Tutorial Part I: Xtext

Jan Köhnlein, Holger Schill, Dennis Hübner - itemis

Create the Xtext Grammar

In this exercise, we are going to create the basic tooling to create instances of the *Android.ecore* model in the following textual syntax:

```
Application MyContacts

Activity MyContact {
      Text firstName
      Text lastName
      Link phoneNumbers -> PhoneNumber
      Spinner country
      Button save
}

Activity PhoneNumber {
      Text prefix
      Text number
      Button save
}
```

1. Create a *New Xtext Project*:

   Project name: *org.eclipse.ese.android.dsl*
   Language name: *org.eclipse.ese.Android*
   Language extensions: *android*

   Uncheck *Create generator project* and click *Finish.*
2. Xtext generates two projects for you and opens an editor for the Xtext Grammar of your language. An Xtext grammar consists of a grammar declaration. In this case it is

   ```
   grammar org.eclipse.ese.Android
      with org.eclipse.xtext.common.Terminals
   ```

   This declaration defines the name of our language and includes a set of useful terminal rules we will call in our own rules later on. The next line tells Xtext to derive an Ecore model from the grammar. As we do already have an Ecore model and do not want to use the rest of the example grammar either, please delete everything below the grammar declaration.
3. To refer to the existing Ecore model, we have to import it in our grammar. This reads as

   ```
   import "platform:/resource/org.eclipse.ese.android.model/model/
   android.ecore"
   ```

where `platform:/resource` refers to a location in the workspace.

4. The rest of the grammar will be a list of *rules*. A rule defines the lexical structure of our models as well as the construction of the abstract syntax tree (AST), which is an instance of our Ecore model. By default, the name of a rule is the name of the EClass of the EObject that will be created by the rule call. The first rule is the entry rule and will create the root object of our model. In our case, this is of type *Application.* which is defined by the keyword *Application* followed by the application‘s name and any number of activities. In Xtext, this becomes

   ```
   Application:
      'Application' name=ID activities+=Activity+;
   ```

   Keywords are quoted in `''`. *ID* is a so called terminal rule that matches strings similar to Java identifiers. The returned string is assigned (‚=‘) to the name property of the AST EObject returned by the rule. *Activity* is the - yet to be defined - rule that parses the text for activities and returns objects of type *Activity*. These returned objects are stored in the multivalued (‚+=‘) property *activities* of the current *Application* object. The plus sign signals one or any number of occurrences of activities at this position.
5. The editor complains about a missing rule *Activity*. You can use a quick fix (CTRL-1) to create a template for it or write it by hand. An activity is declared by the keyword Activity followed be its name and any number (*) of contained widgets in curly braces (as keywords). Try to figure out yourself how this.
6. A widget is either a text, a button, a spinner, or a link. Using Xtext‘s alternative operator ‚|‘ this is written as

   ```
   Widget:
      Text | Button | Spinner | Link;
   ```

7. Figure out the rules for text, button, and spinner yourself.
8. An object of type *Link* has a cross reference *activity* that points to an existing activity. Use ‚[‘ and ‚]‘ around the type name to denote a cross reference.
9. Compare your grammar to the solution on the next page.
10. Next to the grammar file is a MWE2 workflow file that contains the configuration for Xtext‘s code generator. We are fine with the defaults, just run the code generator by right-clicking on the workflow and selecting *Run As -> MWE2 Workflow*. You can watch the generator‘s log in the console view.
11. Once the generator finishes, start a new runtime Eclipse workbench by choosing *Run As -> Eclipse Application* form the context menu of one of the projects. In the spawned workbench, create a new project and a file with the extension *.android.* Try to create your own android model and watch out for

    • Syntax highlighting
    • Code completion (CTRL-SPACE)
    • Outline View and Quick Outline (CTRL-O)
    • Find References (CTRL-G), Open Declaration (F3)
    • Find Model Element (CTRL-SHIFT-F3)
    • ...

Solution

```
grammar org.eclipse.ese.Android with org.eclipse.xtext.common.Terminals

import "platform:/resource/org.eclipse.ese.android.model/model/
android.ecore"

Application:
      'Application' name=ID activities+=Activity+;

Activity:
      'Activity' name=ID '{' widgets+=Widget* '}';

Widget:
      Text | Button | Spinner | Link;

Text:
      'Text' name=ID;

Button:
      'Button' name=ID;

Spinner:
      'Spinner' name=ID;

Link:
      'Link' name=ID '->' activity=[Activity];
```

Adding Icons - Reflective APIs

The UI shows JFace labels for model elements in several places. JFace labels consist of a text and an image. As the defaults do not look to nice, we are going to customize these now. The component that creates the labels for model elements is called the *LabelProvider*. Xtext offers a reflective API to declare labels for model elements.

