ESI

Radboud University

Advanced Xtext Manual on Modularity

Arjan Mooij - ESI (TNO) Eindhoven

Jozef Hooman - ESI (TNO) Eindhoven & Radboud University Nijmegen

V3.0, 2019-07-26 – Eclipse 2019-06 & Xtext 2.18

---

1 Overview

This manual assumes that the reader is familiar with our basic manual [1] on creating Domain Specific Languages (DSLs) with Xtext. In this manual, we address more advanced topics, especially concerning modularity. Section 2 introduces the relevant EMF concepts. Section 3 shows references between files in the runtime workspace. Section 4 explains how DSLs can be combined at the meta-level. An advanced application of sharing parts of grammar is treated in Section 5.

2 EMF concepts

In this section we briefly introduce the EMF concepts that we will use later on. See [2] for an EMF tutorial.

2.1 Resources and EObjects

We first introduce the following EMF concepts:

- *URI*: Uniform Resource Identifier, which can be absolute or relative. URI objects can be *created* based on a string representation of the URI. To turn a relative URI into an absolute URI, it needs to be *resolved* against another absolute URI.

- *EObject*: A modeled EMF object, typically related to a non-terminal. It includes:
  - *EClass*: The corresponding non-terminal from the DSL grammar.

- *Resource*: provides access to a persistent document (e.g., a file on disk). It includes:
  - Absolute URI of the resource.
  - ResourceSet that contains the Resource.
  - Tree of modeled contents, each node is represented in terms of an EObject. The head of the tree corresponds to the first non-terminal in the DSL.
- *ResourceSet*: a tree of related Resource objects, including a notification mechanism for changes in these objects. A specific Resource can be obtained based on an absolute URI.

We will frequently use the following library to convert between URI, Resource and EObject:

```
import org.eclipse.emf.common.util.URI
import org.eclipse.emf.ecore.EObject
import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.emf.ecore.resource.ResourceSet
```

```
class ResourceUtils {
	def static EObject resourceToEObject(Resource resource) {
		return resource?.allContents?.head;
	}

	def static Resource openImport(Resource currentResource, String
importedURIAsString) {
		val URI currentURI = currentResource?.getURI;
		val URI importedURI = URI?.createURI(importedURIAsString);
		val URI resolvedURI = importedURI?.resolve(currentURI);

		val ResourceSet currentResourceSet = currentResource?.resourceSet;
		val Resource resource = currentResourceSet?.getResource(resolvedURI,
true);
		return resource;
	}
}
```

Note the use of Xtend's null-safe navigation operator (indicated by a question mark) to avoid explicit tests on null values; e.g., "resource?.x" is an abbreviation for: "**if** resource == null **then** null **else** resource.x".

2.2 Resource and EObject descriptions

Internally, an index is maintained containing several kinds of descriptions, including:

- *IResourceDescription*: provides information about a Resource, including:
  - URI of the described Resource.
  - List of exported objects, each represented by an IEObjectDescription. By default, the exported objects are the EObjects for which the QualifiedNameProvider computes a name different from null.

- *IEObjectDescription*: provides information about an EObject, including:
  - Qualified name of the described EObject, as computed by the QualifiedNameProvider.
  - URI of the described EObject.
  - EClass of the described EObject.
  - User data for the described EObject, by default it is empty.

The IResourceDescriptions and IEObjectDescriptions are updated in the following situations:

- They are incrementally updated when editing a file within Eclipse (as long as "Project -> Build Automatically" is not disabled);
- They are completely refreshed when performing "Project -> Clean.. -> Clean all projects".

When processing DSL instance files there are two phases with the following steps:

1. For each individual file that has been edited:
   a. Lexing the model into a token stream
   b. Parsing the token stream into a parse tree
   c. Creating an IResourceDescription (and IEObjectDescriptions) for the parse tree

2. For each individual file that has been edited (or a referred file has been edited):
    a. Lexing the model into a token stream
    b. Parsing the token stream into a parse tree
    c. Resolving references (use IResourceDescriptions of referred files) in the parse tree
    d. Performing validation
    e. Performing generators (if there are no referencing or validation errors)

3 References between files in the runtime workspace

In this section we consider a single DSL, and focus on references between multiple files in the runtime workspace.

