greeting*;

Greeting:
```

```
  'Hello' name=ID '!';
```

Please replace that grammar definition with the one for our language:

```
grammar org.example.domainmodel.Domainmodel with
                                    org.eclipse.xtext.xbase.Xbase

generate domainmodel "http://www.example.org/domainmodel/Domainmodel"

Domainmodel:
  elements+=AbstractElement*;

AbstractElement:
  PackageDeclaration | Entity | Import;

PackageDeclaration:
  'package' name=QualifiedName '{'
    elements+=AbstractElement*
  '}';

Import:
  'import' importedNamespace=QualifiedNameWithWildCard;

QualifiedNameWithWildCard :
  QualifiedName  ('.' '*')?;

Entity:
  'entity' name=ValidID
    ('extends' superType=JvmTypeReference)? '{'
    features+=Feature*
  '}';

Feature:
  Property | Operation;

Property:
  name=ValidID ':' type=JvmTypeReference;

Operation:
  'op' name=ValidID
    '('(params+=FullJvmFormalParameter
      (',' params+=FullJvmFormalParameter)*)?')'
    ':' type=JvmTypeReference
    body=XBlockExpression;
```

Let's have a look at what the different grammar constructs mean:

1.
```
grammar org.example.domainmodel.Domainmodel with
        org.eclipse.xtext.xbase.Xbase
```

The first thing to note is that instead of inheriting from the usual *org.eclipse.xtext.common.Terminals* grammar, we make use of *org.eclipse.xtext.xbase.Xbase*.
Xbase allows us to easily reuse and embed modern, statically typed expressions as well as Java type signatures in our language.

2.
```
Domainmodel:
  elements+=AbstractElement*;
```

The first rule in a grammar is always used as the entry or start rule.
It says that a *Domainmodel* contains an arbitrary number (*) of *AbstractElement*s which will be added (+=) to a feature called elements.

3.
```
AbstractElement:
  PackageDeclaration | Entity | Import;
```

The rule *AbstractElement* delegates to either the rule *PackageDeclaration*, the rule *Entity* or the rule *Import*.

4.
```
PackageDeclaration:
  'package' name=QualifiedName '{'
    elements+=AbstractElement*
  '}';
```

A *PackageDeclaration* is used to declare a name space which can again contain any number of *AbstractElement*s.
Xtext has built-in support for qualified names and scoping based on the hierarchy of the produced model. The default implementation will add the package names as the prefix to contained entities and nested packages. The qualified name of an *Entity* 'Baz' which is contained in a *PackageDeclaration* 'foo.bar' will be 'foo.bar.Baz'. In case you do not like the default behavior you will need to use a different implementation of IQualifiedNameProvider.

5.
```
Import:
  'import' importedNamespace=QualifiedNameWithWildCard;

QualifiedNameWithWildCard :
  QualifiedName  ('.' '*')?;
```

The rule *Import* makes use of the namespace support (§8.6.3), too. It basically allows you to get full-blown import functionality as you are used to from Java, just by having these two rules in place.

6.
```
Entity:
  'entity' name=ValidID
    ('extends' superType=JvmTypeReference)? '{'
    features+=Feature*
  '}';
```

The rule *Entity* starts with the definition of a keyword followed by a name.
The *extends* clause which is parenthesized and optional (note the trailing ?) makes use of the rule *JvmTypeReference* which is defined in a super grammar. *JvmTypeReference* defines the syntax for full Java-like type names. That is everything from simple names, over fully qualified names to fully-fledged generics, including wildcards, lower bounds and upper bounds.
Finally between curly braces there can be any number of *Features*, which leads us to the next rule.

7.
```
Feature:
  Property | Operation;
```

The rule *Feature* delegates to either a *Property* or an *Operation*.

8.
```
Property:
  name=ValidID ':' type=JvmTypeReference;
```

A *Property* has a name and makes again use of the inherited rule *JvmTypeReference*.

9.

```
Operation:
  'op' name=ValidID
  '('(params+=FullJvmFormalParameter
    (',' params+=FullJvmFormalParameter)*)?')'
  ':' type=JvmTypeReference
  body=XBlockExpression;
```

*Operation*s also have a signature as expected. Note that also for formal parameters we can reuse a rule from the super grammar. The *Operation*'s body, that is the actual implementation is defined by the rule *XBlockExpression* which is one of the most often used entry rules from Xbase. A block consist of any number of expressions between curly braces such as:

```
{
  return "Hello World" + "!"
}
```

4.3. Step Three: Generate Language Artifacts

Now that we have the grammar in place and defined we need to execute the code generator that will derive the various language components. To do so right click in the grammar editor. From the opened context menu, choose

Run As -> Generate Xtext Artifacts.

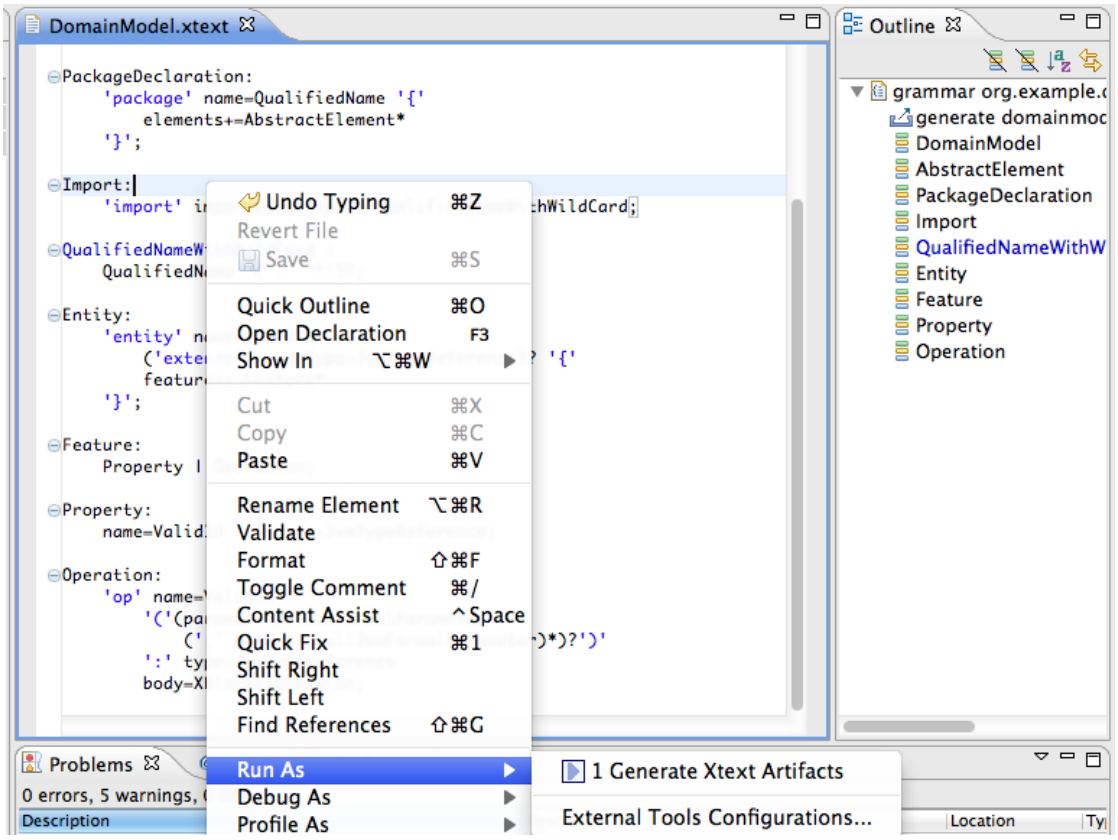
This will trigger the Xtext language generator. It generates the parser and serializer and some additional infrastructure code. You will see its logging messages in the Console View.

4.4. Step Four: Define the Mapping to JVM Concepts

The syntax alone is not enough to make the language work. We need to map the domain specific concepts to some other language in order to tell Xtext how it is executed. Usually you define a code generator or an interpreter for that matter, but languages using Xbase can omit this step and make use of the IJvmModelInferrer.

The idea is that you translate your language concepts to any number of Java types (JvmDeclaredType). Such a type can be a Java class, Java interface, Java annotation type or a Java enum and may contain any valid members. In the end you as a language developer are responsible to create a correct model according to the Java language.

By mapping your language concepts to Java elements, you implicitly tell Xtext in what kind of scopes the various expressions live and what return types are expected from them. Xtext 2.x also comes with a code generator which can translate that Java



model into readable Java code, including the expressions.

If you have already triggered the 'Generate Xtext Artifacts' action, you should find a stub called *org/example/domainmodel/jvmmodel/DomainmodelJvmModelInferrer.xtend* in the src folder. Please replace its contents with the following :

```
package org.example.domainmodel.jvmmodel

import com.google.inject.Inject
import org.example.domainmodel.domainmodel.Entity
import org.example.domainmodel.domainmodel.Operation
import org.example.domainmodel.domainmodel.Property
import org.eclipse.xtext.naming.IQualifiedNameProvider
import org.eclipse.xtext.xbase.jvmmodel.AbstractModelInferrer
import org.eclipse.xtext.xbase.jvmmodel.IJvmDeclaredTypeAcceptor
import org.eclipse.xtext.xbase.jvmmodel.JvmTypesBuilder

class DomainmodelJvmModelInferrer extends AbstractModelInferrer {

  /**
   * a builder API to programmatically create Jvm elements
```

```
 * in readable way.
 */
@Inject extension JvmTypesBuilder

@Inject extension IQualifiedNameProvider

def dispatch void infer(Entity element,
            IJvmDeclaredTypeAcceptor acceptor,
            boolean isPrelinkingPhase) {
    acceptor.accept(
    element.toClass( element.fullyQualifiedName )
  ).initializeLater [
    documentation = element.documentation
    if (element.superType != null)
      superTypes += element.superType.cloneWithProxies
    for (feature : element.features) {
      switch feature {

        Property : {
          members += feature.toField(feature.name, feature.type)
          members += feature.toGetter(feature.name, feature.type)
          members += feature.toSetter(feature.name, feature.type)
        }

        Operation : {
          members += feature.toMethod(feature.name, feature.type) [
            documentation = feature.documentation
            for (p : feature.params) {
              parameters += p.toParameter(p.name, p.parameterType)
            }
            body = feature.body
          ]
        }
      }
    }
  ]
 }
}
```

Let's go through the code to get an idea of what is going on. (Please also refer to the JavaDoc of the involved API for details, especially the JvmTypesBuilder).

1.
```
def dispatch void infer(Entity element,
        IAcceptor<JvmDeclaredType> acceptor,
        boolean isPrelinkingPhase) {
```

Using the dispatch keyword makes sure that the method is called for instances of type *Entity* only. Have a look at the Xtend documentation on polymorphic dispatch to understand Xtend's dispatch functions. Extending AbstractModelInferrer makes sure we don't have to walk the syntax model on our own.

2.
```
acceptor.accept(element.toClass(element.fullyQualifiedName)
...
```

Every JvmDeclaredType you create in the model inference needs to be passed to the *acceptor* in order to get recognized.
The extension method *toClass* comes from JvmTypesBuilder. That class provides a lot of convenient extension methods, which help making the code extremely readable and concise.
Most of the methods accept initializer blocks as the last argument, in which the currently created model element is bound to the implicit variable it. Therein you can further initialize the created Java element.

3.
```
).initializeLater [
```

The type inference has two phases. The first phase happens before linking. Only the empty types have to be created and be passed to the *acceptor*. The types' features are created in the second phase. The actions in this phase are specified in the lambda expression passed to initializeLater. Think of it as an anonymous class.

4.
```
documentation = element.documentation
```

Here for instance we assign some JavaDoc to the newly created element. The assignment is translated to an invocation of the method JvmTypesBuilder *#setDocumentation(JvmIdentifiableElement element,String documentation)* and element.documentation is in fact calling the extension method JvmTypesBuilder *#getDocumentation(EObject element)*
Xtend's extension methods are explained in detail on the Xtend website.

5.

```
if (element.superType != null)
  superTypes += entity.superType.cloneWithProxies
```

Set the *superType* on the inferred element. Note that we have to clone the type reference from the *element.superType*. If we did not do that, the type reference would be automatically removed from the *element*, as *superType* is an EMF containment reference.

6.

```
for (feature : element.features) {
  switch feature {
    Property : {
      // ...
    }
    Operation : {
      // ...
    }
  }
}
```

When iterating over a list of heterogeneous types, the switch expression with its type guards comes in handy. If *feature* is of type *Property* the first block is executed. If it is an *Operation* the second block is executed. Note that the variable *feature* will be implicitly casted to the respective type within the blocks.

7.

```
Property : {
  members += feature.toField(feature.name, feature.type)
  members += feature.toSetter(feature.name, feature.type)
  members += feature.toGetter(feature.name, feature.type)
}
```

For each *Property* we create a field as well as a corresponding getter and setter.

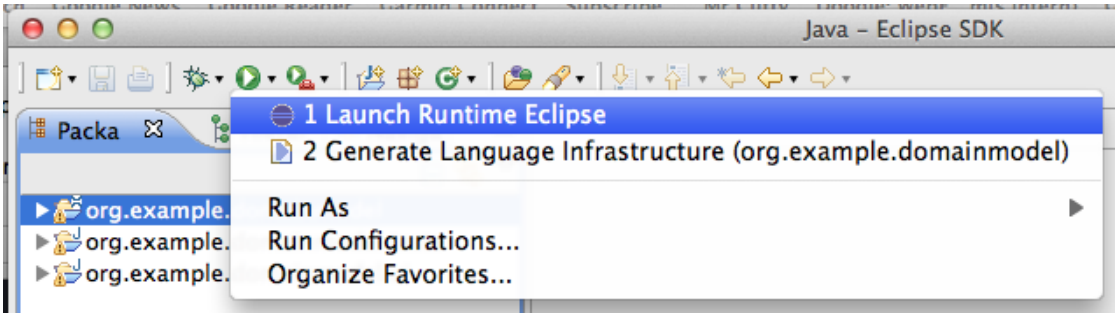
8.

