

Creating a Domain Specific Language (DSL) with Xtext

Arjan Mooij, Jozef Hooman

V5.2, 2017-03-05 – Eclipse Neon

1 Overview

This report explains how a Domain Specific Language (DSL) can be defined and used by means of Eclipse Neon and the Xtext plug-ins (version 2.10). Section 2 describes the steps needed to install Eclipse and all required plug-ins. The creation of a new DSL is explained in Section 3. A number of miscellaneous topics are described in Section 4. Background information and various small topics can be found in Section 5. Section 6 contains a number of useful commands and shortcuts.

We concentrate on textual DSLs with Xtext. To connect text and graphical notations we refer to Sirius (<https://eclipse.org/sirius/>). An example of the combination of Xtext and Sirius is the Eclipse IDE for POOSL, see <http://poosl.esi.nl/>.

2 Install Eclipse and the required plug-ins

1. When not yet available, install Java Runtime Environment (JRE), see <http://www.oracle.com/technetwork/java/javase/downloads>
This manual has been tested with Java SE Runtime Environment 8.
2. Install *Eclipse IDE for Java Developers* from <https://eclipse.org/downloads/eclipse-packages/>.
The current version is called Eclipse Neon. Locate the “eclipse.exe” file in directory “eclipse” which should be used to start Eclipse. It is convenient to create a shortcut on the desktop or pin it to the task bar. Start Eclipse and add a few plugins using
Help > Install new software > Work with (pull-down menu): select
Neon - <http://download.eclipse.org/releases/neon>
Wait till everything is loaded under “Name”,
 - open “Modelling”, select “Ecore Diagram Editor (SDK)” and “Xtext Complete SDK”
 - open “Programming Languages”, select “Xtend IDE”Click Next (2 times), accept license agreement, and Finish; restart Eclipse when asked.
3. Install the Antlr generator library using Help > Install new software ;
Click the “Add” button after “Work with”, fill in:
 - Name: Itemis Updates
 - Location: <http://download.itemis.de/updates/>Click OK; Choose *Xtext Antlr* ;
Next (twice); accept license agreement; Finish; accept license agreement.
Click OK after warning about unsigned content; Restart Eclipse when asked

Note that if this step is omitted, the user is asked to download the ANTLR parser generator manually, as explained in Section 4.8.

4. If there are multiple JREs on the machine, there are a few ways to specify the JRE to be used. We mention two possibilities (only one of them is needed):

- a. Open the "eclipse.ini" file which is at the root directory of the Eclipse installation. Before the line which contains "-vmargs" add the following two lines where the second one mentions the location of the JRE to be used:

```
-vm
C:/Program Files/Java/jre8/bin/javaw.exe
```

This should lead to a eclipse.ini file which contains :

```
...
--launcher.appendVmargs
-vm
C:/Program Files/Java/jre8/bin/javaw.exe
-vmargs
```

- b. Right-click on Eclipse shortcut, select Properties, tab Shortcut; edit the "Target" field to specify the location of the Java Virtual Machine. For instance, replace "C:\eclipse\java-neon\eclipse\eclipse.exe" by:
C:\eclipse\java-neon\eclipse\eclipse.exe -vm "C:\Program Files\Java\jre8\bin\javaw.exe"

Note the use of "javaw" to avoid an additional window.

3 Define and use a Domain Specific Language (DSL)

This section illustrates how a new DSL can be defined and used. An overview of the main ingredients has been depicted in Figure 1. It shows that the DSL infrastructure is constructed in a meta-level workspace, where the syntax of the DSL and the transformations (e.g., to code) are defined. The DSL is used in a runtime workspace, where instances of the DSL can be defined and the results of the transformations (e.g., code) are generated.

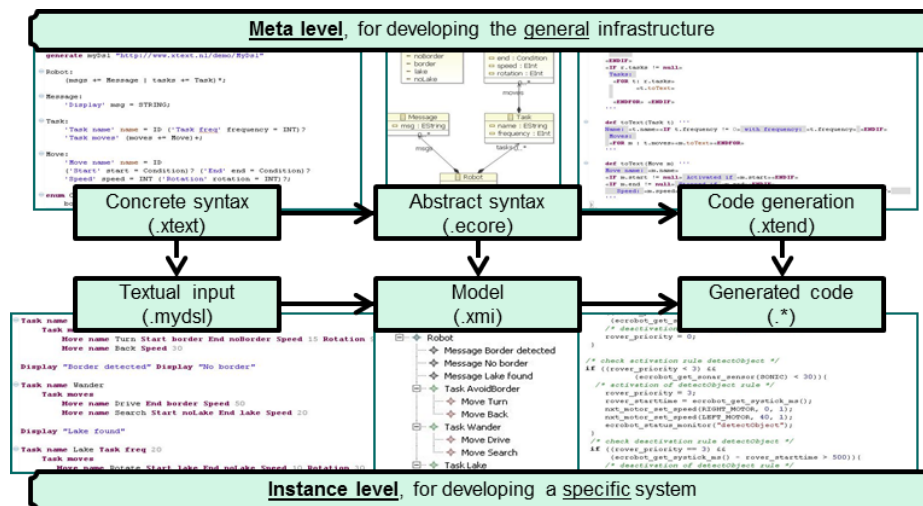
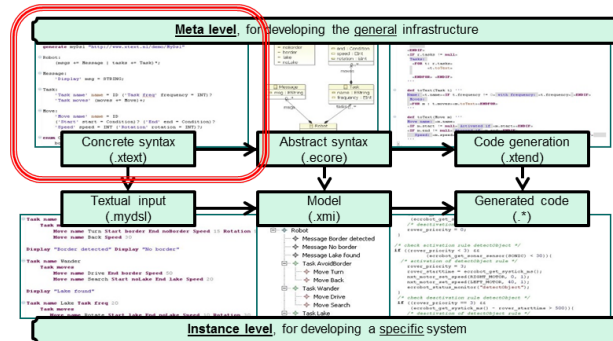


Figure 1 Overview of DSL framework

This section uses a small example to show the details of these ingredients and additional concepts such as validation and scoping.

3.1 DSL definition at the meta-level

This section addresses the definition of syntax of the DSL, as shown in the next figure:



First we create a new workspace:

- Create a new directory “TasksDSL” with an empty subdirectory “TaskDSLMeta” and open Eclipse with workspace “TaskDSLMeta”.
- Eclipse shows the Welcome screen; remove the tick in the bottom right corner before “Always show Welcome at start up” to avoid it will be shown every time. Close the Welcome screen (use Help > Welcome to get it back).