3.1 Preliminaries (Create a new project)

In this subsection we discuss some prerequisites for the following subsections:

- Create an empty folder ModularityMeta
- Open Eclipse with this folder as workspace
- Create a new Xtext project using the default settings (Project name: org.xtext.example.mydsl; Language Name: org.xtext.example.mydsl.MyDsl Extensions: mydsl)

3.2 Default behavior

Based on the basic material from [1], it is already possible to make references between files in the runtime workspace. In this section we introduce a running example that demonstrates the default behavior of Xtext with respect to references. This example is also used in the following sections when we discuss changing the default behavior of Xtext.

3.2.1 Change the grammar

Perform the following steps:

- Edit MyDsl.xtext as follows:

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

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

Person:
	'I' 'am' name=ID
	greetings+=HelloGreeting*
;

HelloGreeting:
	'Hello' person=[Person] '!'
;
```

- Generate the language infrastructure and launch the runtime workspace.
- Create in the runtime workspace a general project with a folder containing two files:
    - basic1.mydsl:

```
I am A
Hello B !
```

- basic2.mydsl:

```
I am B
Hello A !
```

Both files refer to a person declared in the other file. Note that navigation <F3> and content-assist (<CTRL>-<SPACE>) work across files. This is a convenient way of validating the implemented reference mechanism.

3.2.2 Change the generator

Close the runtime workspace and perform the following in the meta-level workspace:

- Create an Xtend Class file ResourceUtils.xtend in org.xtext.example.mydsl.generator as described in section 2.1.
- It might be possible that Eclipse gives the following error:

  The declared package 'null' does not match the expected package '...'

  The easiest way to repair this is to apply the proposed quick-fix:

  Change package declaration to '...
- Edit MyDslGenerator.xtend as follows:

```
package org.xtext.example.mydsl.generator

import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.AbstractGenerator
import org.eclipse.xtext.generator.IFileSystemAccess2
import org.eclipse.xtext.generator.IGeneratorContext
import org.xtext.example.mydsl.myDsl.Person

class MyDslGenerator extends AbstractGenerator {

	override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
IGeneratorContext context) {
		val root = ResourceUtils.resourceToEObject(resource) as Person;
		fsa.generateFile(resource.getURI().lastSegment + '.txt',
toText(root));
	}

	def static toText(Person person) '''
		Person: «person.name»
		Greetings to: «FOR greeting : person.greetings»
«greeting.person.name» («greeting.person.greetings.size»)«ENDFOR»
	'''
}
```

- Launch the runtime workspace, perform a project clean, and observe that the generated text files contain information from both files.

3.3 Global scope provider

In section 3.2, we have seen that with the default settings, instance files can refer to objects in any other file in the Eclipse project. This behavior is similar to Java-like languages, but sometimes it is desired to have more control over the accessible files, e.g., in case there are multiple files that define objects with the same name.

The global scope provider determines which resources are in scope. There are two Xtext policies:

- *DefaultGlobalScopeProvider*: looks for resources in visible containers (i.e., project or package) from the referencing model.
- *ImportUriGlobalScopeProvider*: looks for valid URIs in any "importURI" attribute of any EObject in the referencing model.

The default global scope provider is DefaultGlobalScopeProvider, but this can be changed into ImportUriGlobalScopeProvider as explained in the following subsections.

3.3.1 Change the global scope provider

- Close the runtime workspace and replace MyDslRuntimeModule.xtend file in src/org.xtext.example.mydsl by the following:

```
package org.xtext.example.mydsl

import org.eclipse.xtext.scoping.IGlobalScopeProvider
import org.eclipse.xtext.scoping.impl.ImportUriGlobalScopeProvider

class MyDslRuntimeModule extends AbstractMyDslRuntimeModule {
	override Class<? extends IGlobalScopeProvider> bindIGlobalScopeProvider() {
		return typeof(ImportUriGlobalScopeProvider);
	}
}
```

The override keyword in this code fragment indicates that the ImportUriGlobalScopeProvider must be used instead of the binding defined in AbstractMyDslRuntimeModule (which happens to be DefaultGlobalScopeProvider).

- Generate the language infrastructure again.
- Open the runtime workspace.
- As we have changed the global scope provider, first clean all projects (Project -> Clean… -> Clean all projects) in order to refresh the Eclipse index.

Observe that references to objects in other files are no longer allowed. The reason is that the ImportUriGlobalScopeProvider requires explicit imports.

3.3.2 Change the grammar

- Close the runtime workspace and edit the Xtext grammar as follows:

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

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

Person:
	'I' 'am' name=ID
	(greetings+=HelloGreeting | imports+=Import )*
;

HelloGreeting:
	'Hello' person=[Person] '!'
;

Import:
```

```
	'import' importURI=STRING
;
```