```
Operation : {
  members += feature.toMethod(feature.name, feature.type) [
    documentation = feature.documentation
    for (p : feature.params) {
      parameters += p.toParameter(p.name, p.parameterType)
    }
    body = feature.body
  ]
}
```

Operations are being mapped to a corresponding Java method. The documentation is translated and the parameters are added within the initializer.
The line body = feature.body registers the *Operation*'s expression as the body of the newly created Java method. This defines the scope of the expression. The frameworks deduces the visible fields and parameters as well as the expected return type from that information.

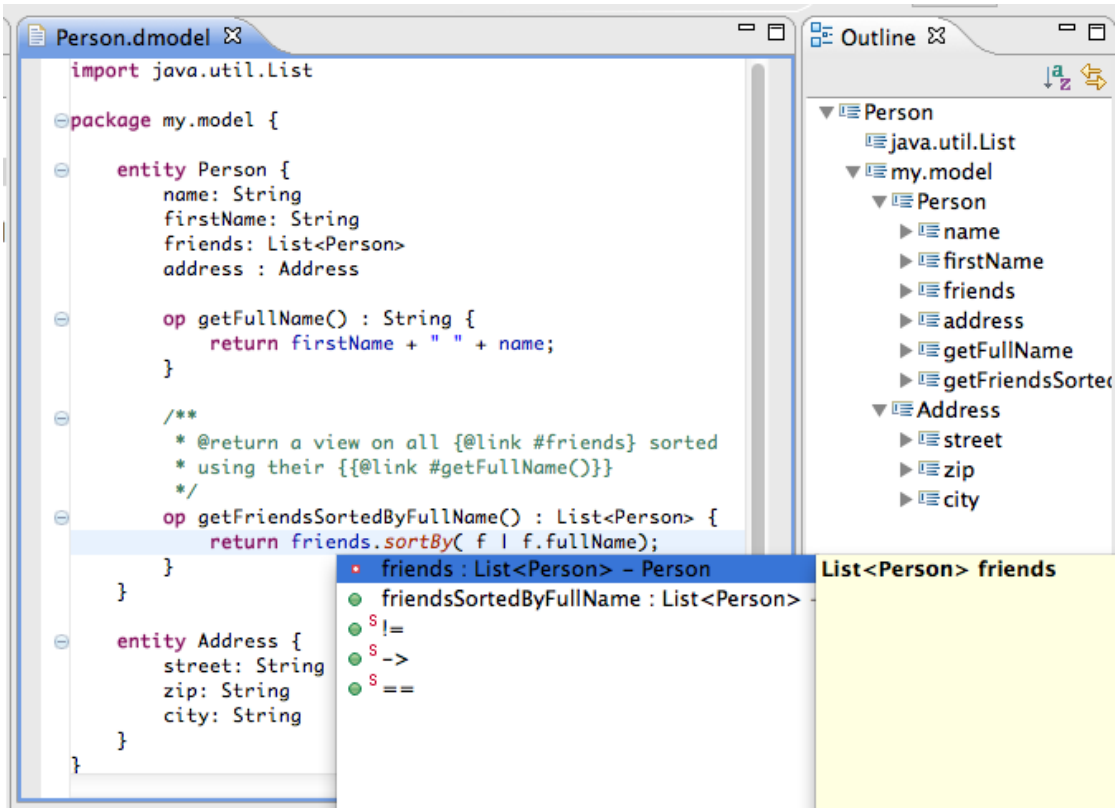
4.5. Step Five : Try the Editor!

We are now able to test the IDE integration, by spawning a new Eclipse using our plug-ins. To do so just use the launch shortcut called "Launch Runtime Eclipse", clicking on the green play button in the tool bar.



In the new workbench, create a Java project (*File -> New -> Project... -> Java Project*. Xbase relies on a small runtime library on the classpath. To add this, right-click on the project and go to *Java Build Path -> Libraries -> Add Library* and choose *Xtend Library*. The create a new file with the file extension you chose in the beginning (**.dmodel*) in the source folder of the Java project. This will open the generated entity editor. Try it and discover the rich functionality it provides. You should also have a look at the preferences of your language to find out what can be individually configured to your users needs.

Have fun!


Person.dmodel
import java.util.List
package my.model {
entity Person {
name: String
firstName: String
friends: List<Person>
address : Address
op getFullName() : String {
return firstName + " " + name;
}
/**
* @return a view on all {@link #friends} sorted
* using their {{@link #getFullName()}}
*/
op getFriendsSortedByFullName() : List<Person> {
return friends.sortBy(f | f.fullName);
}
}
entity Address {
street: String
zip: String
city: String
}
}
friends : List<Person> - Person
friendsSortedByFullName : List<Person>
!=
->
==
List<Person> friends
Outline
Person
java.util.List
my.model
Person
name
firstName
friends
address
getFullName
getFriendsSorte
Address
street
zip
city

Part II.

Reference Documentation

5. Overview

5.1. 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.

5.2. 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 behavior with your own implementations.

5.3. 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.

5.4. 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.

5.5. 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).

5.6. 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.

6. 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. 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.

6.1. 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
 end

resetEvents
  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. Some events are addiitonally marked as being reset events. 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.

6.2. The Syntax

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

6.2.1. 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 (§6.4)).

6.2.2. 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 (§12.1) for

more details) and describes the structure of the instantiated objects. Xtext can infer Ecore models from a grammar (see Ecore model inference (§6.3)) 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 turnarounds 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 EAttributes and EReferences for the different parser rules in your grammar, as described in Ecore model inference (§6.3).

EPackage Import

If you already have an existing EPackage, you can import it using its 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. It is strongly recommended to use the namespace URI instead of the resource uri because it is independent from the concrete location in the file system and much more portable across different machines, easier to configure in the workflow and works better with language mixins. The import via platform URIs or file URIs can be considered deprecated and is only supported for backwards compatibility reasons.

Using Namespace URIs to Import Existing EPackages

You can use namespace URI in order to import existing EPackage. This is generally preferrable. The package will be read from the Xtext index and therefore your grammar is independent from the concrete location of the respective ecore file. You have to make sure though, that the ecore file is contained in a project, that is managed by Xtext. Therefore the project has to have to Xtext project nature attached. Ecore files that reside in referenced Java archives (JARs) are automatically picked up and indexed if the referencing project itself is an Xtext project.

To import an EPackage, you have to state its namespace URI like this:

import "http://www.xtext.org/example/Domainmodel" as dmodel

In order to be able to find the referenced package in the language generator, some configuration values have to be set. It is usually the easiest way to register the generated

EPackage interface in the workflow. The StandaloneSetup offers the respective methods to achieve this. Simply state something like this in the workflow:

```
bean = StandaloneSetup {
  platformUri = "${runtimeProject}/../.."
  scanClassPath = true
  registerGeneratedEPackage =
   "org.eclipse.xtext.example.domainmodel.domainmodel.DomainmodelPackage"
  registerGenModelFile =
   "platform:/resource/.../path/to/Domainmodel.genmodel"
}
```

The registered genmodel is necessary to tell the code generator how the referenced Java classes are named. Please see below for alternatives that allow to register the genmodel, too. They may be handy if you create the genmodel in the workflow itself.

If the generated EPackage interface is not available when the language workflow is executed, you can use another approach to register the reference packages. This may happen if your want to generate EMF classes and the language infrastructure in one and the same workflow. The section in the workflow, that refers the your grammar, allows to set additional resources that should be loaded prior to loading the grammar file. The ecore files that contain the referenced EPackages are a good candidate for preloaded resources.

```
language = {
  loadedResource =
   "platform:/resource/.../path/to/Domainmodel.ecore"
  uri = grammarURI
}
```

You can use either platform URIs or classpath URIs to list the required ecore files (see below for details on both URI schemes).

Important note: EPackages have to refer to each other by means of platform-resource or platform-plugin URIs. Otherwise you'll get validation errors in the grammar editor. However, it'll provide quick fixes to update the ecore files accordingly. There is only one exception to the rule: If you refer to data types from the ecore package or directly to EObject, the namespace URI is valid, too. This is due to special assignability rules for these types. If you craft the EPackage manually, you'll usually face no problems due to these constraints since the reflective Ecore editor inserts platform URIs by default. The other cases and legacy packages (those that were tailored to match the restrictions of older Xtext versions) can be converted with the quick fixes in the grammar editor.

If you used platform-plugin URIs in the ecore files and cannot use the generated EPackage in the workflow, you'll have to register URI mappings from platform-plugin to platform-resource.

```
bean = StandaloneSetup {
  platformUri = "${runtimeProject}/../.."
  scanClassPath = true
  uriMap = {
    from = "platform:/plugin/some.plugin/model/File.ecore"
    to = "platform:/resource/some.plugin/model/File.ecore"
  }
  // assuming that Domainmodel.ecore uses
  // platform:/plugin/some.plugin/model/File.ecore
  registerEcoreFile =
   "platform:/resource/.../path/to/Domainmodel.ecore"
  registerGenModelFile =
   "platform:/resource/.../path/to/Domainmodel.genmodel"
}
```

If you face problems with that approach, it may be necessary to explicitly load the referenced packages in the language configuration of the workflow. You may run into this as soon as you refer to elements from *Ecore.ecore* and want to generated the EMF classes from within the same workflow.

```
language = {
  loadedResource = "platform:/resource/.../path/to/Domainmodel.ecore"
  loadedResource = "platform:/plugin/some.plugin/model/File.ecore"

  uri = "classpath:/.../path/to/Domainmodel.xtext"
  ..
}
```

Using Resource URIs to Import Existing EPackages - Deprecated

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 with resource URIs that point to the generator models (§12.2) 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 - Deprecated

We usually 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 (§10.2) for a full explanation of this useful feature.

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.

6.2.3. 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 (§6.4)). 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 (§8.7).

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 (§6.2.6), 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 IValueConverterService (cf. value converters (§8.7)). 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 (§7.2)).

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: *\n*, *\r*, *\t*, *\b*, *\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 white space rule uses alternatives like this:

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

That is a WS can be made of one or more white space 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 certain 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;

6.2.4. 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 (§6.2.3),
2. Alternatives (§6.2.3),
3. Keywords (§6.2.3) and
4. Rule Calls (§6.2.3).

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 (§6.2.4) 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 (§8.5) 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;
```

6.2.5. 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 white spaces, 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 (§6.4) 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 white space 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 white space (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 (§6.4).

6.2.6. 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 (§6.2.4), are considered to be data type rules and the data type EString is implied if none has been explicitly declared.

Value converters (§8.7) are used to transform the parsed string to the actually returned data type value.

6.2.7. 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 from 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 using an explicitly imported metamodel.

6.2.8. 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 specification 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=BooleanExpression ')'
    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 chosen predicates allow to solve most ambiguities and backtracking can often be disabled.

6.3. 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.

6.3.1. 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.

6.3.2. 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;
```

6.3.3. 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;
```

6.3.4. 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.

6.3.5. 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.

6.4. 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.

To reuse an existing grammar, make sure the grammar file is on the classpath of the inheriting language. If it is in a different plug-in, make sure to add a plug-in dependency in the MANIFEST.MF.

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

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

Statemachine: ..
```

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

Example :

```
grammar my.SuperGrammar

generate super "http://my.org/super"

...
RuleA : "a" stuff=RuleB;
RuleB : "{" name=ID "}";

grammar my.SubGrammar with my.SuperGrammar

import "http://my.org/super"

 ...

Model : (ruleAs+=RuleA)*;

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

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

In addition, you have to register the Ecore models used in your super grammar and their corresponding generator models to the MWE2 workflow of the inheriting language, e.g.

```
Workflow {
    bean = StandaloneSetup {
        scanClassPath = true
        platformUri = "${runtimeProject}/.."
        ...
        // add the FQN of the generated EPackage
        registerGeneratedEPackage = "my.supergrammar.super.SuperPackage"
        // add the platfrom resource URI of the corresponding Ecore genmodel
        registerGenModelFile = "platform:/resource/my/src-gen/my/supergrammar/super/Super.genmodel"
    }
//...
```

You might also want to read about EPackage imports (§6.2.2) for the inheriting grammar.

6.5. 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:
 .;
```

7. Configuration

7.1. 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 (§7.2.1).

Xtext's generator uses a special DSL called MWE2 - the modeling workflow engine (§11) to configure the generator.

7.1.1. 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 (§11).

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:

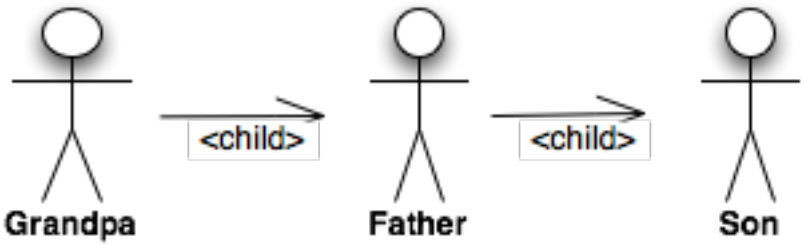
```
module com.mycompany.CreatePersons

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);
  }
}
```



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 (§11). 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).*

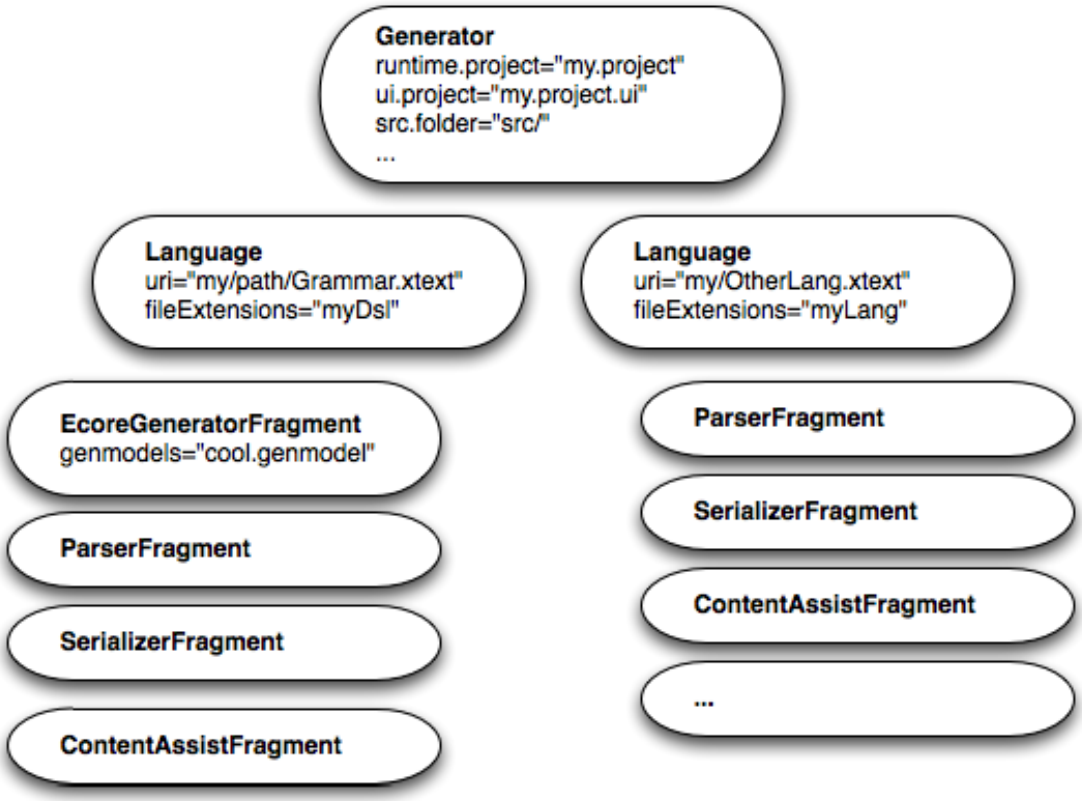
This is the basic idea of the MWE2 language. There are of course a couple of additional concepts and features in the language and we also have not yet talked about the runtime workflow model. Please refer to the dedicated MWE2 reference documentation (§11) for additional information. We will now have a look at the component model used to configure the Language Generator.

7.1.2. 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 generator fragments. The whole generator is composed of theses fragments. We have fragments for generating parsers, the serializer, the EMF code, the outline view, etc.

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 language 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 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 which 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.

7.1.3. 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*
EcoreGeneratorFragment	EMF code for generated models	Model inference (§6.3)
XtextAntlrGeneratorFragment	ANTLR grammar, parser, lexer and related services	
GrammarAccessFragment	Access to the grammar	
ResourceFactoryFragment	EMF resource factory	Xtext Resource (§12.3)
ParseTreeConstructorFragment	Model-to-text serialization	Serialization (§8.8)
ImportNamespacesScopingFragment	Index-based scoping	Index-based namespace scoping (§8.6.1)
JavaValidatorFragment	Model validation	Model validation (§8.4.2)
FormatterFragment	Code formatter	Code formatter (§8.9)
LabelProviderFragment	Label provider	Label provider (§9.1)
OutlineTreeProviderFragment	Outline view configuration	Outline (§9.5)
JavaBasedContentAssistFragment	Java-based content assist	Content assist (§9.2)
XtextAntlrUiGeneratorFragment	Content assist helper based on ANTLR	Content assist (§9.2)

7.2. 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 client 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 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.

7.2.1. 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;
}
```

Note: 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 the Binder 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);
}
```

7.2.2. Obtaining an Injector

In every application wired up with Guice there is usually one point where you initialize an Injector using the modules declared. That injector is used 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();
}
```

In Xtext, you should never instantiate the injector of your language yourself. The sections Runtime Setup (§8.1) and Equinox Setup (§8.2) explain how to access it in different scenarios.

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.

8. Runtime Concepts

Xtext itself and every language infrastructure developed with Xtext is configured and wired-up using dependency injection (§7.2). 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 ISetup-implementations in normal mode and uses Eclipse's extension mechanism when it should be configured in an OSGi environment.

8.1. Runtime Setup (ISetup)

For each language there is an implementation of ISetup generated. It implements a method called createInjectorAndDoEMFRegistration(), which can be called to do the initialization of the language infrastructure.

Caveat: The ISetup class is intended to be used for runtime and for unit testing, only. if you use it in a Equinox scenario, you will very likely break the running application because entries to the global registries will be overwritten.

The setup method returns an Injector, which can further be used to obtain a parser, etc. It also registers the Resource.Factory and generated EPackages 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.

8.2. Setup within Eclipse-Equinox (OSGi)

Within Eclipse we have a generated *Activator*, which creates a Guice Injector using the modules (§7.2.1). In addition an IExecutableExtensionFactory is generated for each language, which is used to create 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 *MyDslExecutableExtensionFactory* name followed by a colon.

```
<extension point="org.eclipse.ui.editors">
  <editor
    class="<MyDsl>ExecutableExtensionFactory:      org.eclipse.xtext.ui.editor.XtextEditor"
    contributorClass=
      "org.eclipse.ui.editors.text.TextEditorActionContributor"
    default="true"
```

```
    extensions="mydsl"
    id="org.eclipse.xtext.example.MyDsl"
    name="MyDsl Editor">
  </editor>
</extension>
```

8.3. 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.

8.4. 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.

8.4.1. 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 ISyntaxErrorMessageProvider-API to customize this messages. Any syntax errors can be retrieved from the Resource using the common EMF API:

- Resource.getErrors()
- Resource.getWarnings()

Linker: Crosslink Validation

Any broken crosslinks can be checked generically. As crosslink resolution is done lazily (see linking (§8.5)), 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:

- Resource.getErrors()
- Resource.getWarnings()

Serializer: Concrete Syntax Validation

The 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:

1. Types: only instances of *MyRule* and *MySubRule* are allowed for this rule. Subtypes are prohibited, since the parser never instantiates unknown subtypes.
2. Features: In case the *MyRule* and *MySubRule* have EStructuralFeatures besides *strVal* and *intVal*, only *strVal* and *intVal* may have non-transient values (§8.8.6).
3. Quantities: The following condition must be true: strVal.size() == intVal.size().
4. Values: It must be possible to convert all values (§8.7) to valid tokens for terminal rule *STRING*. The same is true for *intVal* and *INT*.