3.1.1 Create Xtext project

- Create a new Xtext project:
 - File > New > Project > select: Xtext, choose: Xtext Project, click Next
 - Change the default values of the following fields:
 - Project name: persons.tasks
 - (Language) Name: persons.tasks.TaskDSL (note that last word should start with a capital following java conventions)
 - (Language) Extension: tdsl
 - Finish (use default location and layout) [it takes some time to create a new project]
- Leads to five projects:
 - persons.tasks: Contains the grammar definition and all related components such as parser, lexer, linker, validation, etc.
 - persons.tasks.ide: Platform-independent IDE functionality.
 - persons.tasks.tests: Here unit tests can be placed.
 - persons.tasks.ui: Contains the language-specific text editor and other workbench-related functionality. In this project advanced functionality can be implemented, such as content assist, outline tree, and quick fix functionality.
 - persons.tasks.ui.test: Unit tests for the Eclipse editor.
- See persons.tasks, folder src, package persons.tasks:
 - GenerateTaskDSL.mwe2: some kind of build file
 - TaskDSL.xtext: defines the grammar, contains some default example initially, as shown in Figure 2.

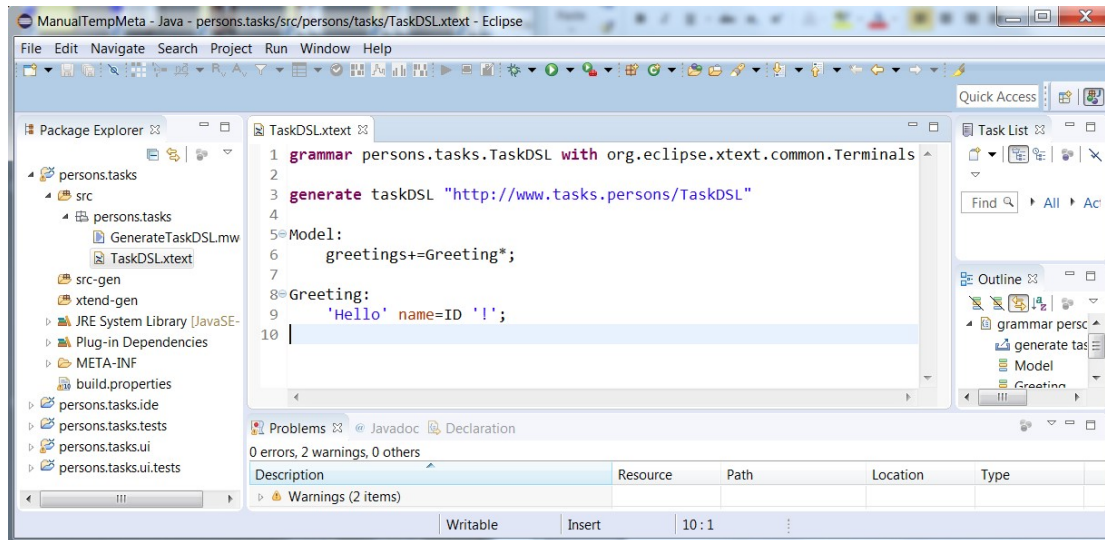


Figure 2 Initial Xtext file

The project structure of the previous step is very suitable for version management, see Section 4.6 for a brief overview of two popular SVN clients. There is no need to archive the large .metadata folder.

3.1.2 Construct grammar of DSL

Change TaskDSL.xtext in *persons.tasks/src/persons.tasks* to the grammar shown below:

```
grammar persons.tasks.TaskDSL with org.eclipse.xtext.common.Terminals
```

```
generate taskDSL "http://www.tasks.persons/TaskDSL"
```

```
Planning: personlist += PersonTasks*
;
PersonTasks: 'Person' name=ID tasks += Task+
;
Task: 'Task' action = Action
      'priority:' prio = INT
      ('duration:' dl = INT unit = TimeUnit)?
;
Action: LunchAction | MeetingAction | PaperAction | PaymentAction
;
LunchAction: 'Lunch' location = ID
;
MeetingAction: 'Meeting' topic = STRING
;
PaperAction: 'Report' report = ID
;
PaymentAction: 'Pay' amount = INT 'euro'
;
enum TimeUnit:
    MINUTE = 'min' |
    HOUR = 'hour' |
    DAY = 'day' |
    WEEK = 'week'
;
```

This defines a grammar to describe tasks of persons by means of a number of production rules:

- The first production rule determines the start nonterminal of the grammar. In this case, rule Planning expresses that the grammar consists of a number of PersonTasks; the “*” indicates zero or more occurrences. The results of the PersonTasks is assigned to feature “personlist”; the notation “+=” indicates a list of elements. In this case the list can be empty.
- The rule for PersonTasks indicates that it starts with terminal string ‘Person’, next feature “name” is defined as an ID which is a predefined terminal, as shown below. Finally, a number of Tasks is assigned to feature “tasks”; the notation “+” indicates one or more elements, so the “tasks” list is not empty.
- A Task starts with terminal string ‘Task’, next an Action is assigned to feature “action”, as indicated by the “=” notation. Similarly, a priority is defined as an INT, which is a non-negative number. Finally, a duration is defined by an INT and a TimeUnit. The duration part is optional, as indicated by the “?” notation.
- An Action is defined as the choice (indicated by “|”) between four types of actions. Note that these four actions are defined using predefined terminals.
- Finally, a TimeUnit is defined as an enumeration, consisting of four cases. Each contains a notation for the meta model on the left in capitals and a terminal on the right (between quotes) for the concrete syntax. As a shorthand, the following can be used:

```
enum TimeUnit:
    min |
    hour |
    day |
    week
```

In general, we prefer the more explicit version.

- The use of an enumeration type for an optional feature is a bit tricky; when the option is not present, the feature will *not* obtain the value “null”, but the default value, which is the first element of the enumeration. In this case, “unit” will obtain the value “MINUTE”. In Section 3.4.3 we will show how this can be improved.

Predefined Xtext terminals [2]:

```
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: .;
```

Note: do not use spaces in terminals.

Generate parser, serializer, and other infrastructure code:

- Right-click in the editor of file TaskDSL.xtext and select Run As -> Generate Xtext Artifacts to generate the language infrastructure.