As mentioned in the explanation of ImportUriGlobalScopeProvider in the introduction of Section 3.3, the attribute name importURI is crucial in this case. Note the special syntax highlighting in the Xtext grammar for “importURI” attributes, similar to the special “name” attribute.

- Generate the language infrastructure and next open the runtime workspace.
- Add an import to the two files basic1.mydsl and basic2.mydsl as follows:
  - basic1.mydsl:

```
I am A
Hello B !
import "basic2.mydsl"
```

  - basic2.mydsl:

```
I am B
Hello A !
import "basic1.mydsl"
```

Note that, also in this case navigation <F3> and content-assist (<CTRL>-<SPACE>) work across files. This is a convenient way of validating the implemented reference mechanism. Note that imports are not transitive; if file basic3.mydsl only imports basic2.mydsl, it cannot refer to Person A of basic1.mydsl.

3.3.3 Change the generator

The generator from Section 3.2 also works in this case. In addition, we show what we can do with the explicit imports. Close the runtime workspace.

- Edit the file MyDslGenerator.xtend as follows (in case of errors, restart Eclipse).

```
package org.xtext.example.mydsl.generator
import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.AbstractGenerator
import org.eclipse.xtext.generator.IFileSystemAccess2
import org.eclipse.xtext.generator.IGeneratorContext
import org.xtext.example.mydsl.myDsl.Import
import org.xtext.example.mydsl.myDsl.Person

class MyDslGenerator extends AbstractGenerator {

	override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
IGeneratorContext context) {
		val root = ResourceUtils.resourceToEObject(resource) as Person;
		fsa.generateFile(resource.getURI().lastSegment + '.txt',
toText(root))
	}

	def static toText(Person person) '''
		Person: «person.name»
		Greetings to: «FOR greeting : person.greetings»
«greeting.person.name» («greeting.person.greetings.size»)«ENDFOR»
		«FOR imp: person.imports»
		«toText(imp)»
		«ENDFOR»
	'''
```

```
	def static toText(Import imp) {
		val Person importedPerson =
	ResourceUtils.resourceToEObject(ResourceUtils.openImport(imp.eResource,
	imp.importURI)) as Person;
		return '''
			Imported file: «imp.importURI»:
				«IF importedPerson === null»
				File cannot be opened.
				«ELSE»
				Number of greetings: «importedPerson.greetings.size»
				«ENDIF»
		''';
	}
}
```

Method toText now also accesses the imported files, and generate text that shows the number of greetings in the imported files.

- Observe the results in the runtime workspace in the src-gen folder.

3.4 Qualified name provider

The qualified name provider determines the qualified names of EObject that can be referred. The default implementation is called DefaultDeclarativeQualifiedNameProvider, which computes a qualified name for each EObject based on the “name” attribute of the EObject, and the “name” attribute of the containing parent EObjects. In this way a hierarchical name is constructed, in which the names are separated by a dot (‘.’).

In the following subsections we show how we can benefit from this hierarchical name, and how we can change the qualified name of an EObject. First we remove a few changes made earlier. Use the default scope by changing MyDslRuntimeModule.xtend file in src/org.xtext.example.mydsl into

```
class MyDslRuntimeModule extends AbstractMyDslRuntimeModule {
}
```

Moreover, make the generator empty:

```
class MyDslGenerator extends AbstractGenerator {

	override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
IGeneratorContext context) {
	}
}
```

3.4.1 Use qualified names inside references

To refer to EObjects in terms of their qualified name, it is enough to change the grammar for the references.