The typical use case for the concrete syntax validator is 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 (§8.8.3), the concrete syntax validator is executed by default before serialization. A textual Xtext editor itself 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.

- 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 IConcreteSyntaxValidator and use it directly. Furthermore, there is an adapter which allows to use 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 IConcreteSyntaxDiagnosticProvider and subclass ConcreteSyntaxDiagnosticProvider.

8.4.2. 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 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 EValidator.Registry can only be filled programmatically. That means contrary to the EPackage.Registry and the Resource.Factory.Registry there is no Equinox extension point to populate the validator registry.

For Xtext we provide a generator fragment (§7.1.2) 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 is 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 4 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 (§9.3) for details.

8.4.3. Validating Manually

As noted above, Xtext uses EMF's EValidator API to register 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);
}
```

8.4.4. Test Validators

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

Testing validators typically works as follows:

1. The test creates some models which intentionally violate some constraints.
2. The test runs some chosen @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 *MyDslFactory* that EMF generates for each EPackage, to construct the tested model elements manually. While the first option has the advantages that you can edit your models in your textual concrete syntax, the second option has the advantage that you can create partial models.

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

Validator:

```
public class MyLanguageValidator extends AbstractDeclarativeValidator {
  @Check
  public void checkFooElement(FooElement element) {
    if(element.getBarAttribute().contains("foo"))
      error("Only Foos allowed", element,
        MyLanguagePackage.FOO_ELEMENT__BAR_ATTRIBUTE, 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 test case and thereby helps to create some common state. In this example, the validator 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 method 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 AssertableDiagnostics which provides several *assert*-methods to verify whether the expected diagnostics are present:

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

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

8.5. Linking

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

1. declaration of a crosslink in the grammar (at least in the Ecore model)
2. specification of linking semantics (usually provided via the scoping API (§8.6))

8.5.1. Declaration of Crosslinks

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

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

Example:

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

The Ecore model inference (§6.3) would create an EClass *ReferringType* with an EReference *referencedObject* of type *Entity* with its containment property set to false. The referenced object would be identified either by a *STRING* and the surrounding information in the current context (see scoping (§8.6)). 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.

8.5.2. 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 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 (§8.6).

A simple implementation of the linking service 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 (§8.6).

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

```
public interface IScopeProvider {

 /**
  * Returns a scope for the given context. The scope
  * provides access to the compatible visible EObjects
  * for a given reference.
  *
  * @param context the element from which 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 IEObjectDescription, which is effectively an abstract description of a real EObject.

8.6.1. Global Scopes and Resource Descriptions

In the state machine example we don't have references across model files. Neither is there a 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

IGlobalScopeProvider. The data structures used to store its elements are described in the next section.

Resource and EObject Descriptions

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 IResourceDescription.

A IResourceDescription contains information about the resource itself which primarily its URI, a list of exported EObjects in the form of IEObjectDescriptions 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 leveraged by Xtext's indexing infrastructure in order to compute the transitive hull of dependent resources.

For users and especially in the context of scoping the most important information is the list of exported EObjects. An IEObjectDescription stores the URI of the actual EObject, its QualifiedName, as well as its 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.

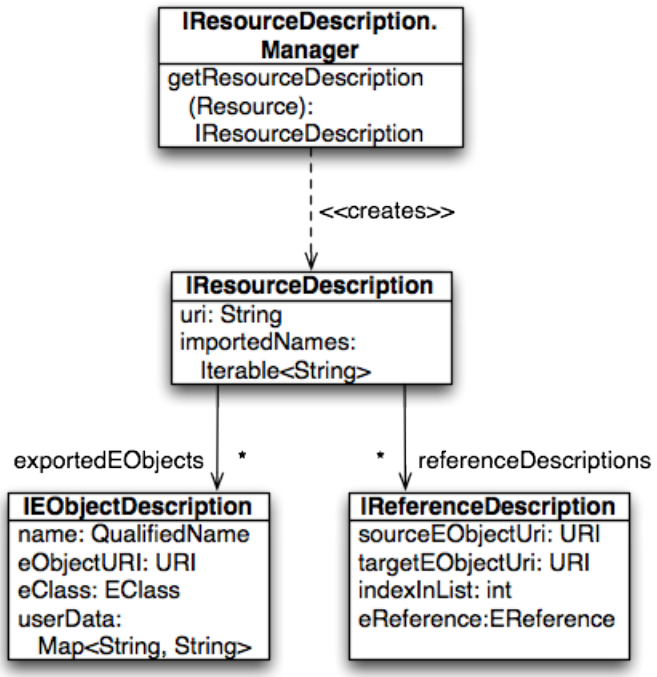


Figure 8.1.: The data model of Xtext's index

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 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 SimpleNameProvider simply returns the plain value, while the 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. To convert between a QualifiedName and its String representation you can use the IQualifiedNameConverter. Here is some Java code showing how to do that:

```
@Inject IQualifiedNameConverter converter;

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

In order to obtain an IResourceDescription.Manager it is best to ask the corresponding 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 cannot 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 are running in a Guice enabled scenario, the code looks like this:

```
@Inject
private IResourceServiceProvider.Registry rspr;

private IResourceDescription.Manager getManager(Resource res) {
  IResourceServiceProvider resourceServiceProvider =
    rspr.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 rspr =
  IResourceServiceProvider.Registry.INSTANCE;
```

However, we strongly encourage you to use dependency injection. Now, that we know how to export elements to be referable 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 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 will have to make sure that you have bound the ImportUriGlobalScopeProvider for the type IGlobalScopeProvider by the means of Guice (§7.2). 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 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 ImportURIScopingFragment in the workflow of your language. It will bind an ImportUriGlobalScopeProvider 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 the classpath 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 DefaultGlobalScopeProvider has to be bound to the IGlobalScopeProvider interface.

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.

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 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 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 IContainer as well as with a list of all IContainers which are visible from there. Note that the index is globally shared between all languages while the IContainer.Manager which 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.getResourceDescriptions()) {
      System.out.println(visibleResourceDesc.getURI());
    }
  }
}
```

Xtext ships two implementations of IContainer.Manager which are as usual bound with Guice: The default binding is to SimpleResourceDescriptionsBasedContainerManager, 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 StateBasedContainerManager and an additional 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 IAllContainersState of the referring language. These will be described in detail in the following sections.

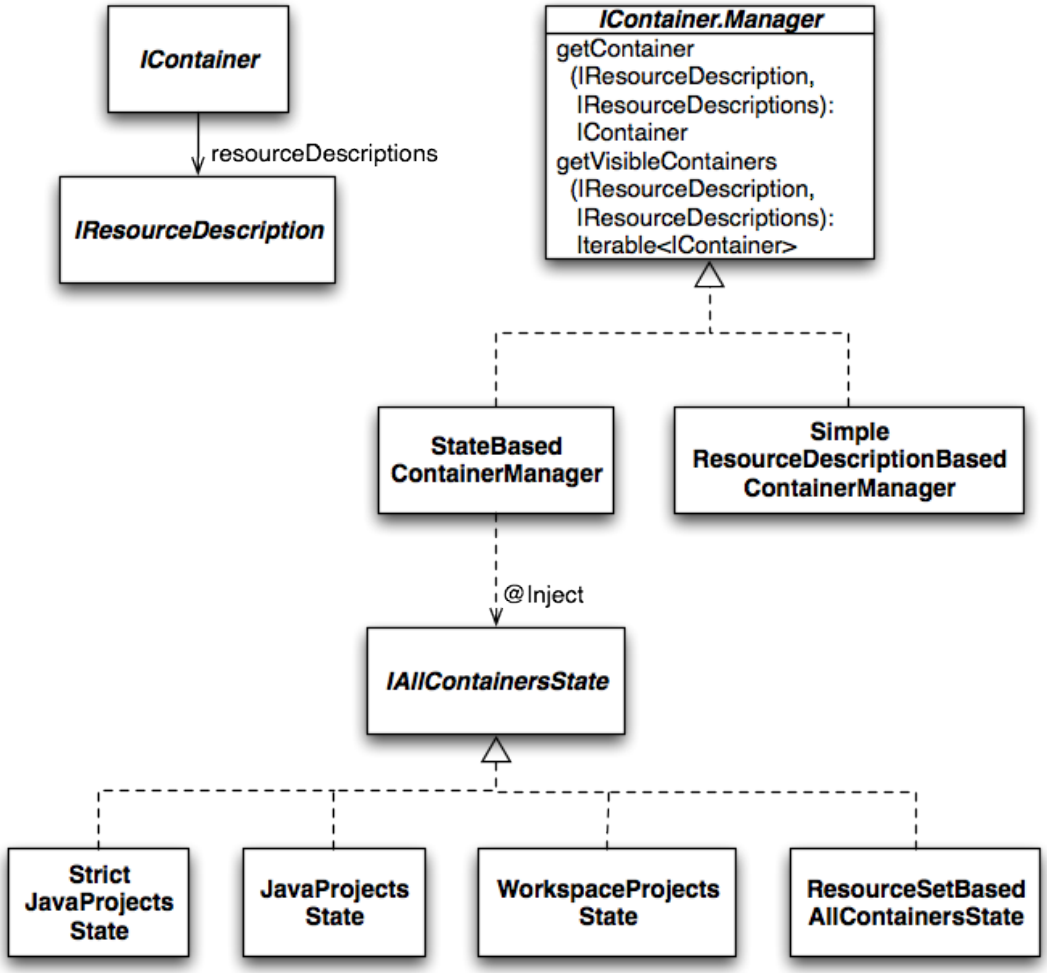


Figure 8.2.: IContainer Management

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

- 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();
}
```

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. A 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 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 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"

  ...
}
```

8.6.2. 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 AbstractDeclarativeScopeProvider 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.

8.6.3. 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 ImportedNamespaceAwareLocalScopeProvider.

IQualifiedNameProvider

The ImportedNamespaceAwareLocalScopeProvider makes use of the so called IQualifiedNameProvider service. It computes QualifiedNames for EObjects. A qualified name consists of several segments

The default implementation uses a simple name look up composes the qualified name of the simple names of all containers and the object itself.

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 QualifiedName qualifiedName(Transition t) {
    if(t.getEvent() == null || !(t.eContainer() instanceof State))
      return null;
    else
      return QualifiedName.create((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.

8.7. Value Converter

Value converters are registered to convert the parsed text into a data type instance and vice versa. The primary hook is the IValueConverterService and the concrete implementation can be registered via the runtime Guice module (§7.2.1). Simply override the corresponding binding in your runtime module like shown in this example:

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

The most simple way to register additional value converters is to make use of AbstractDeclarativeValueCon , which allows to declaratively register an IValueConverter by means of an annotated method.

```
@ValueConverter(rule = "MyRuleName")
public IValueConverter<MyDataType> getMyRuleNameConverter() {
  return new MyValueConverterImplementation();
}
```

If you use the common terminals grammar org.eclipse.xtext.common.Terminals you should extend the DefaultTerminalConverters and override or add 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 . The delegating converter reuses the functionality of the corresponding EMF EFactory.

Many languages introduce a concept for qualified names, i.e. names composed of namespaces separated by a delimiter. Since this is such a common use case, Xtext provides an extensible converter implementation for qualified names. The QualifiedNameValueConverter handles comments and white space gracefully and is capable to use the appropriate value converter for each segment of a qualified name. This allows for 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 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.

8.8. 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:

1. Validating the semantic model. This is optional, enabled by default, done by the concrete syntax validator (§8.4.1) and can be turned off in the save options (§8.8.4).
2. Matching the model elements with the grammar rules and creating a stream of tokens. This is done by the parse tree constructor (§8.8.3).
3. Associating comments with semantic objects. This is done by the comment associator (§8.8.5).
4. Associating existing nodes from the node model with tokens from the token stream.
5. Merging existing white space (§8.8.9) and line-wraps into the token stream.
6. Adding further needed white space or replacing all white space using a formatter (§8.9).

Serialization is invoked when calling XtextResource.save(..). Furthermore, the Serializer provides resource-independent support for serialization. Another situation that triggers serialization is applying Quick Fixes (§9.3) with semantic modifications. Serialization is *not* called when a textual editors contents is saved to disk.

8.8.1. The Contract

The contract of serialization says that a model which is saved (serialized) to its textual representation and then loaded (parsed) again yields a new model that is equal to the original model. Please be aware that this does *not* imply, that loading a textual representation and serializing it back produces identical textual representations. However,

the serialization algorithm tries to restore as much information as possible. That is, if the parsed model was not modified in-memory, the serialized output will usually be equal to the previous input. Unfortunately, this cannot be ensured for each and every case. A use case where is is hardly possible, is shown in the following example:

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

The given *MyRule* 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 other objects.

8.8.2. 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

- preserves existing white spaces (§8.8.9) from the node model.
- preserves existing comments (§8.8.5) 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 IEObjectDescriptions for the same object), the serializer tries to keep the name that was used in the input file.
- preserves the representation of values: For values handled by the value converter (§8.7), 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.

8.8.3. Parse Tree Constructor

The parse tree constructor usually does not need to be customized since it is automatically derived from the Xtext Grammar (§6). 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 (§8.4.1).

If a model can not be serialized, an XtextSerializationException is thrown. Possible reasons are listed below:

- 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 (§8.8.6) 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. Subtypes are not allowed.
- Value conversion (§8.7) fails. The value converter can indicate that a value is not serializeable by throwing a 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:

- 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.

8.8.4. Options

SaveOptions can be passed to XtextResource.save(options) and to Serializer.serialize(..). Available options are:

- *Formatting*. Default: false. If enabled, it is the formatters (§8.9) job to determine all white space information during serialization. If disabled, the formatter only defines white space information for the places in which no white space 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 (§8.4.1) before serializing the model.

8.8.5. Preserving Comments from the Node Model

The 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 behaves as follows, but can be customized:

- 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.

8.8.6. 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 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.

8.8.7. 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 (§8.8.3) needs to decide which value to write during serialization. This decision can be be made by customizing the IValueSerializer.serializeUnassignedValue(EObject, RuleCall, INode).

8.8.8. 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 ITokenSerializer . 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.

8.8.9. Merge White Space

After the parse tree constructor (§8.8.3) has done its job to create a stream of tokens which are to be written to the textual representation, and the comment associator (§8.8.5) has done its work, existing white space 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 white space information in between is kept. In all other cases it is up to the formatter (§8.9) to calculate new white space information.

8.8.10. Token Stream

The parse tree constructor (§8.8.3) and the formatter (§8.9) use an ITokenStream for their output, and the latter for its input as well. This allows for chaining the two components. Token streams can be converted to a String using the TokenStringBuffer and to a Writer using the WriterTokenStream.

```
public interface ITokenStream {

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

8.9. Formatting (Pretty Printing)

A formatter can be implemented via the IFormatter service. Technically speaking, a formatter is a Token Stream (§8.8.10) which inserts/removes/modifies hidden tokens (white space, line-breaks, comments).

The formatter is invoked during the serialization phase (§8.8) and when the user triggers formatting in the editor (for example, using the CTRL+SHIFT+F shortcut).

Xtext ships with two formatters:

- The OneWhitespaceFormatter simply writes one white space between all tokens.
- The AbstractDeclarativeFormatter allows advanced configuration using a FormattingConfig. Both are explained below.

A declarative formatter can be implemented by subclassing AbstractDeclarativeFormatter , 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 FormattingConfig.

The FormattingConfig consist of general settings and a set of formatting instructions:

8.9.1. General FormattingConfig Settings

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.

8.9.2. FormattingConfig Instructions

Per default, the declarative formatter inserts one white space 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:

- after(element): The instruction is applied after the grammar element 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(element): 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 white space before the comma.

- around(element): This is the same as before(element) combined with after(element).
- between(left, right): This matches if *left* directly follows *right* in the document. There may be no other tokens in between *left* and *right*.
- bounds(left, right): This is the same as after(left) combined with before(right).
- range(start, end): The rule is enabled when *start* is matched, and disabled when *end* is matched. Thereby, the rule is active for the complete region which is surrounded by *start* and *end*.

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 *element* 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(element), after(element), 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:

- 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 (§8.7).
- Grammar elements nested in CrossReference.

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

- setIndentationIncrement() increments indentation by one unit at this position. Whether one unit consists of one tab-character or spaces is defined by IIndentationInformation. The default implementation consults Eclipse's IPreferenceStore.
- 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 def, 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 *def* 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 white space, tabs or newlines, a small puppy will die somewhere in this world.
- setNoSpace(): Suppresses the white space between tokens at this position. Be aware that between some tokens a white space is required to maintain a valid concrete syntax.

8.9.3. 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 IGrammarAccess provides convenience methods for this. The find methods are available for the grammar and for each parser rule.

- 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.

8.10. Fragment Provider (Referencing Xtext Models From Other EMF Artifacts)

Although inter-Xtext linking is not done by URIs, you may want to be able to reference your EObject from non-Xtext models. In those cases URIs are used, which are made up of a part identifying the resource 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 to deal with 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 an 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 IFragmentProvider

.Fallback interface allows to delegate to the default strategy - which usually uses the fragment paths described above.