This leads to a number of message on the console, with as “Done” at the end, see Figure 3.

```
<terminated> GenerateTaskDSL.mwe2 [Mwe2 Launch] C:\Program Files\Java\jre1.8.0_73\bin\javaw.exe (19 Jul. 2016 14:09:42)
375 [main] INFO clipse.emf.mwe.utils.GenModelHelper - Registered GenModel 'http://www.eclipse.org/xtext/xbase/Xtype'
386 [main] INFO clipse.emf.mwe.utils.GenModelHelper - Registered GenModel 'http://www.eclipse.org/xtext/xbase/Xbase'
386 [main] INFO clipse.emf.mwe.utils.GenModelHelper - Registered GenModel 'http://www.eclipse.org/xtext/common/JavaVM
765 [main] INFO text.xtext.generator.XtextGenerator - Generating persons.tasks.TaskDSL
2095 [main] INFO nerator.ecore.EMFGeneratorFragment2 - Generating EMF model code
2130 [main] INFO clipse.emf.mwe.utils.GenModelHelper - Registered GenModel 'http://www.tasks.persons/TaskDSL' from 'pl
4072 [main] INFO text.xtext.generator.XtextGenerator - Generating common infrastructure
4119 [main] INFO .emf.mwe2.runtime.workflow.Workflow - Done.
```

Figure 3 Finished generation of infrastructure

Ecore model:

- The ecore model TaskDSL.ecore, which has been generated from the Xtext grammar, can be found in folder *model/generated* of project persons.tasks. Often the meta model is called the *abstract syntax*, whereas the grammar is the *concrete syntax*.

Optional: View diagram of Xtext grammar:

Open the .xtext file and select: Window > Show View > Other... > Xtext > Xtext Syntax Graph. This leads to a graphical representation of the syntax, which is partly shown in Figure 4.

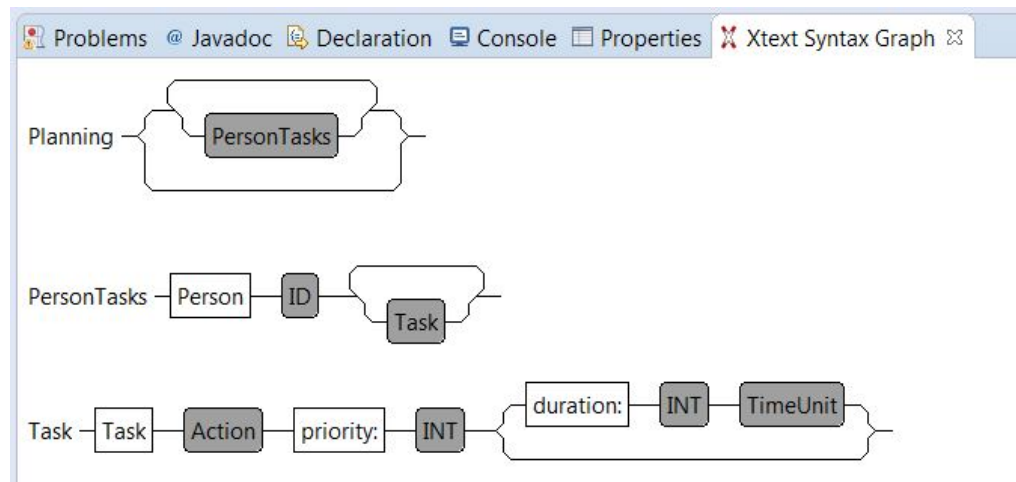
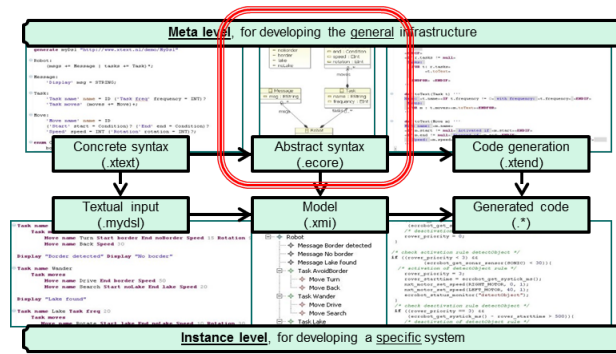


Figure 4 Graphical representation of the grammar

Double-click on an element in the diagram to jump to the corresponding part in the Xtext syntax.

3.1.3 Optional: Ecore diagram

This section describes how the generated ecore model can be visualized. This is not needed for remainder of this manual, but is useful for a good understanding of the ecore model and the model transformations. The next figure shows that this is still at the meta level.



- Right-click on TaskDSL.ecore in persons.tasks/model/generated; select Initialize Ecore Diagram ...; click Finish; in the Create Representation Wizard select Design > Entities; click Next and Finish to select default Representation; click OK to use default name; this opens the taskDSL class diagram
- Right-click inside taskDSL class diagram and select Add Related Elements; select all elements by ctrl-A; Finish
- Click somewhere in the free space of the diagram (to remove the selection). Click Arrange All in upper left corner, see Figure 5. (Click in the diagram if this is not visible.)
- This leads to the diagram of Figure 6. Save the taskDSL class diagram.