- Edit the Xtext grammar as follows:

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

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

```
Person:
	'I' 'am' name=ID
	devices+=Device*
	(greetings+=HelloGreeting | pings+=Ping)*
;

Device:
	'I' 'own' 'device' name=ID
;

HelloGreeting:
	'Hello' person=[Person] '!'
;

Ping:
	'Ping' device=[Device|QualifiedName]
;

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

Note the special syntax highlighting in the Xtext grammar for “name” attributes, and for rule “QualifiedName”. The rule for QualifiedName is not a normal terminal or non-terminal rule, but a so-called data-type rule, which combines several terminals (in this case ID and “.”) into a single value. In a way it looks similar to a terminal rule (which are processed by the lexer), but a data type rule is processed by the parser, and hence is context-sensitive and allows for hidden tokens (such as whitespace).

In this example, a person not only has a name, but may also own multiple devices. In addition to greeting a person, there is the possibility to ping any device (of yourself or another person). To distinguish between devices with the same name from the different persons, qualified names can be used for the ping.

- Generate the language infrastructure and next open the runtime workspace.
- Edit the two files basic1.mydsl and basic2.mydsl as follows:
  - basic1.mydsl:

```
I am A
I own device Smartphone
I own device Tablet

Hello B !
Ping Smartphone
Ping A.Smartphone
Ping B.Smartphone
```

  - basic2.mydsl:

```
I am B
I own device Smartphone
I own device Tablet

Hello A !
```

Note that, also in this case, navigation <F3> and content-assist (<CTRL>-<SPACE>) work across files. This is a convenient way of validating the implemented reference mechanism. In particular, experiment with content-assist after typing the keyword “Ping”.

3.4.2 Change the qualified name provider (names based on attributes)

To change the computed qualified names, we create a custom qualified name provider and bind it in MyDslRuntimeModule.xtend.

- Create a new Xtend Class file MyDslQualifiedNameProvider.xtend in folder src/org.xtext.example.mydsl of project org.xtext.example.mydsl:

```
package org.xtext.example.mydsl

import com.google.inject.Inject
import org.eclipse.emf.ecore.EObject
import org.eclipse.xtext.naming.DefaultDeclarativeQualifiedNameProvider
import org.eclipse.xtext.naming.IQualifiedNameConverter
import org.eclipse.xtext.naming.QualifiedName
import org.xtext.example.mydsl.myDsl.Person


class MyDslQualifiedNameProvider extends DefaultDeclarativeQualifiedNameProvider {
	@Inject
	IQualifiedNameConverter converter = new
IQualifiedNameConverter.DefaultImpl();

	override QualifiedName getFullyQualifiedName(EObject obj) {
		if (obj instanceof Person) {
			val text = obj.getName();
			if (text !== null) {
				return converter.toQualifiedName("Prof" +
text.toUpperCase);
			}
		}

		return super.getFullyQualifiedName(obj);
	}
}
```

- Replace MyDslRuntimeModule.xtend file in src/org.xtext.example.mydsl by the following:

```
package org.xtext.example.mydsl

import org.eclipse.xtext.naming.IQualifiedNameProvider

class MyDslRuntimeModule extends AbstractMyDslRuntimeModule {
	override Class<? extends IQualifiedNameProvider>
bindIQualifiedNameProvider() {
		return typeof(MyDslQualifiedNameProvider);
	}
}
```

- Relaunch the Runtime workspace.

- As we have changed the qualified name provider, first clean all projects (Project -> Clean… -> Clean all projects) in order to refresh the Eclipse index.
- Then try out the following example:
  - basic1.mydsl:

```
I am a
Hello ProfA !
```

Note that, also in this case, navigation <F3> and content-assist (<CTRL>-<SPACE>) work as usual. This is a convenient way of validating the implemented reference mechanism.

3.4.3 Change the qualified name provider (names based on references)

Sometimes the intended custom name of an element is not available as an attribute, but only as a reference to another element. As the qualified name provider is invoked before the reference resolving, the qualified name provider cannot directly access such references. In this subsection we explain a solution.

- Edit MyDsl.xtext to

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

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

Person:
	'I' 'am' name=ID
	greetings+=HelloGreeting*
	alsoGreetings+=AlsoHelloGreeting*
;

HelloGreeting:
	'Hello' person=[Person] '!'
;

AlsoHelloGreeting:
	'Also' 'Hello' greeting=[HelloGreeting] '!'
;
```

In this example, it is desired that AlsoHelloGreeting refers to a HelloGreeting instead of a Person (for which there may not be any greeting). For example, we could extend HelloGreeting with additional attributes that are relevant for the code generators of HelloGreeting and AlsoHelloGreeting. HelloGreeting has no name attribute, but a reference to a Person. Again, we can use a qualified name provider, but we cannot directly access the reference, as this would lead to a circular dependency between name provider and reference resolver.