The following snippet shows how to use qualified names as fragments:

```
public QualifiedNameFragmentProvider implements IFragmentProvider {

  @Inject
  private IQualifiedNameProvider qualifiedNameProvider;

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

8.11. 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 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. the unicode escape sequences *\uXXXX* in Java.

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

8.11.1. 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"
  ...
```

8.11.2. Encoding at Language Runtime

As each language could handle the encoding problem differently, Xtext offers a service here. The 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;
}
```

8.11.3. Encoding of an XtextResource

An 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);
```

8.11.4. Encoding in New Model Projects

The 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. If your language enforces a special encoding that ignores the workspace settings, you'll have to make sure that your wizard uses the right encoding by yourself.

8.11.5. 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.

9. IDE Concepts

For the following part we will refer to the state machine example (§6.1) to describe the different aspects of Xtext's UI features.

9.1. Label Provider

There are various places in the UI in which model elements have to be presented to the user: In the outline view (§9.5), in hyperlinks (§9.6), in content proposals (§9.2), find dialogs etc. Xtext allows to customize each of these appearances by individual implementation of the ILabelProvider interface.

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 DelegatingStyledCellLabelProvider.IStyledLabelProvider, which returns a StyledString (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 AbstractLabelProvider 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 AbstractLabelProvider helps you managing the images: In your implementation of doGetImage(Object) you can as well return an Image, an ImageDescriptor 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 PluginImageHelper to learn about the customizing possibilities.

If you have the 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 (§9.1.1) and an {MyLang}DescriptionLabelProvider (§9.1.2) which you can implement manually.

9.1.1. Label Providers For EObjects

The EObject label provider 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 configureContentProposalLabelProvider(..) in your language's UI module:

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

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 dependency injection for its initialization, e.g.

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

DefaultEObjectLabelProvider

The default implementation of the ILabelProvider interface utilizes the polymorphic dispatcher idiom to implement an external visitor as the requirements of the label provider are kind of a best match for this pattern. It boils 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 SecretCompartmentsLabelProvider
    extends DefaultLabelProvider {

  public String text(Event event) {
    return event.getName() + " (" + event.getCode() + ")";
  }

  public String image(Event event) {
    return "event.gif";
  }

  public String image(State state) {
    return "state.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().

9.1.2. 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 (§8.6.1) 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 DefaultDescriptionLabelProvider. It employs the same polymorphic dispatch mechanism as the DefaultEObjectLabelProvider (§9.1.1). The default text of an IEObjectDescription 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. Instances of IResourceDescription 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);
}
```

9.2. Content Assist

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

- An abstract proposal provider class named Abstract{MyLang}ProposalProvider generated into the *src-gen* folder within the *ui* project, and

- a concrete subclass in the *src*-folder of the *ui* project called {MyLang}ProposalProvider

First we will investigate the generated Abstract{MyLang}ProposalProvider with methods that look like this:

```
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 class contains a *complete**-method for each assigned feature in the grammar and for each rule. The braces in the snippet are place-holders that should give a clue about the naming scheme used to create the various entry points for implementors. The generated proposal provider falls back to some default behavior for cross-references and keywords. Furthermore it inherits the logic that was introduced in grammars that were mixed into the current language.

Clients who want to customize the behavior may override the methods from the AbstractJavaBasedContentProposalProvider or introduce new methods with a specialized first parameter. The framework inspects the type of the model object and dispatches method calls 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.

To provide a dummy proposal for the code of an event instance, you may introduce a specialization of the generated method and implement it as follows. This will propose ZonkID for an event with the name Zonk.

```
public void completeEvent_Code(
  Event event, Assignment assignment,
  ContentAssistContext context, ICompletionProposalAcceptor acceptor) {
  // call implementation of superclass
  super.completeEvent_Code(model, assignment, context, acceptor);

  // compute the plain proposal
  String proposal = event.getName() + "ID";
```

```
  // Create and register the completion proposal:
  // The proposal may be null as the createCompletionProposal(..)
  // methods check for valid prefixes and terminal token conflicts.
  // The acceptor handles null-values gracefully.
  acceptor.accept(createCompletionProposal(proposal, context));
}
```

9.3. Quick Fixes

For validations written using the AbstractDeclarativeValidator (§8.4.2) 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 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 domain model 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 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 IModification. The 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 ISemanticModification 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]));
      }
    }
  );
```

```
}
```

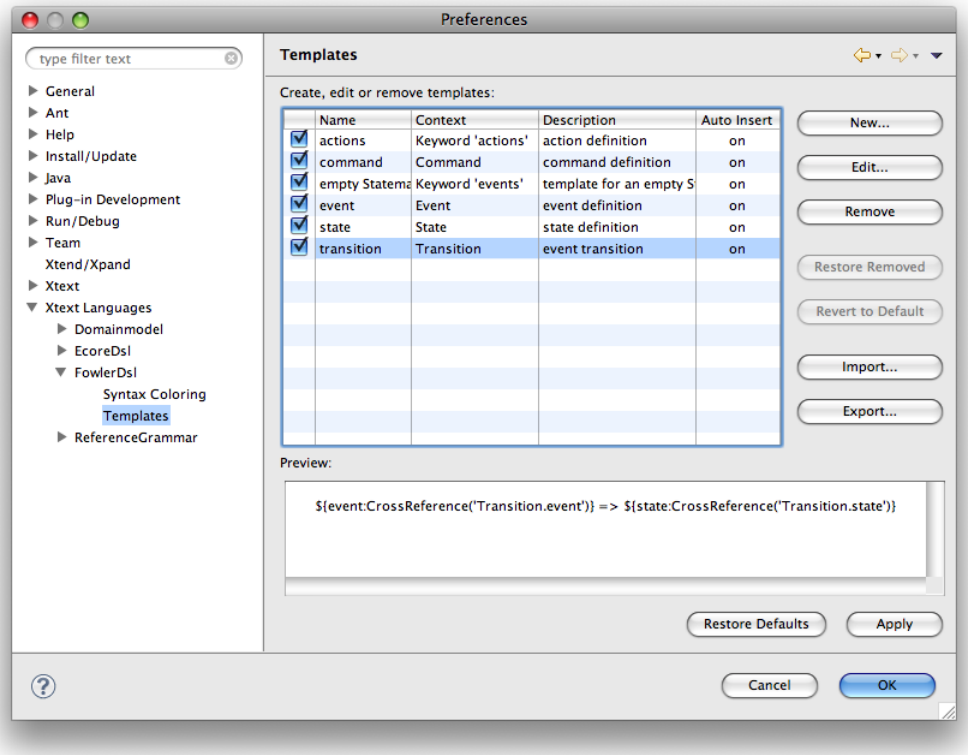
9.3.1. Quickfixes for Linking Errors and Syntax Errors

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

Hence, there is the ISyntaxErrorMessageProvider to assign issue codes to syntactical errors.

9.4. 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 *context types* that follow certain patterns. A context type will be created

1. for each rule ({languageName}.{RuleName}) and
2. for each keyword ({languageName}.kw_{keyword}).

If you don't like these defaults you'll have to subclass XtextTemplateContextTypeRegistry and configure it via Guice (§7.2.1).

In addition to the standard template proposal extension mechanism, Xtext ships with a predefined set of TemplateVariableResolvers to resolve special variable types in templates. Besides the standard template variables available in GlobalTemplateVariables like ${user}, ${date}, ${time}, ${cursor}, etc., these TemplateVariableResolvers support the automatic resolving of cross references enumeration values. Both resolvers are 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. Note that the attribute name is case sensitive. As always in eclipse plug-in development, if the folder *templates* did not exist before, you have to add it to the *bin.includes* in your *build.properties*.

9.4.1. Cross Reference Template Variable Resolver

Xtext comes with a specific template variable resolver called CrossReferenceTemplateVariableResolver , which can be used to pre-populate placeholders for cross-references within a template. The respective template variable is called *CrossReference* and its syntax is as follows:

${<displayText>:CrossReference([<MyPackageName>.]<MyType>.<myRef>)}

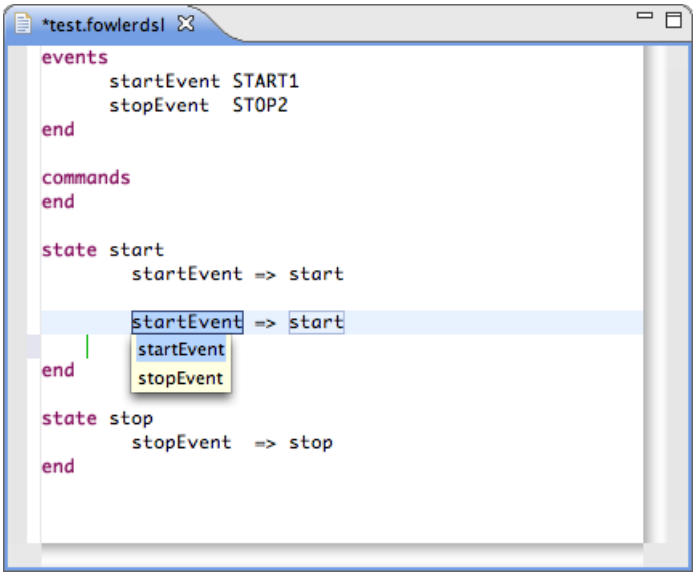
This small example yields the text *event => state* and allows selecting any events and states using a drop down:

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

9.4.2. Enumeration Template Variable Resolver

The EnumTemplateVariableResolver resolves a template variable to EEnumLiterals which are assignment-compatible to the enumeration type declared as the first parameter of the the *Enum* template variable.

The syntax is as follows:



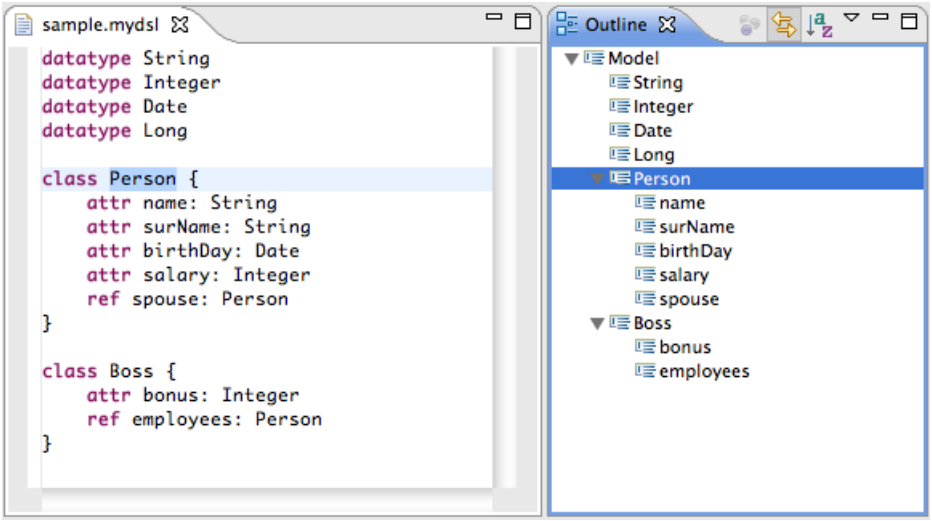
${<displayText>:Enum([<MyPackage>.]<EnumType>)
For example the following template (taken from another example):

```
<template
  name="Entity"
  description="template for an Entity"
  id="entity"
  context="org.eclipse.xtext.example.Domainmodel.Entity"
  enabled="true">
${public:Enum('Visibility')} entity ${Name} {
  ${cursor}
}
</template>
```

yields the text public entity Name {} where the text public is the default value of the Visibility. The editor provides a drop down that is populated with the other literal values as defined in the EEnum.

9.5. 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 IOutlineTreeProvider.

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

9.5.1. Influencing the outline structure

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

An EObjectNode represents a model element. By default, Xtext creates an 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: 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 AbstractOutlineNode. To instantiate these, you have to implement _createNode(parentNode, semanticElement) with the appropriate parameter types.

9.5.2. Styling the outline

You can also customize the icons and texts for an outline node. By default, Xtext uses the label provider (§9.1) 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 DefaultOutlineTreeProvider.

Note that the method _text(modelElement) can return a String or a StyledString. The StylerFactory can be used to create StyledStrings, like in the following 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 PluginImageHelper.

9.5.3. 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 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);
}
```

9.5.4. 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 OutlineFilterAndSorter .IComparator.

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 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;
}
```

9.5.5. 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 QuickOutlineFragment into your workflow.

9.6. 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.

9.6.1. 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 ILocationInFileProvider. The default implementation implements a best effort strategy which can be summarized as:

1. If the model element's type 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 node model contains any top-level tokens corresponding to invocations of the rule *ID* 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.

The location service offers different methods to obtain the region of interest for special use cases. You can either obtain the complete region for an object or only the identifying string which is usually the name of the instance (see getSignificantTextRegion(EObject)). You can also query for the text region of a specific EStructuralFeature by means of getFullTextRegion(EObject, EStructuralFeature, int).

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 (§7.2.1) 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 the

DefaultLocationInFileProvider and override the methods getIdentifierFeature() or useKeyword() to guide the first and last steps of the strategy as described above (see XtextLocationInFileProvider for an example).

9.6.2. Customizing Available Hyperlinks

The hyperlinks are provided by the 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.

9.7. Syntax Coloring

Besides the already mentioned advanced features like content assist (§9.2) and code formatting (§8.9) 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 unintrusive 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.

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

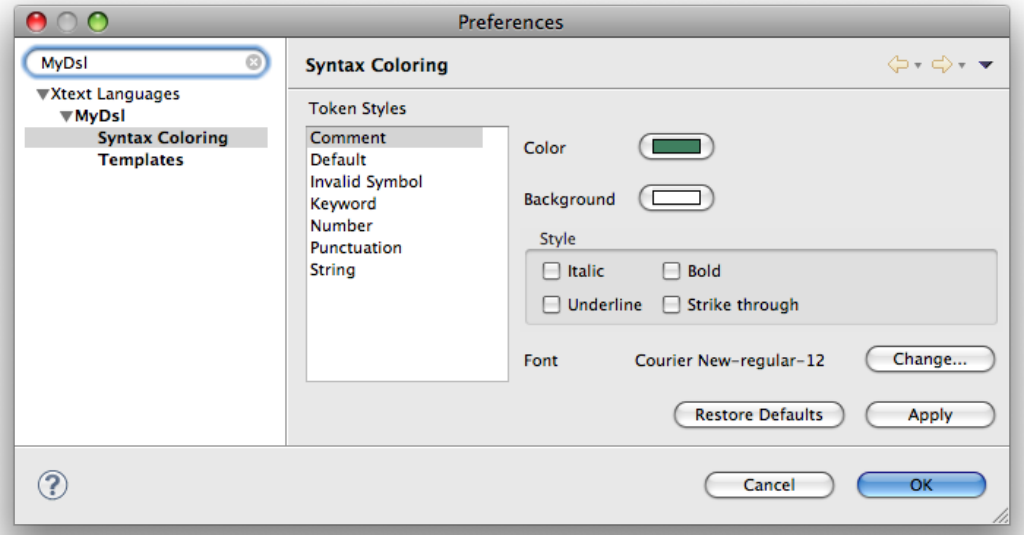
The highlighting is done in two stages. This allows for sophisticated algorithms that are executed asynchronously to provide advanced coloring while simple pattern matching may be used to highlight parts of the text instantaneously. The latter is called lexical highlighting while the first is based on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

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

9.7.1. Lexical Highlighting

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

The 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 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 AbstractAntlrTokenToAttributeIdMapper. To get an idea about it, have a look at the DefaultAntlrTokenToAttributeIdMapper.

9.7.2. Semantic Highlighting

The semantic highlighting stage is executed asynchronously in the background and can be used to calculate 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 IHighlightingConfiguration. The ISemanticHighlightingCalculator is the primary hook to implement the logic that will compute to-be-highlighted ranges based on the model elements.

The framework will pass the current XtextResource and an IHighlightedPositionAcceptor 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 in case of errors in the model. The implementor's task is to navigate the 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 || resource.getParseResult() == null)
    return;

  INode root = resource.getParseResult().getRootNode();
  for (INode node : root.getAsTreeIterable()) {
    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 an own style for each 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 ISemanticHighlightingCalculator should be aware of performance to ensure a good user experience. It is probably not a good idea to traverse everything of your model when you will only register a few highlighted ranges that can be found easier with some typed method calls. It is strongly advised to use purposeful ways to navigate your model. The parts of Xtext's core that are responsible for the semantic highlighting are pretty optimized in this regard as well. The framework will only update the ranges that actually have been altered, for example. This 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.

9.8. Rename Refactoring

Xtext provides rename refactoring of the elements in your langauge. That includes

- a command, handlers and keybindings on both declarations and references,
- inplace linked editing for the new name,
- validation and preview,
- renaming of declaration and all references even across language boundaries.

To enable refactoring support make sure the RefactorElementNameFragment is enabled in the fragment section of the MWE workflow of your language, e.g.