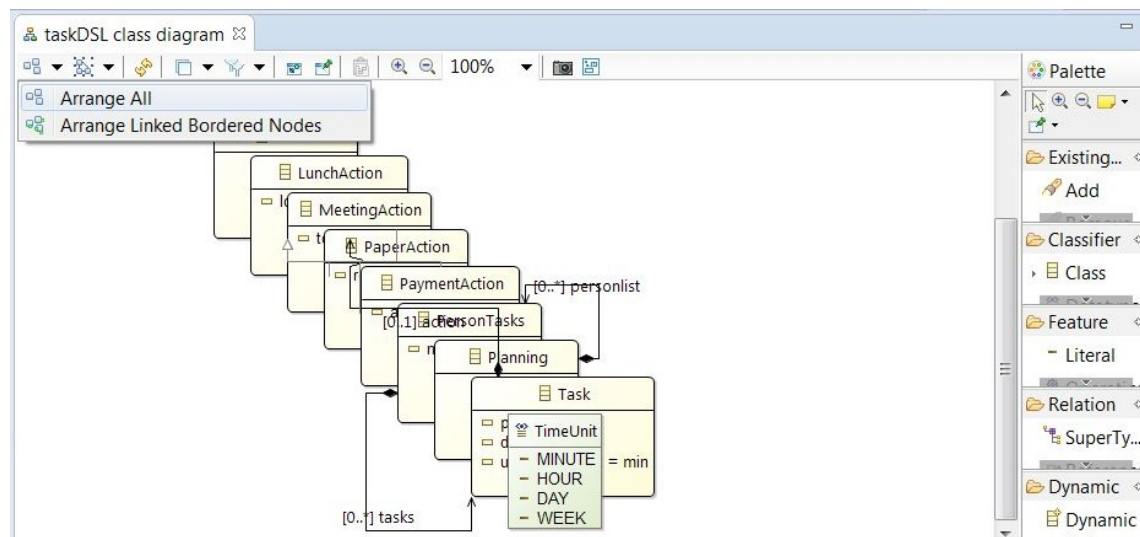


Figure 5 Automatic arrangement of diagram

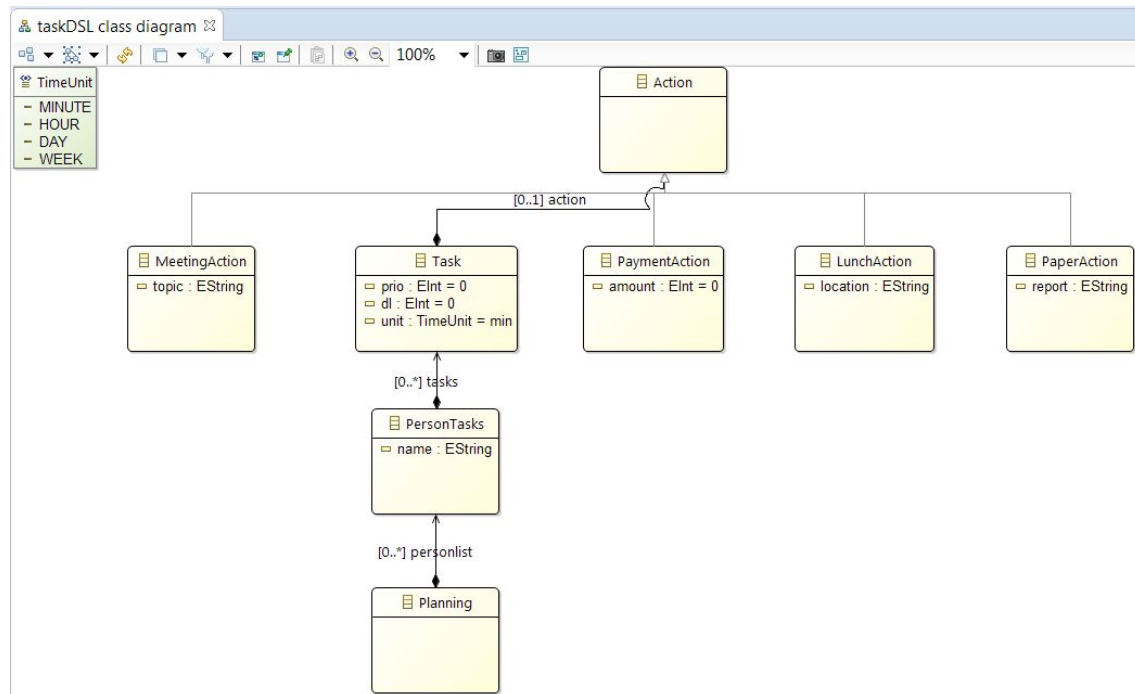


Figure 6 Ecore diagram

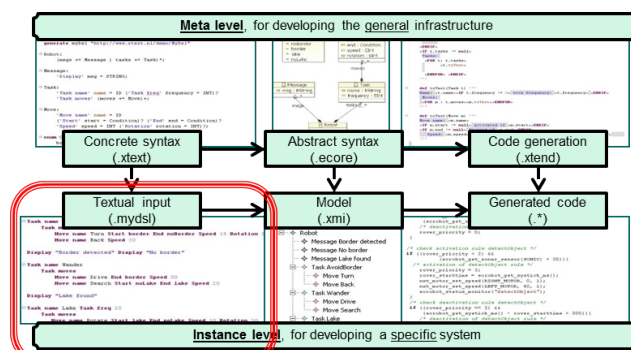
Observe that the cardinalities of the ecore model are based on the assignments in the grammar:

- A basic assignment “=” leads to cardinality [0..1]
- A list assignment “+=” leads to cardinality [0..*]

The iterators “*” and “+” of the grammar are not used, so in our example PersonTasks with zero Tasks is allowed by the ecore model.

3.2 DSL usage in the runtime workspace

Next we consider the so-called runtime workspace where instances of the DSL can be defined.



The runtime workspace can be launched from the meta-level workspace by right-clicking on the project person.tasks in the Package Explorer and select Run As -> Eclipse Application. This leads to a new Eclipse instance with plug-ins for the new language, henceforth called the *runtime workspace*. Observe that the generation of the language infrastructure and the launch of the runtime workspace are available on the down arrow right of the green button; see Figure 7.

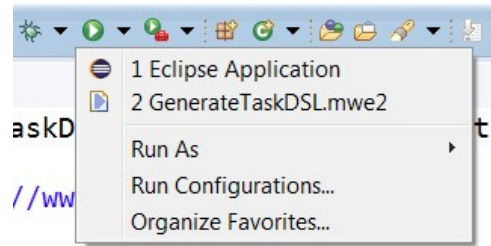


Figure 7 Run button

Close the welcome screen in the runtime workspace and create an instance as follows:

- Create a new project: File > New > Project, select General > Project; Next
 - Give name, e.g.: persons.tasks.models (use default location); Finish
- Create a new folder: New > Folder, parent: persons.tasks.models
 - Give it a name, e.g., Folder name: specs ; Finish
- Create a new file in folder for concrete model – right click on folder:
 - File > New > File
 - Parent: persons.tasks.models/specs
 - File name: spec1.tdsl (use extension “.tdsl” specified before, see Section 3.1.1)
 - Finish; on the question: “Do you want to convert ‘persons.tasks.models’ to an Xtext project?” answer: Yes
- Edit spec1.tdsl, e.g. according to the example below.
 - During editing, “Ctrl-Space” gives possible completions or inserts a keyword if there is no choice.