- Edit MyDslQualifiedNameProvider.xtend in folder src/org.xtext.example.mydsl of project org.xtext.example.mydsl as follows:

```
package org.xtext.example.mydsl

import com.google.inject.Inject
import org.eclipse.emf.ecore.EObject
import org.eclipse.xtext.naming.DefaultDeclarativeQualifiedNameProvider
import org.eclipse.xtext.naming.IQualifiedNameConverter
```

```
import org.eclipse.xtext.naming.QualifiedName
import org.eclipse.xtext.nodemodel.util.NodeModelUtils
import org.xtext.example.mydsl.myDsl.HelloGreeting
import org.xtext.example.mydsl.myDsl.MyDslPackage

class MyDslQualifiedNameProvider extends DefaultDeclarativeQualifiedNameProvider {
	@Inject
	IQualifiedNameConverter converter = new
IQualifiedNameConverter.DefaultImpl();

	override QualifiedName getFullyQualifiedName(EObject obj) {
		if (obj instanceof HelloGreeting) {
			val nodes = NodeModelUtils.findNodesForFeature(obj,
MyDslPackage.Literals.HELLO_GREETING__PERSON);
			for (node : nodes) {
				val text = NodeModelUtils.getTokenText(node);
				if (text !== null) {
					return converter.toQualifiedName(text);
				}
			}
		}

		return super.getFullyQualifiedName(obj);
	}
}
```

The method findNodesForFeature gives the parse tree nodes for the person attribute of HelloGreeting. The method getTokenText converts a parse tree node into plain text. Finally, the toQualifiedName method translates the plain text into a qualified name.

- Generate the language infrastructure.
- Launch the Runtime workspace.
- As we have changed the QualifiedNameProvider, first clean all projects (Project -> Clean… -> Clean all projects) in order to refresh the Eclipse index.
- Then try the following example:
  - basic1.mydsl:
    ```
    I am A
    Hello B !
    ```
  - basic2.mydsl:
    ```
    I am B
    Also Hello B!
    ```

Note that, also in this case, navigation <F3> and content-assist (<CTRL>-<SPACE>) work as usual. This is a convenient way of validating the implemented reference mechanism. In particular note that navigation <F3> from the B on the last line jumps to “Hello B” instead of “I am B”, as it is a reference to HelloGreeting instead of Person.

4 Combining DSLs in the meta-level workspace

In this section we consider multiple DSLs, and focus on how DSLs can be combined in the meta-level workspace. There are two kinds of ways to combine DSLs in the meta-level workspace:

- Enable references to objects of another DSL. For example, when linking related models that are defined in different DSLs.

- Reuse grammar elements from another DSL. For example, when building a library with basic grammar definitions that are used in multiple DSLs.

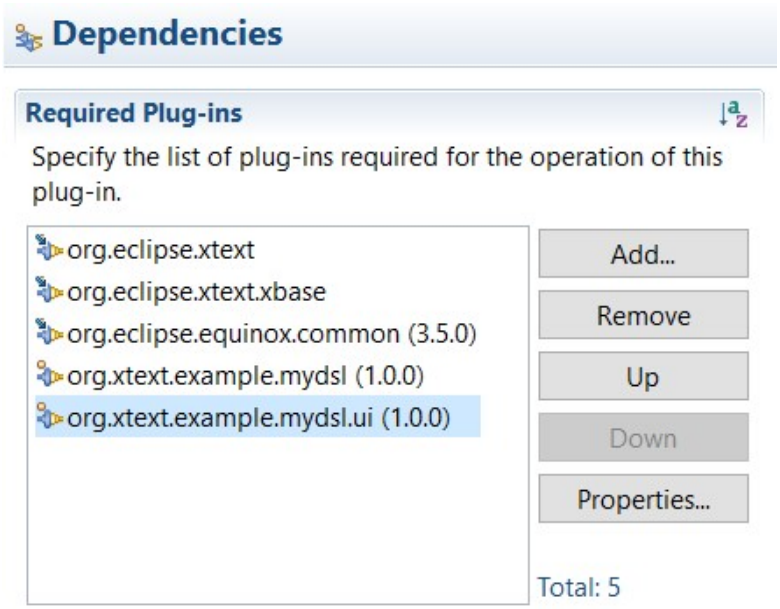
As an example, we introduce a second DSL that does not define any Person, but that does support greetings with references to Persons from the first DSL. To distinguish the two types of greetings, we call them HelloGreeting and HiGreeting respectively.