```
// rename refactoring
fragment = refactoring.RefactorElementNameFragment {}
```

The fragment has an additional flag useJdtRefactoring which can be used to delegate to JDT's refactoring infrastructure for languages using Xbase (§10) and an inferred JVM model (§10.4) (i.e. the domain model example or Xtend).

If you have stuck to the defaults with regard to naming, cross-referencing, and indexing rename refactoring should not need any customization. Give it a try.

9.8.1. Customizing

The most likely component you want to customize is the IRenameStrategy. This component defines how the declaration of the target element is performed. It has two major responsibilities:

- Apply and revert the declaration change on the semantic model (methods applyDeclarationChange and revertDeclarationChange). The default is to look for an EAttribute name on the target object and set its value using EMFs reflective API.
- Create the LTK Change objects of the declaration change. These changes will be aggregated, checked for overlaps, presented to you in the preview and finally executed if you apply the refactoring. The default is to use the ILocationInFileProvider to locate the text range representing the name and create a ReplaceEdit for it.

As the IRenameStrategy is a stateful object, you have to bind a custom IRenameStrategy .Provider to create it.

The second component you might want to customize is the IDependentElementsCalculator . Dependent elements are those elements whose name change when the target element is renamed. For example, when you rename a Java class the qualified names of its inner classes change, too, thus references to these have to be updated as well. This calculation is performed by the IDependentElementsCalculator. By default, all elements contained in the target element are added. This matches Xtext's default strategy of qualified name computation.

9.8.2. Rename Participants

One refactoring can trigger another: When renaming a rule in an Xtext grammar, the returned EClass should be renamed, too. For these cases, you can register a RenameParticipant by the common means of LTK. If the target of the participant is Xtext based, you can use a AbstractProcessorBasedRenameParticipant.

10. Xtext and Java

The following chapter demonstrates how to integrate your own DSL with Java. We will do this in four stages: First, you will learn how to refer to existing Java elements from within your language. Then you will use Xbase to refer to generic types. In the third step, you will map your own DSL's concepts to Java concepts. Last but not least, you will use both Java types and your concepts within Xbase expressions and execute it.

Throughout this chapter, we will step by step improve the domain model example from the tutorial (§3).

10.1. Plug-in Setup

In the following, we are going to use the JVM types model and the Xbase language library. Have a look at your MWE2 workflow and make sure that

- the Xbase models are registered in the standalone setup and
- the TypesGeneratorFragment and the XbaseGeneratorFragment are enabled.

```
bean = StandaloneSetup {
  ...
  registerGeneratedEPackage = "org.eclipse.xtext.xbase.XbasePackage"
  registerGenModelFile = "platform:/resource/org.eclipse.xtext.xbase/model/Xbase.genmodel"
}
...
fragment = types.TypesGeneratorFragment {}
fragment = xbase.XbaseGeneratorFragment {}
```

To avoid running out of memory when regenerating, make sure to run the workflow with reasonably sized heap and PermGen space. We recommend at least

-Xmx512m -XX:MaxPermSize=128m

in the *VM Arguments* section of the *Arguments* tab of the run configuration. If you are experiencing ambiguity warnings from Antlr, the usual countermeasures (§6.2.8) apply. The launch configuration that you get with a new Xtext project is already configured properly.

10.2. Referring to Java Elements using JVM Types

A common case when developing languages is the requirement to refer to existing concepts of other languages. Xtext makes this very easy for other self defined DSLs. However, it is often very useful to have access to the available types of the Java Virtual Machine as well. 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, too.

The implementation will be selected transparently depending on how the client 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 transparently for the clients behind the scenes via different implementations that are bound to specific interfaces by means of Google Guice.

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 jvmTypes
```

The next step is to actually refer to an imported concept. Let's define a mapping to available Java types for the simple data types in the domain model language. This can be done with a simple cross-reference:

```
// simple cross reference to a Java type
DataType:
  'datatype' name=ID
  'mapped-to' javaType=[jvmTypes::JvmType|QualifiedName];
```

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

```
datatype 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. Even the *import* statements will also apply for Java types.

10.2.1. Customization Points

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

The 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 types and return types, generics, available annotations or enumeration literals, it is very easy to define constraints for the referred types.

The 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 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.

10.3. Referring to Java Types Using Xbase

While the JVM types approach from the previous chapter allows to refer to any Java element, it is quite limited when it comes to generics. Usually, a type reference in Java can have type arguments which can also include wildcards, upper and lower bounds etc. A simple cross-reference using a qualified name is not enough to express neither the syntax nor the structure of such a type reference.

Xbase offers a parser rule *JvmTypeReference* which supports the full syntax of a Java type reference and instantiates a JVM element of type JvmTypeReference. So let us start by inheriting from Xbase:

```
grammar org.eclipse.xtext.example.Domainmodel
  with org.eclipse.xtext.xbase.Xbase
```

Because we can express all kinds of Java type references directly now, an indirection for *DataTypes* as in the previous section is no longer necessary. If we start from the domain model example in the tutorial (§3) again, we have to replace all cross-references to *Types* by calls to the production rule *JvmTypeReference*. The rules *DataType*, *Type*, and *QualifiedName* become obsolete (the latter is already defined in Xbase), and the *Type* in *AbstractEntity* must be changed to *Entity*. As we now have all kinds of generic Java collections at hand, *Feature.many* is obsolete, too. The whole grammar now reads:

```
grammar org.eclipse.xtext.example.Domainmodel with
                                org.eclipse.xtext.xbase.Xbase
```

```
generate domainmodel "http://www.eclipse.org/xtext/example/Domainmodel"

Domainmodel:
  (elements += AbstractElement)*
;

PackageDeclaration:
  'package' name = QualifiedName '{'
    (elements += AbstractElement)*
  '}'
;

AbstractElement:
  PackageDeclaration | Entity | Import
;

Import:
  'import' importedNamespace = QualifiedNameWithWildcard
;

QualifiedNameWithWildcard:
  QualifiedName '.*'?
;

Entity:
  'entity' name = ID
           ('extends' superType = JvmTypeReference)?
  '{'
    (features += Feature)*
  '}'
;

Feature:
  name = ID ':' type = JvmTypeReference
;
```

As we changed the grammar, we have to regenerate the language now.

Being able to parse a Java type reference is already nice, but we also have to write them back to their string representation when we generate Java code. Unfortunately, a generic type reference with fully qualified class names can become a bit bulky. Therefore, the ImportManager shortens fully qualified names, keeps track of imported namespaces, avoids name collisions, and helps to serialize JvmTypeReferences by means of the TypeReferenceSerializer. This utility encapsulates how type references may be serialized depending on the concrete context in the output.

The following snippet shows our code generator using an ImportManager in conjunction with as TypeReferenceSerializer. We create a new instance and pass it through the generation functions, collecting types on the way. As the import section in a Java

file precedes the class body, we create the body into a String variable and assemble the whole file's content in a second step.

```
class DomainmodelGenerator implements IGenerator {

  @Inject extension IQualifiedNameProvider
  @Inject extension TypeReferenceSerializer

  override void doGenerate(Resource resource, IFileSystemAccess fsa) {
    for(e: resource.allContents.toIterable.filter(typeof(Entity))) {
      fsa.generateFile(
        e.fullyQualifiedName.toString("/") + ".java",
        e.compile)
    }
  }

  def compile(Entity it) '''
    «val importManager = new ImportManager(true)»
    «val body = body(importManager)»
    «IF eContainer != null»
      package «eContainer.fullyQualifiedName»;
    «ENDIF»

    «FOR i:importManager.imports»
      import «i»;
    «ENDFOR»

    «body»
  '''

  def body(Entity it, ImportManager importManager) '''
    public class «name» «IF superType != null»
      extends «superType.shortName(importManager)» «ENDIF»{
      «FOR f : features»
        «f.compile(importManager)»
      «ENDFOR»
    }
  '''

  def compile(Feature it, ImportManager importManager) '''
    private «type.shortName(importManager)» «name»;

    public «type.shortName(importManager)»
      get«name.toFirstUpper»() {
      return «name»;
    }

    public void set«name.toFirstUpper»(
```

```
      «type.shortName(importManager)» «name») {
      this.«name» = «name»;
    }
  '''

  def shortName(JvmTypeReference ref,
        ImportManager importManager) {
    val result = new StringBuilderBasedAppendable(importManager)
    ref.serialize(ref.eContainer, result);
    result.toString
  }
}
```

10.4. Inferring a JVM Model

In many cases, you will want your DSLs concepts to be usable as Java elements. E.g. an *Entity* will become a Java class and should be usable as such. In the domain model example, you can write

```
entity Employee extends Person {
  boss: Person
...

entity Person {
  friends: List<Person>
...
```

i.e. use entities instead of Java types or even mix Java types as List with entities such as *Person*. One way to achieve this is to let your concepts inherit from a corresponding JVM type, e.g. let *Entity* inherit from JvmGenericType. But this would result in a lot of accidentally inherited properties in your domain model. In Xbase there is an alternative: You can simply define how to derive a JVM model from your model. This *inferred JVM model* is the representation of your concepts in the typesystem of Xbase.

The main component for the inferred JVM model is the IJvmModelInferrer. It has a single method that takes the root model element as an argument and produces a number of JvmDeclaredTypes. As Xbase cannot guess how you would like to map your concepts to JVM elements, you have to implement this component yourself. This usually boils down to using an injected JvmTypesBuilder to create a hierarchy of JVM elements. The builder helps to initialize the produced types with sensible default and encapsulates the logic that associates the source elements with the derived JVM concepts. As this kind of transformation can be elegantly implemented using polymorphic dispatch functions and extension methods, it is a good choice to write the IJvmModelInferrer in Xtend. It

becomes even simpler if you inherit from the AbstractModelInferrer which traverses the input model and dispatches to its contents until you decide which elements to handle.

The inference runs in two phases: In the first phase all the types are created with empty bodies. This way you make sure all types exist and are referrable when you create their members in the second phase. Use acceptor.accept(JvmDeclaredType) for the first phase and provide the initialization code for the second phase. You have to pass a lambda expression (§10.6.3) to the method initializeLater() on the return type of the acceptor.accept() method.

For our domain model example, we implement a polymorphic dispatch function *infer* for *Entities* to transform them into a JvmGenericType in the first phase. In the second phase, we add a JvmField and corresponding accessors for each *Property*. The final *DomainmodelJvmModelInferrer* looks like this:

```
class DomainmodelJvmModelInferrer extends AbstractModelInferrer {

  @Inject extension JvmTypesBuilder

  @Inject extension IQualifiedNameProvider

  def dispatch void infer(Entity element,
              IJvmDeclaredTypeAcceptor acceptor,
              boolean isPrelinkingPhase) {

    acceptor.accept(element.toClass(element.fullyQualifiedName)).initializeLater [
      documentation = element.documentation
      for (feature : element.features) {
        members += feature.toField(feature.name, feature.type)
        members += feature.toSetter(feature.name, feature.type)
        members += feature.toGetter(feature.name, feature.type)
      }
    ]
  }
}
```

10.4.1. Linking and Indexing

As Java elements and your concepts are now represented as JVM model elements, other models can now transparently link to Java or your DSL. In other words, you can use a mapped element of your DSL in the same places as the corresponding Java type.

The Xbase framework will automatically switch between the JVM element or the DSL element when needed, e.g. when following hyperlinks. The component allowing to navigate between the source model and the JVM model is called IJvmModelAssociations, the read-only antagonist of the IJvmModelAssociator that is used by the JvmTypesBuilder.

By default, the inferred model is indexed (§8.6.1), so it can be cross referenced from other models.

10.5. Using Xbase Expressions

Xbase is an expression language that can be embedded into Xtext languages. Its syntax is close to Java, but it additionally offers type inference, lambda expressions, a powerful switch expression and a lot more. For details on this expression langugae, please consult the reference documentation (§10.6) and the Xbase tutorial *(File > New > Example > Xtext Examples > Xbase Tutorial)*.

Xbase ships with an interpreter and a compiler that produces Java code. Thus, it is easy to add behavior to your DSLs and make them executable. As Xbase integrates tightly with Java, there is usually no additional code needed to run your DSL as part of a Java application.

10.5.1. Making Your Grammar Refer To Xbase

If you want to refer to EClassifiers from the Xbase model, you need to import Xbase first:

```
import "http://www.eclipse.org/xtext/xbase/Xbase" as xbase
```

Now identify the location in your grammar where you want references to Java types and Xbase expressions to appear and call the appropriate rules of the super grammar. Adding Xbase expression to the domainmodel example leads to the additional concept *Operation.*: An *Operation*'s parameters are *FullJvmFormalParameters*. The production rule for *FullJvmFormalParameters* expects both the name and the type here. That is reasonable since the type of parameters should not be inferred. The opertion's return type is a *JvmTypeReference* and its *body* is an *XBlockExpression*. The final parser rule reads as:

```
Operation:
  'op' name=ValidID '('
  (params+=FullJvmFormalParameter (',' params+=FullJvmFormalParameter)*)? ')'
  ':' type=JvmTypeReference
    body=XBlockExpression;
```

If you are unsure which entry point to choose for your expressions, consider the *XBlock-Expression*.

To integrate *Operations* in our models, we have to call this rule. We copy the previous *Feature* to a new rule *Property* and let *Feature* become the supertype of *Property* and *Operation*:

```
Feature:
```

```
  Property | Operation
;

Property:
  name = ID ':' type = JvmTypeReference
;
```

Note: You will have to adapt the IJvmModelInferrer to these changes, i.e. rename *Feature* to *Property* and create a JvmOperation for each *Operation*. We leave that as an exercise :-)

If you are done with that, everything will work out of the box. Since each expression is now logically contained in an operation, all the scoping rules and visibility constraints are implied from that context. The framework will take care that the operation's parameters are visible inside the operation's body and that the declared return types are validated against the actual expression types.

There is yet another aspect of the JVM model that can be explored. Since all the coarse grained concepts such as types and operations were already derived from the model, a generator can be used to serialize that information to Java code. There is no need to write a code generator on top of that. The JvmModelGenerator knows how to generate operation bodies properly.

10.5.2. Using the Xbase Interpreter

Sometimes it is more convenient to interpret a model that uses Xbase than to generate code from it. Xbase ships with the XbaseInterpreter which makes this rather easy.

An interpreter is essentially an external visitor, that recursively processes a model based on the model element's types. By now you should be aware that polymorphic dispatching is exactly the technology needed here. In the XbaseInterpreter, the dispatch method is called *_evaluate<SomeDescription>* and takes three parameters, e.g.

```
protected Object _evaluateBlockExpression(XBlockExpression literal,
                  IEvaluationContext context,
                  CancelIndicator indicator)
```

The IEvaluationContext keeps the state of the running application, i.e. the local variables and their values. Additionally, it can be *fork*ed, thus allowing to shadow the elements of the original context. Here is an example code snippet how to call the XbaseInterpreter:

```
@Inject private XbaseInterpreter xbaseInterpreter;

@Inject private Provider<IEvaluationContext> contextProvider;
```

```
...
public Object evaluate(XExpression expression, Object thisElement) {
  IEvaluationContext evaluationContext = contextProvider.get();
  // provide initial context and implicit variables
  evaluationContext.newValue(XbaseScopeProvider.THIS, thisElement);

