1 Scoping

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

Example:

```
events
    nothingImportant  MYEV
end

state idle
    nothingImportant => idle
end
```

The grammar rule for transitions looks like this:

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

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

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

```
public interface IScopeProvider {

    /**
    * Returns a scope for the given context. The scope
    * provides access to the compatible visible EObjects
    * for a given reference.
    *
    * @param context the element from which an element shall
      be referenced
    * @param reference the reference to be used to filter the
      elements.
```

```
     * @return {@link IScope} representing the inner most {
        @link IScope} for
     *         the passed context and reference. Note for
        implementors: The
     *         result may not be <code>null</code>. Return
     *         <code>IScope.NULLSCOPE</code> instead.
     */
     getScope(EObject context, EReference reference);

}
```

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

1.1 Global Scopes and *IResourceDescriptions*

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

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

1.2 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 *org.eclipse.xtext/src/org.eclipse.xtext.resource.IResourceDescription*.

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

For users and especially in the context of scoping the most important information is the list of exported *EObjects*. An *org.eclipse.xtext/src/org.eclipse.xtext.resource.IEObjectDescription* contains information about the *URI* to the actual *EObject* and the qualified name of

that element as well as the corresponding *EClass*. In addition one can export arbitrary information using the *user data* map. The following diagram gives an overview on the description classes and their relationships.

A language is configured with a default implementation of *IResourceDescription.Manager* which computes the list of exported *IEObjectDescriptions* by iterating the whole EMF model and applying the *getQualifiedName(EObject obj)* from *org.eclipse.xtext/src/org.eclipse.xtext.naming.IQual* on each *EObject*. If the object has a qualified name an *IEObjectDescription* is created and exported (i.e. added to the list). If an *EObject* doesn't have a qualified name, the element is considered to be not referable from outside the resource and consequently not indexed. If you don't like this behavior, you can implement and bind your own implementation of *IResourceDescription.Manager*.

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

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

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

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

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

```
@Inject
private IResourceServiceProvider.Registry
    resourceServiceProviderRegistry;

private IResourceDescription.Manager getManager(Resource res)
    {
IResourceServiceProvider resourceServiceProvider =
```

```
resourceServiceProviderRegistry.getResourceServiceProvider(res
    .getURI());
return resourceServiceProvider.getResourceDescriptionManager()
    ;
}
```

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

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

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

1.2.1 Global Scopes Based On Explicit Imports (ImportURI Mechanism)

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

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

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

This effectively allows import statements to be declared before the events section. In addition you'll have to make sure that you have bound the *org.eclipse.xtext/src/org.eclipse.xtext.scoping.impl.Imp* for the type *IGlobalScopeProvider* by the means of Guice ( ). 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 *org.eclipse.xtext.generator/src/org.eclipse.xtext.* in the workflow of your language. It will bind an *org.eclipse.xtext/src/org.eclipse.xtext.scoping.impl.ImportUriGlo* (*org.eclipse.xtext/src/org.eclipse.xtext.resource.ignorecase.IgnoreCaseImportUriGlobalScopeProvider* if the *caseInsensitve* flag is set) that handles *importURI* features.

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

Instead of explicitly referring to imported resources, the other possibility is to have some kind of external configuration in order to define what is visible from outside a resource. Java for instances uses the notion of classpaths to define containers (jars and class folders) which contain any referenceable elements. In the case of Java also the order of such entries is important.

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

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 *org.eclipse.xtext/src/org.eclipse.xtext.resource.IResourceDescriptions* (Note the plural form!). The bound singleton in the UI scenario is even aware of unsaved editor changes, such that all linking happens to the latest maybe unsaved version of the resources. You will find the Guice configuration of the global index in the UI scenario in *org.eclipse.xtext.ui.shared/src/org.eclipse.xtext.ui.shared.internal.SharedModule*.

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

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

```
@Inject
IContainer.Manager containerManager;

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

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

p. Xtext offers a couple of strategies for managing containers: If you're running

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

JDT-Based Container Manager

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

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

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

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

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

in the runtime module and

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

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

1.2.3 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();
}
```

1.2.4 *ResourceSet*-Based Containers

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

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

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

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

...
}
```

1.3 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

.

1.3.1 Declarative Scoping

If you have to define scopes for certain contexts, the base class *org.eclipse.xtext/src/org.eclipse.xtext.scoping.impl.* 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 which also corresponds to the *type* parameter.

So if you for example have a state machine with a *Transition* object owned by its source *State* and you want to compute all reachable states (i.e. potential target states), the corresponding method could be declared as follows (assuming the cross-reference is declared by the *Transition* type and is called *target*):

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

If such a method does not exist, the implementation will try to find one for the context object's container. Thus in the example this would match a method with the same name but *State* as the type of the first parameter. It will keep on walking the containment hierarchy until a matching method is found. This container delegation allows to reuse the same scope definition for elements in different places of the containment hierarchy. Also it may make the method easier to implement as the elements comprising the scope are quite often owned or referenced by a container of the context object. In the example the *State* objects could for instance be owned by a containing *StateMachine* object.

If no method specific to the cross-reference in question was found for any of the objects in the containment hierarchy, the implementation will start looking for methods matching the other signature (with the *EClass* parameter). Again it will first attempt 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.

1.4 Imported Namespace-Aware Scoping

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

1.4.1 IQualifiedNameProvider

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

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.

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

1.4.2 Importing Namespaces

The *ImportedNamespaceAwareLocalScopeProvider* looks up *EAttributes* with name 'importNamespace' 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.
a:dependencyInjection