4.1 Preliminaries (Create a new project with modified manifest)

In this subsection we discuss some prerequisites for the following subsections:

- Create a new Xtext project in the workspace ModularityMeta used in the previous sections. Most of the default settings are modified automatically by adding a 1 at the end (Project name: org.xtext.example.mydsl1; Extensions: mydsl1). In addition manually change Language Name to org.xtext.example.mydsl1.MyDsl1 (add a 1 at the end).

- In MANIFEST.MF in org.xtext.example.mydsl1/META-INF: in tab "Dependencies" under "Required Plug-ins", add the project that contains the imported .xtext file; in this case: org.xtext.example.mydsl. Moreover, add org.xtext.example.mydsl.ui.



- Do the same in the MANIFEST.MF files at the following locations:
  - org.xtext.example.mydsl1.ui/META-INF
  - org.xtext.example.mydsl1.ide/META-INF (only if this optional project exists)

4.2 References to objects of another DSL

4.2.1 Change the grammar

- In the MyDsl1.xtext file in this new project, add an import statement referring to the other grammar (the URI can be found after "generate" in the imported .xtext file):

```
grammar org.xtext.example.mydsl1.MyDsl1 with org.eclipse.xtext.common.Terminals

generate myDsl1 "http://www.xtext.org/example/mydsl1/MyDsl1"
```

```
import "http://www.xtext.org/example/mydsl/MyDsl" as imported

JustGreetings:
	greetings+=HiGreeting *
;

HiGreeting:
	'Hi' person=[imported::Person] '!'
;
```

- In the GenerateMyDsl1.mwe2 file (in org.xtext.example.mydsl1/src/ org.xtext.example.mydsl1), inside the StandardLanguage{ ... } block, add the following line after the fileExtensions attribute:

  ```
  referencedResource =
  "platform:/resource/org.xtext.example.mydsl/model/generated/MyDsl.genmodel"
  ```

  (which refers to the location of the genmodel file of the referenced grammar)

In general, the genmodel file can be found in folder model/generated of the referenced language project. Multiple genmodels can be referenced by adding multiple referencedResource lines.

- Generate the language infrastructure for the second Xtext project.
- Launch the Runtime workspace and create in a new folder a file
  - import.mydsl1:

    ```
    Hi B!
    Hi A!
    ```

Note that we can refer to all persons declared in .mydsl files in the same project. If we would like to refer to other persons defined in .mydsl file outside of the current project, we should use the ImportUriGlobalScopeProvider as described in Section 3.3.

4.2.2 Change the generator

In the generator, one can refer to objects of the other DSL in the usual way.

- Close the Runtime workspace
- Edit the MyDsl1Generator.xtend file of the second project as follows:

```
package org.xtext.example.mydsl1.generator

import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.AbstractGenerator
import org.eclipse.xtext.generator.IFileSystemAccess2
import org.eclipse.xtext.generator.IGeneratorContext
import org.xtext.example.mydsl1.myDsl1.HiGreeting
import org.xtext.example.mydsl1.myDsl1.JustGreetings
import org.xtext.example.mydsl.generator.ResourceUtils

class MyDsl1Generator extends AbstractGenerator {

	override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
IGeneratorContext context) {
```

```
                    val root = ResourceUtils.resourceToEObject(resource) as
JustGreetings
                    fsa.generateFile(resource.getURI().lastSegment + ".txt",
toText(root));

      }

      def static toText(JustGreetings newgreet) '''
            New Person Greetings
                  «FOR greet: newgreet.greetings»
                  «toText(greet)»
                  «ENDFOR»
      '''

      def static toText(HiGreeting greet)'''
            Say hi to «greet.person.name» («greet.person.greetings.size»)
      '''
}
```

- Launch the Runtime workspace.
- Clean the project and inspect the generated file.