  IEvaluationResult result = xbaseInterpreter.evaluate(expression,
    evaluationContext, CancelIndicator.NullImpl);
  if (result.getException() != null) {
    // handle exception
  }
  return result.getResult();
}
```

10.6. Xbase Language Reference

This document describes the expression language library Xbase. Xbase is a partial programming language implemented in Xtext and is meant to be embedded and extended within other programming languages and domain-specific languages (DSL) written in Xtext. Xtext is a highly extendable language development framework covering all aspects of language infrastructure such as parsers, linkers, compilers, interpreters and even full-blown IDE support based on Eclipse.

Developing DSLs has become incredibly easy with Xtext. Structural languages which introduce new coarse-grained concepts, such as services, entities, value objects or state-machines can be developed in minutes. However, software systems do not consist of structures solely. At some point a system needs to have some behavior, which is usually specified using so called *expressions*. Expressions are the heart of every programming language and are not easy to get right. On the other hand, expressions are well understood and many programming languages share a common set and understanding of expressions.

That is why most people do not add support for expressions in their DSL but try to solve this differently. The most often used workaround is to define only the structural information in the DSL and add behavior by modifying or extending the generated code. It is not only unpleasant to write, read and maintain information which closely belongs together in two different places, abstraction levels and languages. Also, modifying the generated source code comes with a lot of additional problems. This has long time been the preferred solution since adding support for expressions (and a corresponding execution environment) for your language has been hard - even with Xtext.

Xbase serves as a language library providing a common expression language bound to the Java platform (i.e. Java Virtual Machine). It consists of an Xtext grammar, as well as reusable and adaptable implementations for the different aspects of a language infrastructure such as an AST structure, a compiler, an interpreter, a linker, and a static

analyzer. In addition it comes with implementations to integrate the expression language within an Xtext-based Eclipse IDE. Default implementations for aspects like content assistance, syntax coloring, hovering, folding and navigation can be easily integrated and reused within any Xtext based language.

Conceptually and syntactically, Xbase is very close to Java statements and expressions, but with a few differences:

- No checked exceptions
- Everything is an expression, there are no statements
- Lambda expressions
- Type inference
- Properties
- Simple operator overloading
- Powerful switch expressions

10.6.1. Lexical Syntax

Xbase comes with a small set of terminal rules, which can be overridden and hence changed by users. However the default implementation is carefully chosen and it is recommended to stick with the lexical syntax described in the following.

Identifiers

Identifiers are used to name all constructs, such as types, methods and variables. Xbase uses the default identifier-syntax from Xtext - compared to Java, they are slightly simplified to match the common cases while having less ambiguities. They start with a letter *a-z*, *A-Z* or an underscore followed by more of these characters or any digit *0-9*.

Escaped Identifiers

Identifiers must not have the same spelling as any reserved keyword. However, this limitation can be avoided by escaping identifiers with the prefix ^. Escaped identifiers are used in cases when there is a conflict with a reserved keyword. Imagine you have introduced a keyword service in your language but want to call a Java property *service*. In such cases you can use the escaped identifier ^service to reference the Java property.

Syntax

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

Examples

- Foo
- Foo42
- FOO
- _42
- _foo
- ^extends

Comments

Xbase comes with two different kinds of comments: Single-line comments and multi-line comments. The syntax is the same as the one known from Java (see § 3.7 Comments).

White Space

The white space characters ' ', '\t', '\n', and '\r are allowed to occur anywhere between the other syntactic elements.

Reserved Keywords

The following list of words are reserved keywords, thus reducing the set of possible identifiers:

1. as
2. case
3. catch
4. default
5. do
6. else
7. extends
8. false
9. finally
10. for

11. if
12. instanceof
13. new
14. null
15. return
16. super
17. switch
18. throw
19. true
20. try
21. typeof
22. val
23. var
24. while

In case some of the keywords have to be used as identifiers, the escape character for identifiers (§10.6.1) comes in handy.

10.6.2. Types

Basically all kinds of JVM types are available and referable.

Arrays

Arrays cannot be declared explicitly, but they can be passed around and they are (if needed) transparently converted to a List of the component type.

In other words, the return type of a Java method that returns an array of ints (int[]) can be directly assigned to a variable of type List<Integer>. Due to type inference this conversion happens implicitly. The conversion is bi-directional: Any method that takes an array as argument can be invoked with a List instead.

Simple Type References

A simple type reference only consists of a *qualified name*. A qualified name is a name made up of identifiers which are separated by a dot (like in Java).

There is no parser rule for a simple type reference, as it is expressed as a parameterized type references without parameters.

Examples

- java.lang.String
- String

Function Types

Xbase introduces *lambda expressions*, and therefore an additional function type signature. On the JVM-Level a lambda expression (or more generally any function object) is just an instance of one of the types in Functions, depending on the number of arguments. However, as lambda expressions are a very important language feature, a special sugared syntax for function types has been introduced. So instead of writing Function1<String,Boolean> one can write (String)=>Boolean.

For more information on lambda expressions see section 10.6.3.

Examples

- =>Boolean // predicate without parameters
- ()=>String // provider of string
- (String)=>Boolean // One argument predicate
- (Mutable)=>void // A procedure doing side effects only
- (List<String>, Integer)=>String

Parameterized Type References

The general syntax for type references allows to take any number of type arguments. The semantics as well as the syntax is almost the same as in Java, so please refer to the third edition of the Java Language Specification.

The only difference is that in Xbase a type reference can also be a function type. In the following the full syntax of type references is shown, including function types and type arguments.

Examples

- String
- java.lang.String
- List<?>
- List<? extends Comparable<? extends FooBar>
- List<? super MyLowerBound>
- List<? extends =>Boolean>

Primitives

Xbase supports all Java primitives. The conformance rules (e.g. boxing and unboxing) are also exactly like defined in the Java Language Specification.

Conformance and Conversion

Type conformance rules are used in order to find out whether some expression can be used in a certain situation. For instance when assigning a value to a variable, the type of the right hand expression needs to conform to the type of the variable.

As Xbase implements the type system of Java it also fully supports the conformance rules defined in the Java Language Specification.

Some types in Xbase can be used synonymously even if they do not conform to each other in Java. An example for this are arrays and lists or function types with compatible function parameters. Objects of these types are implicitly converted by Xbase on demand.

Common Super Type

Because of type inference Xbase sometimes needs to compute the most common super type of a given set of types.

For a set *[T1,T2,...Tn]* of types the common super type is computed by using the linear type inheritance sequence of *T1* and is iterated until one type conforms to each *T2,..,Tn*. The linear type inheritance sequence of *T1* is computed by ordering all types which are part if the type hierarchy of *T1* by their specificity. A type *T1* is considered more specific than *T2* if *T1* is a subtype of *T2*. Any types with equal specificity will be sorted by the maximal distance to the originating subtype. *CharSequence* has distance 2 to *StringBuilder* because the supertype *AbstractStringBuilder* implements the interface, too. Even if *StringBuilder* implements *CharSequence* directly, the interface gets distance 2 in the ordering because it is not the most general class in the type hierarchy that implements the interface. If the distances for two classes are the same in the hierarchy, their qualified name is used as the compare-key to ensure deterministic results.

10.6.3. Expressions

Expressions are the main language constructs which are used to express behavior and compute values. The concept of statements is not supported, but instead powerful expressions are used to handle situations in which the imperative nature of statements would be helpful. An expression always results in a value (it might be the value null though). In addition, every resolved expressions is of a static type.

Literals

A literal denotes a fixed unchangeable value. Literals for strings, numbers, booleans, null and Java types are supported.

String Literals

String literals can either use 'single quotes' or "double quotes" as their terminals. When using double quotes all literals allowed by Java string literals are supported. In addition new line characters are allowed, i.e. in Xbase string literals can span multiple lines. When using single quotes the only difference is that single quotes within the literal have to be escaped and double quotes do not.

See § 3.10.5 String Literals

In contrast to Java, equal string literals within the same class do not neccessarily refer to the same instance at runtime.

Examples

- 'Foo Bar Baz'
- "Foo Bar Baz"
- ```
  "the quick brown fox  jumps over the lazy dog."
  ```
- 'Escapes : \' '
- "Escapes : \" "

Number Literals

Xbase supports roughly the same number literals as Java with a few notable differences. As in Java 7, you can separate digits using _ for better readability of large numbers.

An integer literal represents an int, a long (suffix L) or even a BigInteger (suffix BI). There are no octal number literals.

```
42
1_234_567_890
0xbeef    // hexadecimal
077       // decimal 77 (*NOT* octal)
42L
0xbeef#L // hexadecimal, mind the '#'
0xbeef_beef_beef_beef_beef#BI // BigInteger
```

A floating-point literal creates a double (suffix D or omitted), a float (suffix F) or a BigDecimal (suffix BD). If you use a . sign you have to specify both, the integer and the fractional part of the mantissa. There are only decimal floating-point literals.
```

```
42d     // double
0.42e2  // implicit double
0.42e2f // float
4.2f    // float
0.123_456_789_123_456_789_123_456_789e2000bd // BigDecimal
```

Boolean Literals

There are two boolean literals, true and false which correspond to their Java counterpart of type *boolean*.

- true
- false

Null Literal

The null literal is, as in Java, null. It is compatible to any reference type and therefore always of the null type.

- null

Type Literals

Type literals are specified using the keyword typeof :

- typeof(String) which yields String.class

Type Casts

Type cast behave the same as in Java, but have a slightly more readable syntax. Type casts bind stronger than any other operator but weaker than feature calls.

The conformance rules for casts are defined in the Java Language Specification.

Examples

- my.foo as MyType
- (1 + 3 * 5 * (- 23)) as BigInteger

Infix Operators / Operator Overloading

There are a couple of common predefined infix operators. In contrast to Java, the operators are not limited to operations on certain types. Instead an operator-to-method mapping allows users to redefine the operators for any type just by implementing the corresponding method signature. The following defines the operators and the corresponding Java method signatures / expressions.

e1 += e2	e1.operator_add(e2)
e1 || e2	e1.operator_or(e2)
e1 && e2	e1.operator_and(e2)
e1 == e2	e1.operator_equals(e2)
e1 != e2	e1.operator_notEquals(e2)
e1 < e2	e1.operator_lessThan(e2)
e1 > e2	e1.operator_greaterThan(e2)
e1 <= e2	e1.operator_lessEqualsThan(e2)
e1 >= e2	e1.operator_greaterEqualsThan(e2)
e1 -> e2	e1.operator_mappedTo(e2)
e1 .. e2	e1.operator_upTo(e2)
e1 => e2	e1.operator_doubleArrow(e2)
e1 << e2	e1.operator_doubleLessThan(e2)
e1 >> e2	e1.operator_doubleGreaterThan(e2)
e1 <<< e2	e1.operator_tripleLessThan(e2)
e1 >>> e2	e1.operator_tripleGreaterThan(e2)
e1 <> e2	e1.operator_diamond(e2)
e1 ?: e2	e1.operator_elvis(e2)
e1 <=> e2	e1.operator_spaceship(e2)
e1 + e2	e1.operator_plus(e2)
e1 - e2	e1.operator_minus(e2)
e1 * e2	e1.operator_multiply(e2)
e1 / e2	e1.operator_divide(e2)
e1 % e2	e1.operator_modulo(e2)
e1 ** e2	e1.operator_power(e2)
! e1	e1.operator_not()
- e1	e1.operator_minus()

The table above also defines the operator precedence in ascending order. The blank lines separate precedence levels. The assignment operator += is right-to-left associative

in the same way as the plain assignment operator = is. Consequently, a = b = c is executed as a = (b = c). All other operators are left-to-right associative. Parentheses can be used to adjust the default precedence and associativity.

Short-Circuit Boolean Operators

If the operators || and && are used in a context where the left hand operand is of type boolean, the operation is evaluated in short circuit mode, which means that the right hand operand is not evaluated at all in the following cases:

1. in the case of || the operand on the right hand side is not evaluated if the left operand evaluates to true.
2. in the case of && the operand on the right hand side is not evaluated if the left operand evaluates to false.

Examples

- my.foo = 23
- myList += 23
- x > 23 && y < 23
- x && y || z
- 1 + 3 * 5 * (- 23)
- !(x)
- my.foo = 23
- my.foo = 23

With Operator

The *with* operator => executes the lambda expression (§10.6.3) with a single parameter on the right-hand side with a given argument on its left-hand side. The result is the left operand after applying the lambda expression. In combination with the implicit parameter (§10.6.3) it this allows very convenient initialization of newly created objects. Example:

```
val person = new Person => [
  firstName = 'John'
  lastName = 'Coltrane'
]
// equivalent to
val person = new Person
```

```
person.firstName = 'John'
person.lastName = 'Coltrane'
```

Assignments

Local variables (§10.6.3) can be reassigned using the = operator. Also properties can be set using that operator: Given the expression

```
myObj.myProperty = "foo"
```

The compiler first looks for an accessible Java Field called myProperty on the declared or inferred type of myObj. If such a field can be found, the expressions translates to the following Java code:

```
myObj.myProperty = "foo";
```

Remember, in Xbase everything is an expression and has to return something. In the case of simple assignments the return value is the value returned from the corresponding Java expression, which is the assigned value.

If there is no accessible field on the left operand's type, a method called setMyProperty(OneArg) (JavaBeans setter method) is looked up. It has to take one argument of the type (or a super type) of the right hand operand. The return value of the assignment will be the return value of the setter method (which is usually of type void and therefore the value null). As a result the compiler translates to :

```
myObj.setMyProperty("foo")
```

Feature Calls

A feature call is used to access members of objects, such as fields and methods, but it can also refer to local variables and parameters, which are made available by the current expression's scope.

Property Access

Feature calls are directly translated to their Java equivalent with the exception, that access to properties follows similar rules as described in section 10.6.3. That is, for the expression

```
myObj.myProperty
```

the compiler first looks for an accessible field myProperty in the type of myObj. If no such field exists it tries to find a method called myProperty() before it looks for the getter methods getMyProperty(). If none of these members can be found, the expression is unbound and a compiliation error is indicated.

Null-Safe Feature Call

Checking for null references can make code very unreadable. In many situations it is ok for an expression to return null if a receiver was null. Xbase supports the safe navigation operator ?. to make such code more readable.

Instead of writing

```
if ( myRef != null ) myRef.doStuff()
```

one can write

```
myRef?.doStuff()
```

Static Feature Calls

To access static members of a type, use ::, e.g.

```
java::util::Collections::singleton("Lonesome Cowboy")
System::err.println("An error occurred")
```

Implicit variables 'this' and 'it'

If the current scope contains a variable named this or it, the compiler will make all its members available implicitly. That is if one of

```
it.myProperty
this.myProperty
```

is a valid expression

```
myProperty
```

is valid as well. It resolves to the same feature as long as there is no local variable myProperty declared, which would have higher precedence.

As this is bound to the surrounding object in Java, it can be used in finer-grained constructs such as function parameters. That is why it.myProperty has higher precedence than this.myProperty. it is also the default parameter name in lambda expressions (§10.6.3).

Constructor Call

Construction of objects is done by invoking Java constructors. The syntax is exactly as in Java.

Examples

- ```
  new String()
  ```
- ```
  new java.util.ArrayList<java.math.BigDecimal>()
  ```

Lambda Expressions

A lambda expression is a literal that defines an anonymous function. Xbase' lambda expressions are allowed to access variables of the declarator. Any final variables and parameters visible at construction time can be referred to in the lambda expression's body. These expressions are also known as closures.

Lambda expressions are surrounded by square brackets (\[]):

```
myList.findFirst([ e | e.name==null ])
```

When a function object is expected to be the last parameter of a feature call, you may declare the lambda expression after the parentheses:

```
myList.findFirst() [ e | e.name==null ]
```

Since in Xbase parentheses are optional for method calls, the same can be written as:

```
myList.findFirst[ e | e.name==null ]
```

This example can be further simplified since the lambda's parameter is available as the implicit variable it, if the parameter is not declared explicitly:

```
myList.findFirst[ it.name==null ]
```

Since it is implicit, this is the same as:

```
myList.findFirst[ name==null ]
```

Another usecase for lambda expressions is to store function objects in variables:

```
val func = [ String s | s.length>3 ]
```

Typing

Lambda expressions produce function objects. The type is a function type (§10.6.2), parameterized with the types of the lambda's parameters as well as the return type. The return type is never specified explicitly but is always inferred from the expression. The parameter types can be inferred if the lambda expression is used in a context where this is possible.

For instance, given the following Java method signature:

```
public T <T>getFirst(List<T> list, Function0<T,Boolean> predicate)
```

the type of the parameter can be inferred. Which allows users to write:

```
newArrayList( "Foo", "Bar" ).findFirst[ e | e == "Bar" ]
```

instead of

```
newArrayList( "Foo", "Bar" ).findFirst[ String e | e == "Bar" ]
```

Function Mapping

An Xbase lambda expression is a Java object of one of the *Function* interfaces that are part of the runtime library of Xbase. There is an interface for each number of parameters (up to six parameters). The names of the interfaces are

- Function0<ReturnType> for zero parameters,
- Function1<Param1Type, ReturnType> for one parameters,
- Function2<Param1Type, Param2Type, ReturnType> for two parameters,
- ...
- Function6<Param1Type, Param2Type, Param3Type, Param4Type, Param5Type, Param6Type, ReturnType> for six parameters,

or

- Procedure0 for zero parameters,
- Procedure1<Param1Type> for one parameters,
- Procedure2<Param1Type, Param2Type> for two parameters,
- ...
- Procedure6<Param1Type, Param2Type, Param3Type, Param4Type, Param5Type, Param6Type> for six parameters,

if the return type is void.

In order to allow seamless integration with existing Java libraries such as the JDK or Google Guava (formerly known as Google Collect) lambda expressions are auto coerced to expected types if those types declare only one abstract method (methods from java.lang.Object don't count).

As a result given the method Collections.sort(List<T>, Comparator<? super T>) is available as an extension method, it can be invoked like this

```
newArrayList( 'aaa', 'bb', 'c' ).sort [
  e1, e2 | if ( e1.length > e2.length ) {
       -1
      } else if ( e1.length < e2.length ) {
       1
      } else {
       0
      }
]
```

Implicit Parameter it

If a lambda expression has a single parameter whose type can be inferred, the declaration of the parameter can be omitted. Use it to refer to the parameter inside the lambda expression's body.

```
val (String s)=>String function = [ toUpperCase ]
  // equivalent to [it | it.toUpperCase]
```

Examples

- [| "foo"] // lambda expression without parameters
- [String s | s.toUpperCase()] // explicit argument type
- [a, b, c | a+b+c] // inferred argument types

If Expression

An if expression is used to choose two different values based on a predicate. While it has the syntax of Java's if statement it behaves like Java's ternary operator (predicate ? thenPart : elsePart), i.e. it is an expression that returns a value. Consequently, you can use if expressions deeply nested within other expressions.