Person Alice Task Report Strategy priority: 5

Person Bob Task Pay 5000 euro priority: 2

Person Fred Task Lunch Canteen priority: 8 duration: 1 hour

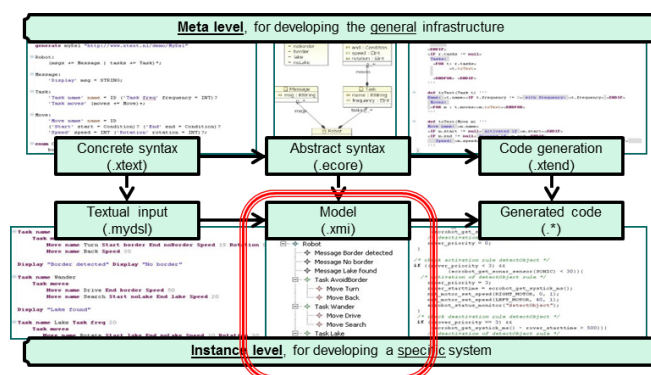
Person Alice Task Meeting "Demo" priority: 4 duration: 90 min

Person Carol Task Meeting "Training" priority: 7 duration: 3 day

Person Dave Task Report Overview priority: 2 duration: 9 week

Person Bob Task Pay 3500 euro priority: 3

Next we explain how to view the DSL instance as an instance of the ecore model, see the next figure:



To view this instance of the ecore model, right click on spec1.tdsl, select “Open With / Other ...”, choose (from Internal editors): Sample Ecore Model Editor. Right click on an element and select Show Properties View to inspect details, such as the value of the prio attribute and the values of the optional duration.

To return to the DSL view, right click on spec1.dsl and select “Open With / TaskDSL editor”.

3.3 Changing the grammar

In this section we illustrate more concepts by making some changes to the grammar. First close the runtime workspace; it always has to be restarted after changes in the grammar.

3.3.1 Using references

As a first step we introduce a separate entity for persons and refer it in the task descriptions. In the previous grammar, we adapt the rules for Planning and Task and replace PersonTasks by Person:

```
Planning: 'Persons:' persons += Person+
        tasks += Task*
;
Person: name=ID
;
Task: 'Task' action = Action
     'person' person = [Person]
     'priority:' prio = INT
     ('duration:' dl = INT unit = TimeUnit)?
;
```

Persons are identified by their name and the feature “person” of a Task *refers* to a person, indicated by the square brackets around the type “[Person]”. Implicitly, this means that the “name” feature of the type is used to refer to it. More explicitly, this can also be written as “[Person | ID]”; in this way also other feature can be used for references. The other parts of the grammar are not changed.

- Regenerate the language infrastructure by right-clicking in the editor of file TaskDSL.xtext and select Run As -> Generate Xtext Artifacts.
- In case an ecore diagram has been generated (Section 3.1.3), answer Yes to the question about removing the TaskDSL.aird file, which is related to the taskDSL class diagram.

3.3.1.1 Optional: view modified ecore diagram

Similar to Section 3.1.3, the new ecore diagram can be inspected. After automatic arrangement and some manual editing, the diagram of Figure 8 can be obtained.

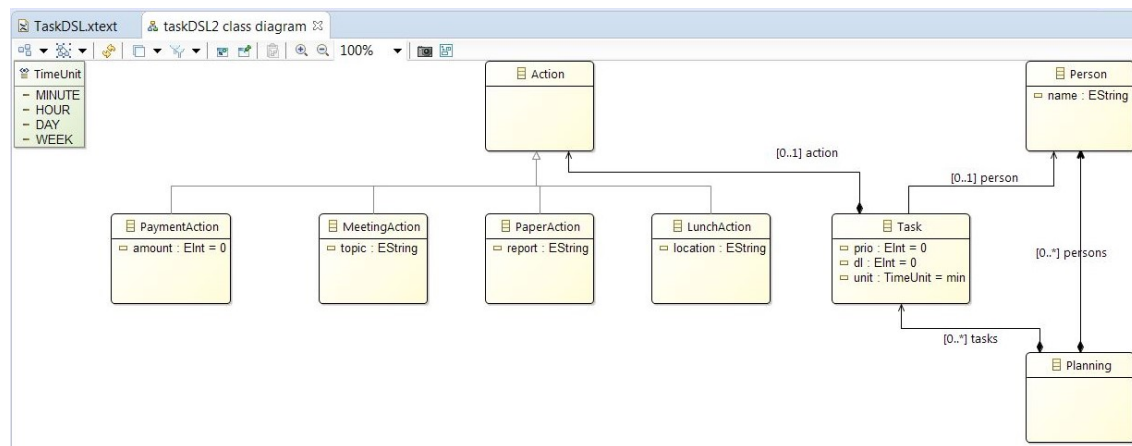


Figure 8 New ecore diagram

3.3.2 Use the new grammar

To use the new grammar:

- Launch the runtime workspace, using the pull-down menu right of the green “run” button shown in Figure 7 and selecting Eclipse Application.

In the runtime workspace, observe that the previous instance “spec1.tdsl” now contains many errors. Create a new file “spec2.tdsl” and define an instance, such as the one below:

```
Persons: Alice Bob Carol Dave Fred
Task Report Strategy person Alice priority: 5
Task Pay 5000 euro person Bob priority: 2
Task Lunch Canteen person Fred priority: 8 duration: 1 hour
Task Meeting "Demo" person Alice priority: 4 duration: 90 min
Task Meeting "Training" person Carol priority: 7 duration: 3 day
Task Report Overview person Dave priority: 2 duration: 9 week
Task Pay 3500 euro person Bob priority: 3
```

When the cursor is placed on a reference, e.g. a person name in a task, then <F3> navigates to the definition.

3.3.3 Allow more freedom in the definition of instances

As a next change, we attach a name to the planning, allow more freedom in the order of defining persons and tasks, and make it possible to have multiple persons related to a task. Close the runtime workspace and change the grammar as follows (showing only the changed part):

```
Planning: 'Planning' name = ID
        (persons += Person | tasks += Task)*
;
Person: 'Person:' name=ID
;
Task: 'Task:' action = Action
      'persons:' persons += [Person]+
      'priority:' prio = INT
      ('duration:' dl = INT unit = TimeUnit)?
;
```

To use the new grammar, regenerate the language infrastructure and launch the runtime workspace if there are no errors. Create an instance, for instance “spec3.tdsl”:

```
Planning DepartmentABC
```

```
Task: Report Strategy persons: Alice Carol priority: 5
Task: Pay 5000 euro persons: Bob priority: 2
```

```
Person: Alice Person: Fred
```

```
Task: Lunch Canteen persons: Fred priority: 8 duration: 1 hour
Task: Meeting "Demo" persons: Alice priority: 4 duration: 90 min
```

```
Person: Bob
```

Task: Meeting "Training" persons: Carol priority: 7 duration: 3 day
 Task: Report Overview persons: Dave priority: 2 duration: 9 week
 Task: Pay 3500 euro persons: Bob priority: 3

Person: Carol Person: Dave

Note:

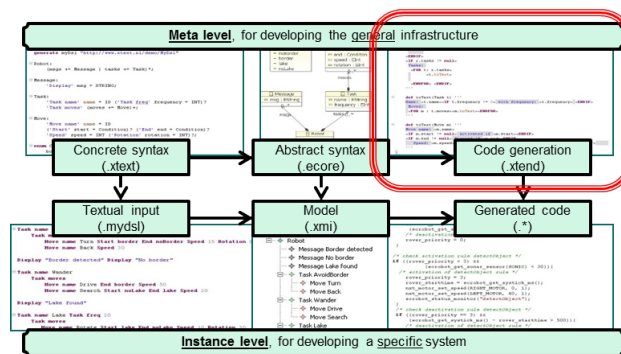
- It is possible to add comments to an instance using `//` and `/* ... */`, see the predefined Xtext terminals `SL_COMMENT` (Single Line Comment) and `ML_COMMENT` (Multi Line Comment) mentioned in Section 3.1.2.