4.3 Extend the grammar of another DSL

4.3.1 Change the grammar

- Edit MyDsl1.xtext to

```
grammar org.xtext.example.mydsl1.MyDsl1 with org.xtext.example.mydsl.MyDsl

generate myDsl1 "http://www.xtext.org/example/mydsl1/MyDsl1"

Group:
      'Group'
      persons+=Person+
;
```

This DSL makes it possible to define a group of persons in a single file; the grammar is very short, because we reuse the earlier defined grammar from org.xtext.example.mydsl.

The first line (especially the “with”-clause) indicates that we are extending grammar MyDsl. This inheritance mechanism is transitive, so in MyDsl1 we can use the grammar elements from both MyDsl and Terminals (which was extended by MyDsl).

- Generate the language infrastructure

4.3.2 Change the generator

- Adapt MyDslGenerator.xtend of the first DSL into

```
package org.xtext.example.mydsl.generator

import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.AbstractGenerator
import org.eclipse.xtext.generator.IFileSystemAccess2
import org.eclipse.xtext.generator.IGeneratorContext
import org.xtext.example.mydsl.myDsl.Person
```

```
class MyDslGenerator extends AbstractGenerator {

	override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
IGeneratorContext context) {
		val root = ResourceUtils.resourceToEObject(resource) as Person;
		fsa.generateFile(resource.getURI().lastSegment + '.txt',
toText(root));
	}

	def static toText(Person person) '''
		Person: «person.name»
		Greetings to: «FOR greeting : person.greetings»
«greeting.person.name» («greeting.person.greetings.size»)«ENDFOR»
	'''
}
```

- Adapt MyDsl1Generator.xtend into

```
package org.xtext.example.mydsl1.generator

import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.AbstractGenerator
import org.eclipse.xtext.generator.IFileSystemAccess2
import org.eclipse.xtext.generator.IGeneratorContext
import org.xtext.example.mydsl.generator.MyDslGenerator
import org.xtext.example.mydsl1.myDsl1.Group
import org.xtext.example.mydsl.generator.ResourceUtils

class MyDsl1Generator extends AbstractGenerator {

	override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
IGeneratorContext context) {
		val root = ResourceUtils.resourceToEObject(resource) as Group
		fsa.generateFile(resource.getURI().lastSegment + ".txt",
toText(root));

	}

	private def static toText(Group group) '''
		Group members
			«FOR person: group.persons»
			«MyDslGenerator.toText(person)»
			«ENDFOR»
	'''
}
```

Note that the generator is very short, because we reuse the earlier defined generator from org.xtext.example.mydsl.

Experiment with the new grammar in the runtime workspace. For example, create a file group.mydsl1 with the following contents:

```
Group
I am A
I am B
```

5 Sharing parts of a grammar

In this section, we show how a certain grammar structure can be shared. As an example, suppose we have a grammar with two types: Item and Aspect. The aim is to define ItemExpressions and AspectExpressions with the structure of Boolean expressions, but with different basic elements, namely Item and Aspect, respectively. An ItemExpression referring to an Aspect should lead to an error and vice versa. The goal is illustrated by the next example:

```
Item A
Item B
Item C

Aspect X
Aspect Y
Aspect Z

Item Expression: A and (B or C)
Aspect Expression: X or (Y and !Z)

Item Expression: A or X
Aspect Expression: A or X
```

For general information about parsing expressions, we refer to our basic manual [1]. Here we show how this specific goal can be achieved without duplicating the commonalities between ItemExpressions and AspectExpressions.

5.1 Preliminaries (Create a new project)

In this subsection we discuss some prerequisites for the following subsections:

- Create an empty folder SharingPartsMeta
- Open Eclipse with this folder as workspace
- Create a new Xtext project using the default settings (Project name: org.xtext.example.mydsl; Language Name: org.xtext.example.mydsl.MyDsl Extensions: mydsl)

5.2 Change the grammar

Perform the following steps:

- Edit MyDsl.xtext as follows:

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

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

Model:
	(aspectDeclarations+=AspectDeclaration |
	 itemDeclarations+=ItemDeclaration |
	 expressions+=Expression
	)*;

Declaration: // Do not remove!
	AspectDeclaration | ItemDeclaration
```

```
;