1. To work with the JavaBeans interfaces of the Ecore model, add a new dependency to *org.eclipse.ese.android.model* in the *MANIFEST.MF* of the *org.eclipse.ese.android.dsl* and reexport it.
2. Xtext has already created a stub class *AndroidLabelProvider.java*. Open it in an editor.
3. The label provider uses reflection to select the right methods for a given model element type. E.g., to define a new text for an *Application,* you have to implement a method with the signature

   ```
   String text(Application application)
   ```

   Customize some of the texts, e.g. by adding the type of the element.
4. Images are looked up in a folder called *icons/* in the root of the plug-in project *org.eclipse.ese.android.model.ui*. Create it and copy the icons we provided in the model plug-in. To pick an icon for a model element, you have to implement a method *image()*, e.g.

   ```
   String image(Application application) {
      return "Application.gif";
   }
   ```

   Pick up the icons for all types.
5. Start the runtime workbench and watch your labels in action.

Validation and Quick Fixes - Annotation-based APIs

While our grammar already restricts the degrees of freedom for our models a lot, it is often necessary to pose additional rules to ensure a model is semantically consistent. For example, an activity that is not the default (first) activity of an application and is never referred to by a link will never be shown and is therefore obsolete. For cases like this, Xtext provides an API to create warnings and errors.

1. Make sure you have the dependency to *org.eclipse.ese.android.model* in the *MANIFEST.MF* of the *org.eclipse.ese.android.dsl* as described in the previous exercise.
2. Once again, Xtext has already created (and registered) a stub class called *AndroidJavaValidator.java* in which we will implement our validation rules. Open it in an editor.
3. A validation rule is a method with the Java annotation *@Check*. It can have an arbitrary name and must define the element to be checked as a parameter, e.g.

   ```
   @Check
   public void checkApplicationNameWithCapital(Application application) {
   ```

4. Inside this method, you can inspect the argument and call *warning()* or *error()* methods to report issues, e.g.

   ```
   if (!Character.isUpperCase(application.getName().charAt(0))) {
      warning("Name should start with a capital", application,
              AndroidPackage.APPLICATION__NAME, CAPITALIZE);
   }
   ```

   The parameters of these methods are
   • the message,
   • the erroneous model element,
   • a constant denoting the feature to be underlined,
   • a string to identify this issue in quick fixes (optional),
   • additional user data strings to be used in quick fixes (also optional).
5. A link that points to its containing activity might not make to much sense. Write a validation rule that reports a warning if this is the case.
   Hints:
   • The container of an element can be obtained by calling *eContainer()* and cast the result to the expected type, e.g.

     ```
     Activity activity = (Activity) link.eContainer();
     ```

   • The 3rd parameter of the warning method should be

     ```
     AndroidPackage.LINK__NAME
     ```

6. Write an additional validation rule that checks if an activity is either the first activity of an application or referenced by a link.
7. Restart the editor and test the validation rules.

8. Some validation issues can be fixed in an obvious way, e.g. by converting the first letter of a name to upper case. This is called quick fix. As before, the stub class to do that already exists in the file *AndroidQuickfixProvider.java*.
9. As in the validator a quick fix is marked with a Java annotation @Fix. The parameter of this annotation should be the string to denote the error condition, i.e. the fourth parameter when the issue was reported. The method itself has an arbitrary name and always takes the same two parameters: The *issue* containing all information about the issue such as text position, error code etc. and an *issueResolutionAcceptor* to register resolutions to, e.g.

   ```
   @Fix(AndroidJavaValidator.CAPITALIZE)
   public void capitalizeName(final Issue issue,
                              IssueResolutionAcceptor acceptor) {
   ```

10. Within the method you register resolutions to the acceptor, e.g.

   ```
   acceptor.accept(issue,
      "Capitalize name", "Capitalize the name.", "upcase.gif",
      new ISemanticModification() {
         public void apply(EObject o, IModificationContext context) {
   ```

```
            Application application = (Application) o;
            String name = application.getName();
            application.setName(name.substring(0,1).toUpperCase()
                  + name.substring(1));
            }
    });
```

The parameters of the accept method are

- the text to show for this quick fix,
- the longer description of the quickfix,
- the name of the icon,
- the modification as an *ISemanticModification*.

11. Write a quick fix that removes an element and register it to the two validation issues you implemented before.
    Hint: To delete an EObject call

    ```
    EcoreUtil.remove(object)
    ```

12. Restart the editor and watch your quick fixes in action.