3.4 Model transformation (e.g., to generate code or text) with Xtend

We discuss the techniques to transform models to, for instance, code, visualizations, or documentation. As an example, the models expressed in the DSL are transformed to plain text.

3.4.1 Define the model transformations at the meta level

We start in the meta-level workspace with the definition of a transformation:



- Close the runtime workspace, start working in meta-level workspace `TaskDSLMeta`.
- Open the file `TaskDSLGenerator.xtend` in `persons.tasks/src/persons.tasks.generator`. The generated file is shown in Figure 9. Note that a split screen can be obtained by clicking on a tab and dragging it to the text area; then a vertical or horizontal screen outline will become visible; release the mouse for the desired lay-out. This is, e.g., useful to have the `xtext` file and definition of the transformation next to each other.
- Model transformations can be written using the Xtend language [3], which can be considered a simplified version of the Java programming language. For instance, many type information can be omitted. Xtend can be freely mixed with Java.

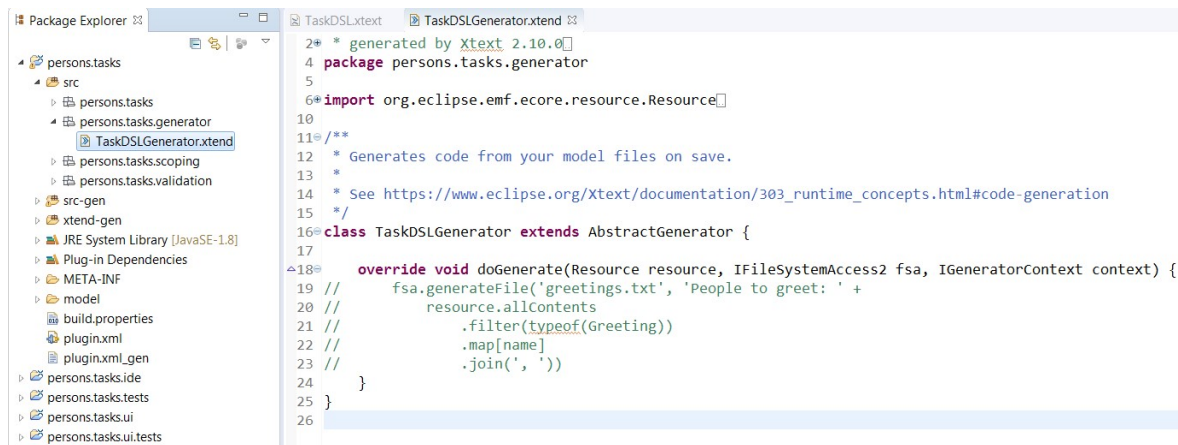


Figure 9 Generated part of model transformation

- Edit the text generation in the “doGenerate” method in file “TaskDSLGenerator.xtend” as follows, defining the root of the transformation and calling a method “toText”, which will be defined below, as follows:

```
class TaskDSLGenerator extends AbstractGenerator {

    override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
                           IGeneratorContext context) {
        val root = resource.allContents.head as Planning;
        if (root != null) {
            fsa.generateFile("PlanningInfo.txt", TextGenerator.toText(root))
        }
    }
}
```

The “root” constant contains the top-level element of the instance of the.ecore model to which the transformation is applied. In our running example, this element is of type “Planning”. Note that keyword “as” is used in Xtend for a type cast. Resolve the error “Planning cannot be resolved to a type” by adding “import persons.tasks.taskDSL.Planning”, e.g., using the quick fix. Note that in Eclipse, a resource is a project, a folder, or a file; here it denotes a model instance. The third parameter of “doGenerate” is not used here; for advanced Xtend users, IGeneratorContext provides access to a CancelIndicator.

Next we define the method “toText”, using a class with auxiliary methods.

Auxiliary methods

It is often convenient to define general methods in a separate class such that they can be used by different transformations. As a basic example, we define a general method in an xtend class.

- Right-click on package “persons.tasks.generator”; select New > Xtend Class; Name: Auxiliary; Finish
- Define class Auxiliary as follows:

```
class Auxiliary {

    def static List<Action> getActions(Planning root) {
        var List<Action> actionlist = new ArrayList<Action>()
```

```

        for (Task t : root.tasks){
            actionlist.add(t.action)
        }
        return actionlist;
    }
}

```

Note that the last definition could also have been written in a very concise way as follows:

```

def static List<Action> getActions(Planning root) {
    return root.tasks.map[t|t.action]
}

```

The definitions above will lead to errors because types cannot be resolved. This can be solved by clicking on the error and selecting a quick fix. A general solution is to organize the imports using <CTRL>-<SHIFT>-O (pressing all three keys simultaneously). In this case this will lead to pop-up windows where a user has to make a choice. Choose in such a way that the following imports are obtained:

```

import java.util.List
import persons.tasks.taskDSL.Action
import java.util.ArrayList
import persons.tasks.taskDSL.Planning
import persons.tasks.taskDSL.Task

```

Note:

- The definitions above are based on the containment hierarchy of classes, going down in this hierarchy using the attributes of classes. It is also possible to go up in this hierarchy using “eContainer” which returns for an EObject the containing object or null if there is no such object. The result has to be casted to the required type, as done in the next example:

```

def static Planning getPersonPlanning(Person person) {
    return person.eContainer() as Planning
}