AspectDeclaration:
	'Aspect' name=ID
;

ItemDeclaration:
	'Item' name=ID
;

Expression:
	AspectExpression | ItemExpression
;

AspectExpression:
	'Aspect' 'Expression' ':' expression=BooleanExpression
;

ItemExpression:
	'Item' 'Expression' ':' expression=BooleanExpression
;

// === Boolean expression =======

BooleanExpression:
	BooleanExpressionLevel1
;

BooleanExpressionLevel1 returns BooleanExpression:
	BooleanExpressionLevel2
	( {BoolenExpressionLevel2.left=current}
		operator=BinaryBooleanOperator
		  right=BooleanExpressionLevel2
	)*
;

BooleanExpressionLevel2 returns BooleanExpression:
	UnaryBooleanExpression | BooleanExpressionLevel3
;

UnaryBooleanExpression:
	operator=UnaryBooleanOperator expression=BooleanExpressionLevel3
;

BooleanExpressionLevel3 returns BooleanExpression:
	BooleanBrackets | ComponentReference
;

BooleanBrackets:
	'(' expression = BooleanExpression ')'
;

ComponentReference:
	component=[Declaration|ID]
;

enum BinaryBooleanOperator:	and = 'and' | or = 'or' | xor = 'xor';
enum UnaryBooleanOperator:	not = '!';
```

Note that the definition of “Declaration” is gray as the parser does not use the rule. However, it causes “Declaration” to be a superclass of AspectDeclaration and ItemDeclaration, and this is used by the reference in the rule for “ComponentReference”.

- Generate the language infrastructure and next open the runtime workspace.
- Create a file test.mydsl with the following contents :

```
Item A
Item B
Item C

Aspect X
Aspect Y
Aspect Z

Item Expression: A and (B or C)
Aspect Expression: X or (Y and !Z)

Item Expression: A or X
Aspect Expression: A or X
```

Note that this grammar allows (gives no error) the occurrence of, for instance, an Aspect in an ItemExpression.

5.3 Change the local scope provider

The scope provider defines the scopes for references in their certain context. We modify the scope provider to exclude unintended combinations of Aspects and Items.

Perform the following steps:

- Edit file MyDslScopeProvider.xtend in org.xtext.example.mydsl.scoping:

```
package org.xtext.example.mydsl.scoping

import org.eclipse.emf.ecore.EObject
import org.eclipse.emf.ecore.EReference
import org.eclipse.xtext.scoping.Scopes
import org.xtext.example.mydsl.myDsl.AspectExpression
import org.xtext.example.mydsl.myDsl.ItemExpression
import org.xtext.example.mydsl.myDsl.Model
import org.xtext.example.mydsl.myDsl.MyDslPackage.Literals

class MyDslScopeProvider extends AbstractMyDslScopeProvider {
	override getScope(EObject context, EReference reference) {
		if (reference == Literals.COMPONENT_REFERENCE__COMPONENT) {
			var ancestor = context;
			while (!(ancestor === null || ancestor instanceof
AspectExpression || ancestor instanceof ItemExpression)) {
				ancestor = ancestor.eContainer();
			}
			val model = ancestor?.eContainer as Model;
			if (ancestor instanceof AspectExpression) {
				return Scopes.scopeFor(model.aspectDeclarations);
			}
			if (ancestor instanceof ItemExpression) {
				return Scopes.scopeFor(model.itemDeclarations);
			}
```

```
            }
            return super.getScope(context, reference);
        }
}
```

If the ComponentReference occurs in the context of an AspectExpression, then only the AspectDeclarations are in scope; similarly if the ComponentReference occurs in the context of an ItemExpression. In both cases, we use that the parent of the AspectExpression or ItemExpression element is a Model element, which contains the relevant declarations.

It can be verified in the runtime workspace that we indeed obtain the desired DSL instance, as mentioned in the first paragraph of this section.

References

[1] A. Mooij and J. Hooman, Creating a Domain Specific Language (DSL) with Xtext, http://www.cs.kun.nl/J.Hooman/DSL/Creating_a_Domain_Specific_Language_(DSL)_with_Xtext.pdf

[2] EMF Tutorial - What every Eclipse developer should know about EMF http://eclipsesource.com/blogs/tutorials/emf-tutorial/