An expression if (p) e1 else e2 results in either the value e1 or e2 depending on whether the predicate p evaluates to true or false. The else part is optional which is a shorthand for else null. That means

```
if (foo) x
```

is the a short hand for

```
if (foo) x else null
```

Typing

The type of an if expression is calculated by the return types T1 and T2 of the two expression e1 and e2. It uses the rules defined in section 10.6.2.

Examples

- if (isFoo) this else that
- if (isFoo) { this } else if (thatFoo) { that } else { other }
- if (isFoo) this

Switch Expression

The switch expression is a bit different from Java's. First, there is no fall through which means only one case is evaluated at most. Second, the use of switch is not limited to certain values but can be used for any object reference instead.

For a switch expression

```
switch e {
  case e1 : er1
  case e2 : er2
  ...
  case en : ern
  default : er
}
```

the main expression e is evaluated first and then each case sequentially. If the switch expression contains a variable declaration using the syntax known from for loops (§10.6.3), the value is bound to the given name. Expressions of type Boolean or boolean are not allowed in a switch expression.

The guard of each case clause is evaluated until the switch value equals the result of the case's guard expression or if the case's guard expression evaluates to true. Then the right hand expression of the case evaluated and the result is returned.

If none of the guards matches the default expression is evaluated and returned. If no default expression is specified the expression evaluates to null.

Example:

```
switch myString {
  case myString.length>5 : 'a long string.'
  case 'foo' : "It's a foo."
  default : "It's a short non-foo string."
}
```

Type guards

In addition to the case guards one can add a so called *Type Guard* which is syntactically just a type reference (§10.6.2) preceding the than optional case keyword. The compiler will use that type for the switch expression in subsequent expressions. Example:

```
var Object x = ...;
switch x {
  String case x.length()>0 : x.length()
  List<?> : x.size()
  default : -1
}
```

Only if the switch value passes a type guard, i.e. an instanceof operation returns true, the case's guard expression is executed using the same semantics as explained above. If the switch expression contains an explicit declaration of a local variable or the expression references a local variable, the type guard works like an automated cast. All subsequent references to the switch value will be of the type specified in the type guard.

Typing

The return type of a switch expression is computed using the rules defined in the section on common supertypes (§10.6.2). The set of types from which the common super type is computed corresponds to the types of each case's result expression. If a switch expression's type is computed using the expected type from the context, it is sufficient to return the expected type if all case branches types conform to the expected type.

Examples

- ```
  switch foo {
    Entity : foo.superType.name
    Datatype : foo.name
    default : throw new IllegalStateException
  }
  ```
- ```
  switch x : foo.bar.complicated('hello',42) {
    case "hello42" : ...
    case x.length<2 : ...
    default : ....
  }
  ```

Variable Declarations

Variable declarations are only allowed within blocks (§10.6.3). They are visible in any subsequent expressions in the block. Generally, overriding or shadowing variables from outer scopes is not allowed. However, it can be used to overload the implicit variable (§10.6.3) it, in order to subsequently access an object's features in an unqualified manner.

A variable declaration starting with the keyword val denotes an unchangeable value, which is essentially a final variable. In rare cases, one needs to update the value of a reference. In such situations the variable needs to be declared with the keyword var, which stands for variable.

A typical example for using var is a counter in a loop.

```
{
  val max = 100
  var i = 0
  while (i > max) {
    println("Hi there!")
    i = i +1
  }
}
```

Variables declared outside a lambda expression using the var keyword are not accessible from within a the lambda expression.

Typing

The return type of a variable declaration expression is always void. The type of the variable itself can either be explicitly declared or be inferred from the right hand side expression. Here is an example for an explicitly declared type:

```
var List<String> msg = new ArrayList<String>();
```

In such cases, the right hand expression's type must conform (§10.6.2) to the type on the left hand side.

Alternatively the type can be left out and will be inferred from the initialization expression:

```
var msg = new ArrayList<String> // -> type ArrayList<String>
```

Blocks

The block expression allows to have imperative code sequences. It consists of a sequence of expressions, and returns the value of the last expression. The return type of a block is also the type of the last expression. Empty blocks return null. Variable declarations (§10.6.3) are only allowed within blocks and cannot be used as a block's last expression.

A block expression is surrounded by curly braces and contains at least one expression. It can optionally be terminated by a semicolon.

Examples

```
{
  doSideEffect("foo")
  result
}
```

```
{
  var x = greeting();
  if (x.equals("Hello ")) {
    x+"World!";
  } else {
    x;
  }
}
```

For Loop

The for loop for (T1 variable : iterableOfT1) expression is used to execute a certain expression for each element of an array of an instance of Iterable. The local variable is final, hence canot be updated.

The return type of a for loop is void. The type of the local variable can be left out. In that case it is inferred from the type of the array or Iterable returned by the iterable expression.

- ```
  for (String s : myStrings) {
    doSideEffect(s);
  }
  ```
  ```

- ```
  for (s : myStrings)
    doSideEffect(s)
  ```

While Loop

A while loop while (predicate) expression is used to execute a certain expression unless the predicate is evaluated to false. The return type of a while loop is void.

Examples

- ```
  while (true) {
    doSideEffect("foo");
  }
  ```
- ```
  while ( ( i = i + 1 ) < max )
    doSideEffect( "foo" )
  ```

Do-While Loop

A do-while loop do expression while (predicate) is used to execute a certain expression until the predicate is evaluated to false. The difference to the while loop (§10.6.3) is that the execution starts by executing the block once before evaluating the predicate for the first time. The return type of a do-while loop is void.

Examples

- ```
  do {
    doSideEffect("foo");
  } while (true)
  ```
- ```
  do doSideEffect("foo") while ((i=i+1)<max)
  ```

Return Expression

Although an explicit return is often not necessary, it is supported. In a lambda expression for instance a return expression is always implied if the expression itself is not of type void. Anyway you can make it explicit:
  ```

```
listOfStrings.map [ e |
  if (e==null)
    return "NULL"
  e.toUpperCase
]
```

Throwing Exceptions

Like in Java it is possible to throw Throwable. The syntax is exactly the same as in Java.

```
{
  ...
  if (myList.isEmpty)
    throw new IllegalArgumentException("the list must not be empty")
  ...
}
```

Try, Catch, Finally

The try-catch-finally expression is used to handle exceptional situations. You are not forced to declare checked exceptions. If you don't catch checked exceptions, they are rethrown in a way the compiler does not complain about a missing throws clause, using the sneaky-throw technique introduced by Lombok. The syntax again is the same known from Java.

```
try {
  throw new RuntimeException()
} catch (NullPointerException e) {
  // handle e
} finally {
  // do stuff
}
```

10.6.4. Extension Methods

Languages extending Xbase might want to contribute to the feature scope. Besides that, one can of course change the whole implementation as it seems fit. There is a special hook, which can be used to add so-called extension methods to existing types.

Xbase itself comes with a standard library of such extension methods adding support for various operators for the common types, such as String, List, etc.

These extension methods are declared in separate Java classes. There are various ways how extension methods can be added. In the simplest case the language designer predefines which extension methods are available. Language users cannot add additional library functions using this mechanism.

Another alternative is to have them looked up by a certain naming convention. Also for more general languages it is possible to let users add extension methods using imports or similar mechanisms. This approach can be seen in the language Xtend, where extension methods are lexically imported through static imports and/or dependency injection.

The precedence of extension methods is always lower than real member methods, i.e. you cannot override member features. Also the extension members are not invoked polymorphicly. If you have two extension methods on the scope (foo(Object) and foo(String)) the expression (foo as Object).foo would bind and invoke foo(Object).

Examples

- foo
- my.foo
- my.foo(x)
- oh.my.foo(bar)

Builder Syntax

If the last argument of a method call is a lambda expression, it can be appended to the method call. Thus,

```
foo(42) [ String s | s.toUpperCase ]
```

will call a Java method with the signature

```
void foo(int, Function1<String, String>)
```

Used in combination with the implicit parameter name in lambda expressions (§10.6.3) you can write extension libraries (§10.6.4) to create and initialize graphs of objects in a concise builder syntax like in Groovy. Consider you have a set of library methods

```
HtmlNode html(Function1<HtmlNode, Void> initializer)
HeadNode head(HtmlNode parent, Function1<HeadNode, Void> initializer)
...
```

that create DOM elements for HTML pages inside their respective parent elements. You can then create a DOM using the following Xbase code:

```
html([ html |
  head(html, [
    // initialize head
  ])
] )
```

Appending the lambda expression parameters and prepending the parent parameters using extension syntax yields

```
html() [ html |
  html.head() [
    // initialize head
  ]
]
```

Using implicit parameter it and skipping empty parentheses you can simplify this to

```
html [
  head [
    // initialize head
  ]
]
```

11. 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.

11.1. Examples

Let's start with a couple of examples to demonstrate some usage scenarios for MWE2. The first example 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 example uses the life-cycle methods of the IWorkflowComponent to print the execution time of the workflow.

11.1.1. 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 Mwe2Runner, the component SayHello must be nested in a root workflow:

```
module HelloWorld

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

The package org.eclipse.emf.mwe2.runtime.workflow of the class Workflow 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!

11.1.2. 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 supertype 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. Many tedious tasks that developers face in their daily work can be addressed by a chain of rather simple components. MWE2 can be used to automate these tasks with minimum effort.

11.1.3. A Stop-Watch

The last example demonstrates how to combine the MWE2 concepts to create a simple stop-watch that allows to measure the execution time of a set of components. The idea is to add the very same stop-watch 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 stop-watch may use the following pattern. The instance of the class StopWatch has to be added as the first component and the last component to a workflow. Every component 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
}
```

11.2. Language Reference

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

- 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 setABC(..) and addXYZ(..) 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) {..}
}
```

11.2.1. 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.

11.2.2. 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.

11.2.3. 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, 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 once in the workflow and reuse this fragment across certain other file locations.

11.2.4. 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.

11.2.5. 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 use forward references in a MWE2 file.

11.2.6. 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.

11.3. 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.

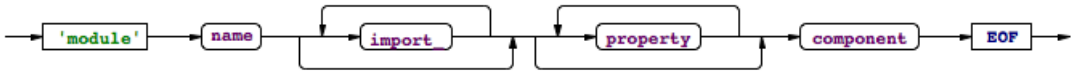
MWE2 is not sensitive to white space and allows to define line-comments and block comments everywhere. The syntax is the same as one is used to from the Java language:

```
// 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 a statement delimiter at any place.



11.3.1. Module

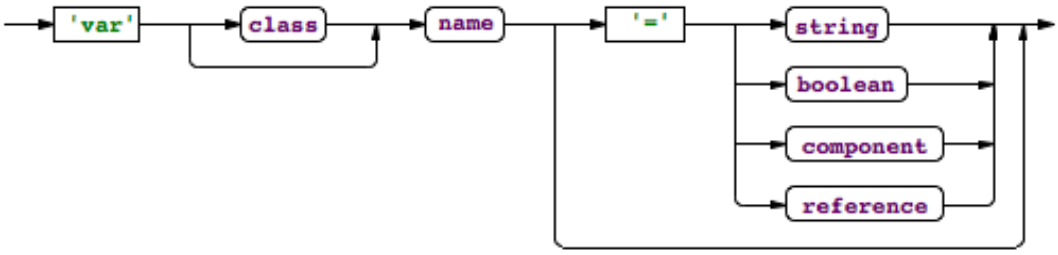
A module consists of four parts. The very first statement in a **.mwe2* file is the module declaration. The name of the module must follow the naming convention for Java classes. That MWE2 file's name must therefore be the same as the last segment of the module-name and it has to be placed in the appropriate 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 wildcard or they import a concrete class or module. MWE2 can handle relative imports in case one uses the wildcard notation:

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

11.3.2. Property

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

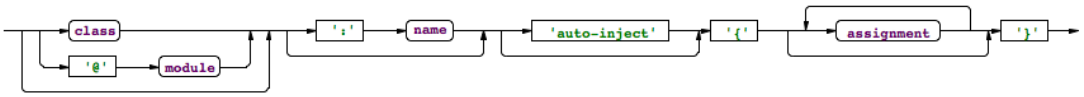


Each declared property is locally visible in the module. It furthermore defines an assignable feature of the module in case one refers to it from another module. Properties may either have a default value or they are considered to be *mandatory*. If the type of property is omitted it will be inferred from the default value. The default type of a property is String. That is, 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.

11.3.3. Component

The building block 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.

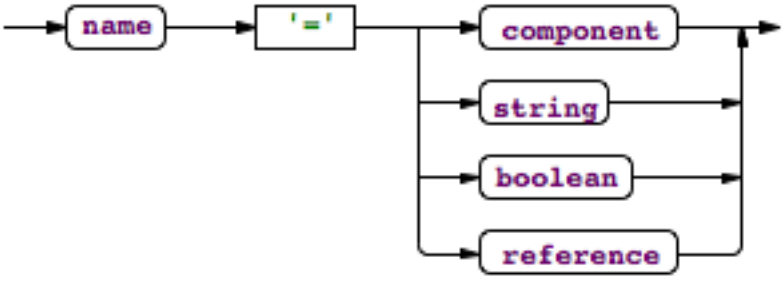


The type of the component can be derived in many cases except for the root component. That's why it's optional in the component literal. If no type is given, it will be inferred from the left side of the assignment. The assigned feature can either be a declared property of the module or a set- or add-method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



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

11.3.4. 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 descriptive 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.

11.3.5. Boolean Literals

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

11.3.6. 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.

12. 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.

12.1. Model, Ecore Model, and Ecore

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 tutorial (§2), the AST looks similar to this

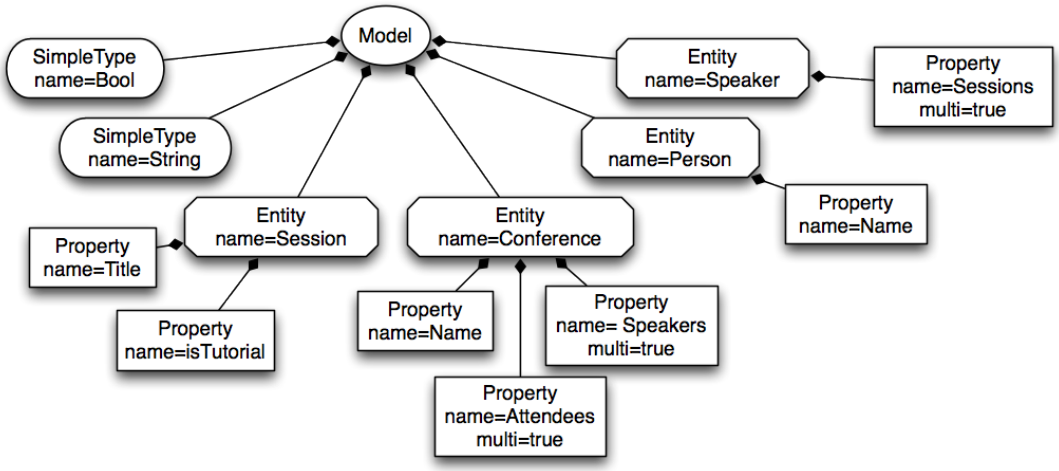


Figure 12.1.: Sample AST

The *AST* should contain the essence of your textual models. It abstracts 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 (§6.2.2) for details). The next diagram shows the meta model of our example:

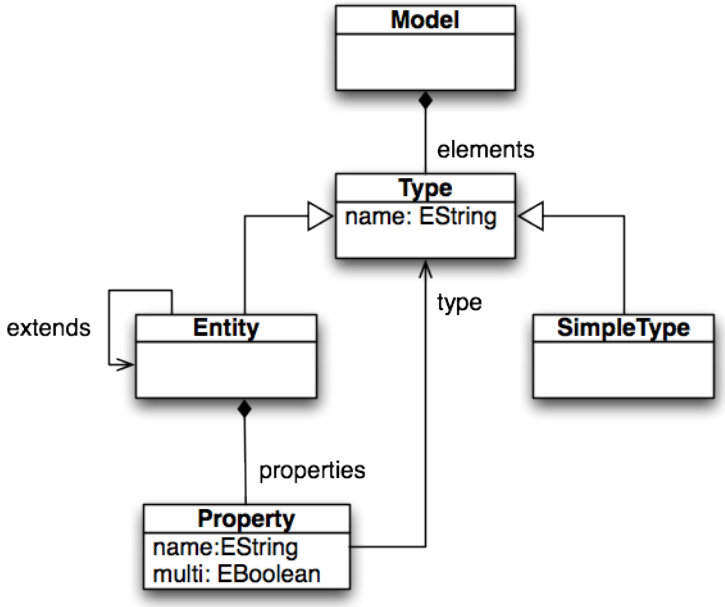


Figure 12.2.: Sample meta model

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 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 is *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 EAttributes 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:

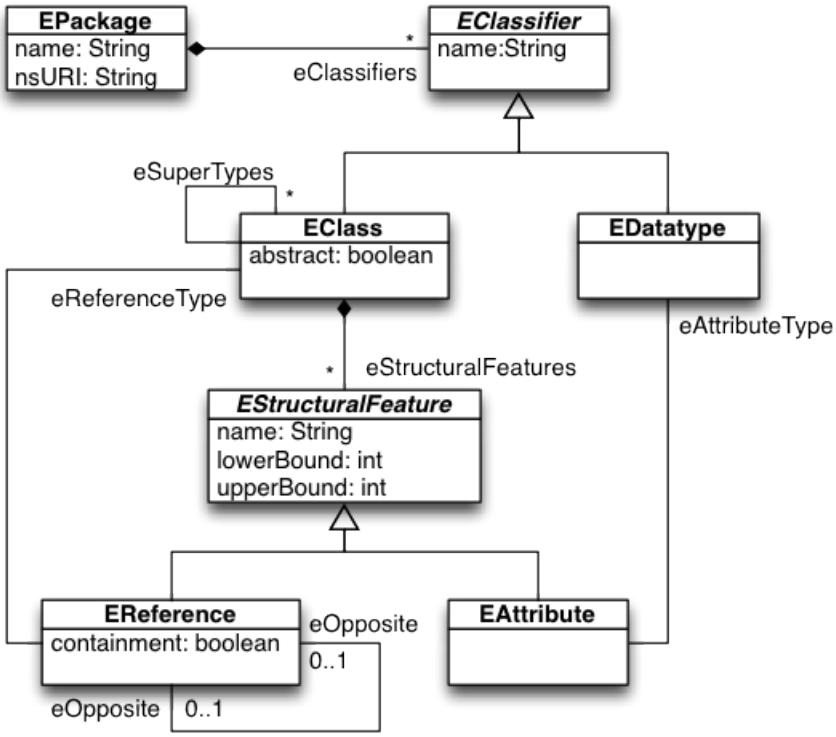


Figure 12.3.: Ecore concepts

12.2. 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

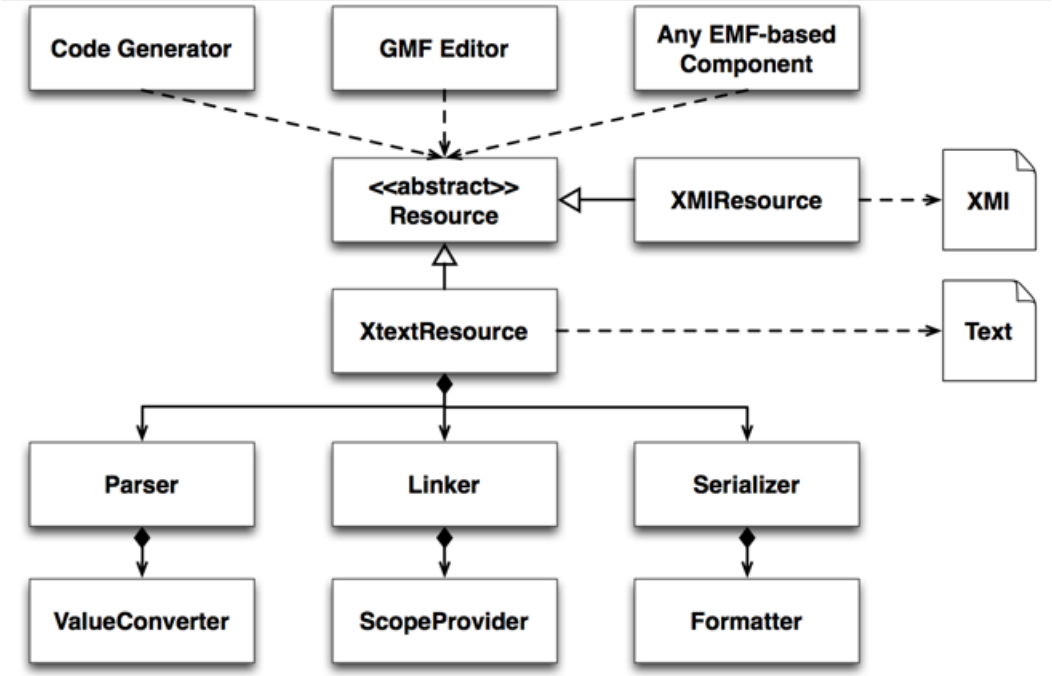
- A Java interface and a Java class for each EClassifier in your Ecore model. By default, all classes will implement the interface EObject, linking a lot of runtime functionality.
- A Java bean property for each EStructuralFeature (member variable, accessor methods)
- A package interface and class, holding 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.

12.3. XtextResource Implementation

Xtext provides an implementation of EMF's resource, the XtextResource. This does not only encapsulate the parser that converts text to an EMF model but also the serializer (§8.8) working the opposite direction. That way, an Xtext model just looks like any other Ecore-based model from the outside, making it amenable for the use by other EMF based tools. In fact, the Xpand templates in the generator plug-in created by the Xtext wizard do not make any assumption on the fact that the model is described in Xtext, and they would work fine with any model based on the same Ecore model of the language. So in the ideal case, you can switch the serialization format of your models to your self-defined DSL by just replacing the resource implementation used by your other modeling tools.

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

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



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

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

- Prefer optional rule calls (cardinality *?* or ***) to mandatory ones (cardinality *+* or default), such that missing references will not obstruct serialization.

- You should not use an Xtext-Editor on the same model instance as a self-synchronizing other editor, e.g. a canonical GMF editor (see subsection 12.4.1 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 (howto (§8.10)) 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 (howto (§8.6)) to make the names of all linkable elements in cross-references unique.
- Provide an IFormatter (howto (§8.9)) 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 Resource.Factory, 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.

12.4. Integration with GMF Editors

We do no longer maintain the GMF example code and have removed it from our installation. You can still access the last version of the source code form our source code repository.

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.

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.

12.4.1. 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 data types. 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 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* 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.

12.4.2. 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 attribute'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 6.2.4 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*. *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.

12.4.3. 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.

Part III.

Appendix

13. Migrating from Xtext 1.0.x to 2.0

Most of the tasks when migrating to Xtext 2.0 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 2.0. 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 1.0.x with a new Xtext project based on 2.0. Especially the new fragments in the workflow are a good indicator for useful new features.

13.1. 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!

13.2. Migrating Step By Step

Before you start the migration to Xtext 2.0, you should make sure that no old plug-ins are in your target platform.

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.

13.2.1. Update the Plug-in Dependencies and Import Statements

You should update the version constraints of the plug-in dependencies in your manifest files from version *1.0.x* to *2.0* if you specified any concrete versions. Also the constraint of *org.antlr.runtime* must be updated from *[3.0.0,3.0.2)* to *3.2.0*.

The next step is to fix the import statements in your classes to match the refactored naming scheme in Xtext. This fixes most of the problems in the manually written code.

13.2.2. Introduction of the Qualified Name

With Xtext 2.0 an object for dealing with qualified names has been introduced: QualifiedName . The qualified name is now split into segments which can be queried. The lower-case ver-

sion of the qualified name is cached so that the performance of case insensitive languages is improved. The signature of the methods used by the DefaultDeclarativeQualifiedNameProvider changed to QualifiedName qualifiedName(Object).

The IQualifiedNameConverter converts qualified names from/to their String representation. This is also where you specify the separator and wildcard strings. If you already know the segments of a qualified name, you can also create it using QualifiedName.create(String ...).

QualifiedName is the new type of the *name* properties in the IEObjectDescription. So if you have customized indexing, e.g. implemented your own IResourceDescription .Manager, you will have to create qualified names instead of strings. IEObjectDescriptions are also used in other places such as scoping (§8.6), linking (§8.5), serialization (§8.8), content assist (§9.2)...

Furthermore, the method IQualifiedNameProvider.getQualifiedName(EObject) has been renamed to getFullyQualifiedName(EObject).

13.2.3. Changes in the index and in find references

In Xtext 1.0.x the interfaces IResourceDescriptions, IResourceDescription and IContainer have several methods to query them for contained elements. In Xtext 2.0 there is a common interface ISelectable for this use case which is extended by the interfaces mentioned above. For further details have a look at the interface ISelectable.

The default indexing for Xtext resources as it is defined in DefaultResourceDescriptionManager has changed. Only cross-references pointing to elements outside the current resource are indexed. Furthermore, the DefaultResourceDescriptionManager can now be easier customized with an IDefaultResourceDescriptionStrategy.

For Ecore files only EPackages, EClassifiers and EStructuralFeatures are indexed, each with both, the *nsURI* and the *name* of the containing EPackage in their qualified name.

There is a new interface to find references to Xtext elements: IReferenceFinder. It allows to distinguish searches in the local Resource from global index searches. Local searches will yield *all* local cross references independent of the indexing strategy.

13.2.4. Rewritten Node Model

To reduce memory consumption, the node model has been redesigned in Xtext 2.0. We no longer use EMF, but a chained list of compressed nodes instead.

The package *org.eclipse.xtext.nodemodel* now contains the relevant interfaces to program against. The new interfaces follow the naming convention of other types in the framework. They are called INode, ICompositeNode and ILeafNode. That way, most of the migration will be done by prefixing the old names with an *I* and use the organize imports tool. Please make sure not to program against concrete or abstract classes.

If you used the node model a lot, you should have a closer look at the new APIs. The EObject API is no longer available on the nodes. Instead, you we offer a couple of Iterables for traversing the tree. Where appropriate, helper methods of the former *ParseTreeUtil* and *NodeUtil* have become members of the nodes, e.g. NodeUtil.getAllContents(AbstractNode) has become INode.getAsTreeIterable() The remaining methods have been converted and moved to the new NodeModelUtils.

13.2.5. New Outline

The outline view has been completely re-implemented. To use the new one remove the following fragments from your workflow

```
fragment = outline.TransformerFragment{}
fragment = outline.OutlineNodeAdapterFactoryFragment{}
```

and add

```
fragment = outline.OutlineTreeProviderFragment {}
```

After generating a new class named *MyDslOutlineTreeProvider* is generated. The API changed completely. For that reason you should take a closer look at the chapter on the outline (§9.5). The old classes named *MyDslTransformer* and *MyDslOutlineNodeAdapterFactory* have become obsolete and should be removed after having migrated your code to the new API.

13.2.6. AutoEditStrategy

In Xtext 1.0.x your AutoEditStrategy extends the class *DefaultAutoEditStrategy* which implements the interface IAutoEditStrategy. In Xtext 2.0 the DefaultAutoEditStrategyProvider should be extended instead. The only thing you have to do is to change the superclass from *DefaultAutoEditStrategy* to DefaultAutoEditStrategyProvider. The interface AbstractEditStrategyProvider.IEditStrategyAcceptor changed from accept(IAutoEditStrategy) to accept(IAutoEditStrategy, String). The last parameter represents the *contentType* of the document. Constants could be found in the IDocument and in the TerminalsTokenTypeToPartitionMapp
.

As a example the configure method could look like this one:

```
@Override
protected void configure(IEditStrategyAcceptor acceptor) {
  super.configure(acceptor);
  acceptor.accept(new YourAutoEditStrategy(),
        IDocument.DEFAULT_CONTENT_TYPE);
}
```

The last thing you have to do is to change the binding of the *IAutoEditStrategy* in the *MyDslUIModule* from

```
public Class<? extends IAutoEditStrategy> bindIAutoEditStrategy()
```

to

```
public Class<? extends AbstractEditStrategyProvider>
  bindAbstractEditStrategyProvider() { .. }
```

13.2.7. Other Noteworthy API Changes

The *src* folders are generated once, so existing code will not be overwritten but has to be updated manually.

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.

- In the interface IGlobalScopeProvider the method getScope(EObject,EReference) has been removed. Use getScope(Resource, EReference, Predicate<IEObjectDescription>) instead.
- The interface *IAntlrParser* has been removed. Use the IParser instead.
- The methods error(..) and warning(..) in the AbstractDeclarativeValidator used to accept integer constants representing the EStructuralFeature which caused the issues. These integer parameters were replaced by the feature itself, e.g. from error(String, Integer) to error(String, EStructuralFeature). Use the generated EPackage.Literals to access the EStructuralFeatures.
- The enum *DiagnosticSeverity* has been renamed to Severity.
- The class *TextLocation* has been replaced by the interface ITextRegion with an immutable implementation TextRegion.
- In Xtext 1.0.x the class EObjectAtOffsetHelper provided several static methods to resolve elements. In Xtext 2.0 these methods aren't static anymore. For that reason you could create an instance of this class or let Guice do the job for you:

  ```
  @Inject private EObjectAtOffsetHelper eObjectAtOffsetHelper;
  ```

- The method EObjectAtOffsetHelper.resolveElementAt(XtextResource, int, TextLocation) changed to resolveElementAt(XtextResource, int). You have to use the ILocationInFileProvider to compute the TextRegion.

- The ILocationInFileProvider now offers methods getSignificantTextRegion() and getFullTextRegion() to make the distinction between the name and the full region of an element. The old getLocation() method was removed.

If you experience further problems, please refer to the newsgroup.

13.3. Now go for then new features

After migrating, some of the new features in Xtext 2.0 will be automatically available. Others require further configuration. We recommend exploring

- Xbase,
- the new Xtend,
- rename refactorings
- the compare view,
- rich hovers,
- the Xtext syntax graph,
- support for syntactic predicates (§6.2.8),
- the generated debug grammar,
- terminal fragments (§6.2.3),
- document partitions and auto edit
- the redesigned outline view (§9.5),
- and the quick fixes for the Xtext grammar language (§9.3).

For an overview over the new features consult our New and Noteworthy online.

14. Migrating from Xtext 0.7.x to 1.0

For the sake of completeness, here is how you migrate from Xtext 0.7.x to Xtext 1.0, so you might do a migration to 2.0 in two steps. Nevertheless, there have been so many new features and changes that it probably makes more sense to just copy the grammar and start with a new Xtext 2.0 project. The grammar language is fully backward compatible.

14.1. Migrating Step By Step

Once again, 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.

14.1.1. 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 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.

14.1.2. 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 subpackages" in the rename 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.

14.1.3. 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.

14.1.4. 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*.

- 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 comparison 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 *MyDslExecutableExtensionFactory* 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 *MyDslExecutableExtensionFactory* and thus not covered by the validation of the *plugin.xml*.
- 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.

14.1.5. 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 *MyDslStandaloneSetup* inheriting form the generated *MyDslStandaloneSetupGenerated* 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.

- The method IScopeProvider.getScope(EObject,EClass) has been removed. Use getScope(EObject,EReference) instead.
- Renamed *DefaultScopeProvider* to SimpleLocalScopeProvider. There have been further significant changes in the scoping API that allow for optimized implementations. Consult the section on scoping (§8.6) for details.
- The return type of AbstractInjectableValidator.getEPackages() was changed from List<? extends EPackage> to List<EPackage>.
- The parser interfaces now use Reader instead of InputStream to explicitly address encoding. Have a look at the section on encoding (§8.11) 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 (§9.1) 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 *MyDslUiModule* 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*.
  Note: The outline has been re-implemented in Xtext 2.0.
- 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*.
  Note: IStateAccess was split into IReadAccess and IWriteAccess in Xtext 2.0.
- 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 (§8.6.1) 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 (§8.10) 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.

For an overview over the new features in Xtext 1.0 consult our New and Noteworthy online.

List of External Links

http://code.google.com/p/google-guice/
http://www.ietf.org/rfc/rfc2396.txt
http://www.eclipse.org/Xtext/documentation/helios/new_and_noteworthy.php
http://docs.oracle.com/javase/specs/jls/se7/html/index.html
http://www.xtend-lang.org
http://www.eclipse.org/forums/index.php?t=thread&frm_id=27
http://download.eclipse.org/modeling/emf/emf/javadoc/2.6.0/org/eclipse/emf/ecore/change/package-summary.html
http://www.eclipse.org/Xtext/documentation/indigo/new_and_noteworthy.php
http://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html#jls-5.5
http://git.eclipse.org/c/tmf/org.eclipse.xtext.git/tree/plugins/org.eclipse.xtext/src/org/eclipse/xtext/Xtext.xtext
http://xtext.itemis.com
http://projectlombok.org/features/SneakyThrows.html
http://help.eclipse.org/ganymede/topic/org.eclipse.cdt.doc.user/tasks/cdt_t_imp_code_temp.htm
http://docs.oracle.com/javase/specs/jls/se7/html/jls-3.html#jls-3.10.5
http://martinfowler.com/bliki/SyntacticNoise.html
http://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html
http://martinfowler.com/books.html#dsl
http://www.eclipse.org/modeling/emf/
http://docs.oracle.com/javase/specs/jls/se7/html/jls-3.html#jls-3.7
http://blogs.itemis.de/stundzig/archives/773
http://blog.efftinge.de/2009/01/xtext-scopes-and-emf-index.html
http://git.eclipse.org/c/tmf/org.eclipse.xtext.git/tree/examples/org.eclipse.xtext.xtext.ui.examples/contents
http://download.oracle.com/javase/1.5.0/docs/guide/intl/encoding.doc.html
http://www.xtend-lang.org/documentation
http://www.eclipse.org/modeling/gmp/?project=gmf

Todo list