```

Text generation

Similarly, create in the same package an Xtend file “TextGenerator.xtend” with the following contents - the brackets << ... >> can be obtained by typing <CTRL>-<SPACE> :

```

class TextGenerator {
    def static toText(Planning root)'''
        Info of the planning «root.name»
        All Persons:«"\n"»
        «FOR p : root.persons»«"\t"»«p.name»«"\n"»«ENDFOR»
        All actions of tasks:
        «FOR t : root.tasks BEFORE "===== \n"

SEPARATOR " &" AFTER "====="»
        «action2Text(t.action)»«infoAction(t)»
        «ENDFOR»

        Other way of listing all tasks:
        «FOR a: Auxiliary.getActions(root) SEPARATOR " , " »
        «action2Text(a)»
        «ENDFOR'''

    def static dispatch action2Text(LunchAction action)'''
        Lunch at location «action.location»'''

```

```

def static dispatch action2Text(MeetingAction action)'''
    Meeting with topic «action.topic'''

def static dispatch action2Text(PaperAction action)'''
    Paper for journal «action.report'''

def static dispatch action2Text(PaymentAction action)'''
    Pay «action.amount» euro'''

def static infoAction(Task t)'''
    «IF t.unit != null» with duration: «t.dl» «toText(t.unit)»«ENDIF'''

def static CharSequence toText(TimeUnit u){
    switch(u){
        case TimeUnit::MINUTE: return 'm'
        case TimeUnit::HOUR: return 'h'
        case TimeUnit::DAY: return 'd'
        case TimeUnit::WEEK: return 'w'
    }
}

```

As above, add the missing imports.

Function “toText” is defined using a so-called rich string (the part between the triple quotes “””). Rich strings may contain expressions between angle brackets << ... >>. Inside these angle brackets, one can access attributes of objects and use expressions like FOR and IF. The return type of a rich string is CharSequence.

Note that the “FOR” construct allows a few optional parts which specify what to insert

- BEFORE: before all iterations (if there is at least one iteration)
- SEPARATOR: between all iterations (if there are at least two iterations)
- AFTER: after all iterations (if there is at least one iteration)

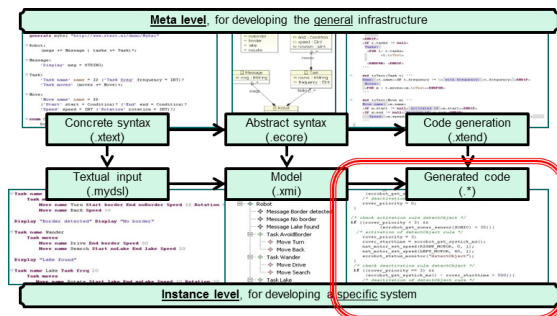
Method “action2Text” is defined for each subtype of type Action in a polymorphic way using the “dispatch” keyword.

For more info about Xtend, see [4] and for instance [5]. Note that a wrong use of the angle brackets << ... >> might lead to strange error messages.

Xtend can be considered as a DSL on top of Java. The Java programs generated from the Xtend code mentioned above can be found in *persons.tasks/xtend-gen/persons.tasks.generator*. Note that this layering might accidentally lead to strange errors in the Xtend editor. After corrections it might be needed to close and reopen the editor or even restart the workspace.

3.4.2 Use the model transformations in the runtime workspace

We investigate the results of the transformation defined in the meta-level workspace at the instance level.



Open the runtime workspace. Since we already have a correct instance (spec3.tdsl), we can generate a text file from it by running from the menu:

- Project > Clean (Clean all projects)

Note that the first time folder *src-gen* is generated. In general, do not place manually written files in this folder. The same holds for other generated folders such as *xtend-gen* and *model/generated*. In this example, *src-gen* contains the generated text file which is partly shown in Figure 10.

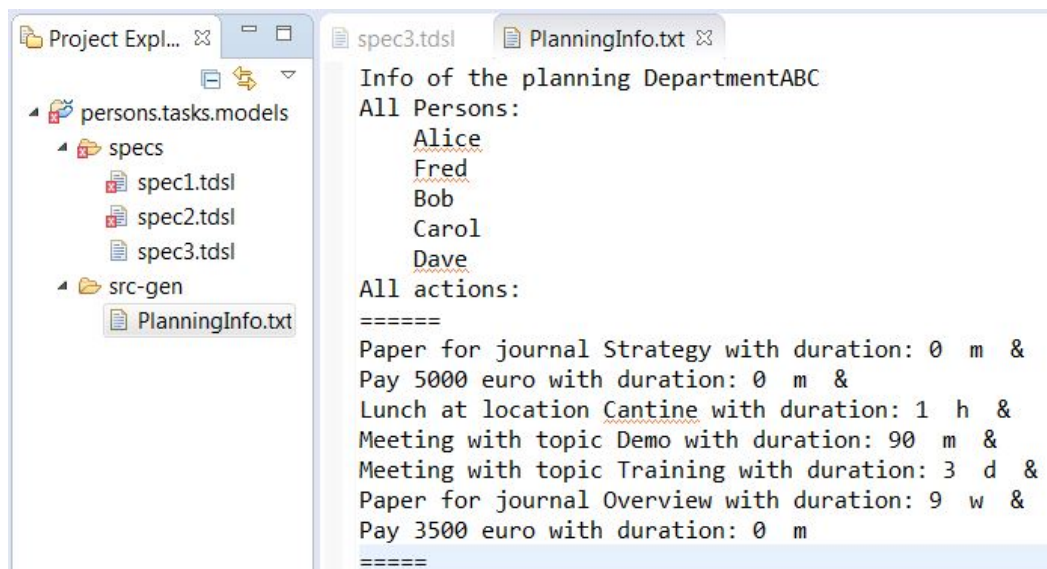


Figure 10 Part of the generated text file

Make a few changes to the instance and observe that the generator is called again when the instance is saved. Define a few more instance; observe that:

- the transformation is only applied to correct instances
- file PlanningInfo.txt corresponds to the last saved instance
- after Project > Clean file PlanningInfo.txt corresponds to the last instance

When the generation takes a considerable amount of time, it might not be convenient that it is executed automatically after every save (<CTRL>-S). Then the automatic build can be switched off by selecting Project and unchecking "Build automatically" (it can also be unchecked in: Window > Preferences > General > Workspace). Then the generation can be started using the options under Project or using <CTRL>-B.

Finally observe that the translation of the optional duration part of a task is not correct. If this part is not present in the instance, the translation shows the default values 0 (for INT) and “m” (the first element of the enum). This will be solved in the next section.

3.4.3 Improved model transformations

Close the runtime workspace, start working in meta-level workspace TaskDSLMeta. We will improve and extend the generator by correcting the problem with the optional part, add a html generator, and put the generated files in separate directories.

Reference to optional feature

To allow a convenient check on optional features, we add an indirection by referring to a type. In this example, we add a type Duration and refer to it in the optional feature. Hence, modify file TaskDSL.xtext as follows:

```
Task: 'Task:' action = Action
      'persons:' persons += [Person] +
      'priority:' prio = INT
      ('duration:' duration = Duration)?
;
Duration:
    dl = INT unit = TimeUnit
;
```

Regenerate the language infrastructure. This will lead to an error in TextGenerator.xtend. In this file, rewrite infoAction(Task t) as follows:

```
def static infoAction(Task t)'''
    «IF t.duration != null» with duration: «t.duration.dl»
    «toText(t.duration.unit)»«ENDIF'''
```

Note: if the Xtend file reports an error on a missing “duration” feature, regenerate the language infrastructure again or even restart the workspace.

Html generation

Create a new Xtend class “HtmlGenerator.xtend” in package persons.tasks.generator, as described before. Insert the following:

```
class HtmlGenerator {
    def static toHtml(Planning root)'''
        <html>
        <head>
        <style>
            table, th, td
            { border:1px solid black;
              padding:5px;}
            table
            { border-spacing:15px; }
        </style>
        </head>
        <body>
            <H1>Planning: «root.name»</H1>
            «ListTasks(root)»
        </body>
```

```

</html>'''

def static listTasks(Planning root)'''
    <table style="width:300px">
    <tr>
    <th>Persons</th>
    <th>Action</th>
    <th>Priority</th>
    </tr>
    «FOR t: root.tasks»
    <tr>
    <td>«ListPersons(t)»</td>
    <td>«action2Html(t.action)»</td>
    <td>«t.prio»</td>
    </tr>
    «ENDFOR»
    </table>'''

def static listPersons(Task task)'''
    «FOR p : task.persons SEPARATOR " , " AFTER " "» «p.name» «ENDFOR'''

def static dispatch action2Html(LunchAction action)'''
    Lunch at «action.location»'''
def static dispatch action2Html(MeetingAction action)'''
    Meeting'''
def static dispatch action2Html(PaperAction action)'''
    Paper for «action.report»'''
def static dispatch action2Html(PaymentAction action)'''
    Pay'''
}

```

Import the missing types from persons.tasks.taskDSL.

Extend generator

We add the html generator to the TaskDSLGenerator and place the generated files in a directory which depends on the file name of the instance:

```

class TaskDSLGenerator extends AbstractGenerator {

    override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
                           IGeneratorContext context) {
        val root = resource.allContents.head as Planning;
        if (root != null) {
            var path = "generated/" + resource.getURI().lastSegment + "/"
            fsa.generateFile(path+"PlanningInfo.txt", TextGenerator.toText(root))
            fsa.generateFile(path+"PlanningOverview.html",HtmlGenerator.toHtml(root))
        }
    }
}

```

Note that “resource.getURI().lastSegment” yields the last part of the name of the URI of the resource, which is in this case the name of the source file.

3.5 Using debug mode

The debug mode is an alternative way to launch the Runtime workspace which is useful during the try-out of transformations. When the runtime workspace is launched via the run button (see Figure 7), it has to be restarted when the transformations have been changed. While developing the

transformations, it is more convenient to use the debug mode, using the down arrow of the green “bug” icon, left of the green “run” button (see Figure 11) and selecting Eclipse Application.

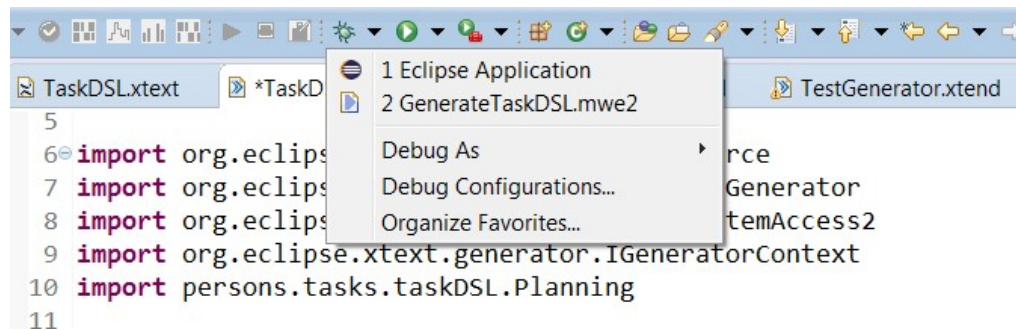


Figure 11 Launch runtime in debug mode

In this mode it is possible to place breakpoints in files of the meta-level workspace, for instance, in TaskDSLGenerator.xtend. To place or remove a breakpoint, right-click on a line and select “Toggle Breakpoint”. When, after a Project > Clean or a change and save in the runtime workspace, a breakpoint is hit, there will be a question at the meta-level workspace whether the Debug perspective has to be opened; answer “yes”. In this perspective, for instance, tracing information and values of variables can be inspected.

In the debug mode, changes in the body of methods are immediately available at the runtime level. This avoids time-consuming restarts of the runtime workspace. This is still needed for larger changes, e.g., when adding or deleting methods, but then the user is warned to restart the runtime workspace.

Experiment with the debug mode on our running example:

- Launch the Runtime workspace in debug mode
- Perform a Project > Clean and observe the results; to view the html files, right-click on the file, select Open With > Web Browser
- Make change in one of the generators at the meta level, save it, and observe the result of a Project > Clean in the runtime workspace. (Refresh open files with <F5>.)

3.6 Boolean assignment operator

The Boolean assignment operator “?=” implies that a feature is of type Boolean. As an example, we extend the Planning rule as follows:

```
Planning: 'Planning' name = ID
    (anonymous ?= 'anonymous')?
    (persons += Person | tasks += Task)*
;
```

Then feature “anonymous” is true if and only if keyword ‘anonymous’ is present in an instance. It can be used, for instance, to check in HtmlGenerator.xtend whether person names have to be shown:

```
def static listTasks(Planning root)'''
    <table style="width:300px">
```

```

</tr>
<th>Persons</th>
<th>Action</th>
<th>Priority</th>
</tr>
«FOR t: root.tasks»
<tr>
<td>«IF (!root.anonymous)»«listPersons(t)»«ENDIF»</td>
<td>«action2Html(t.action)»</td>
<td>«t.prio»</td>
</tr>
«ENDFOR»
</table>'''

```

3.7 Validation

This section explains how to add validation checks to the DSL developed above. Starting point of validation is the file “TaskDSLValidator.xtend” in package `src/persons.tasks.validation`, see Figure 12.

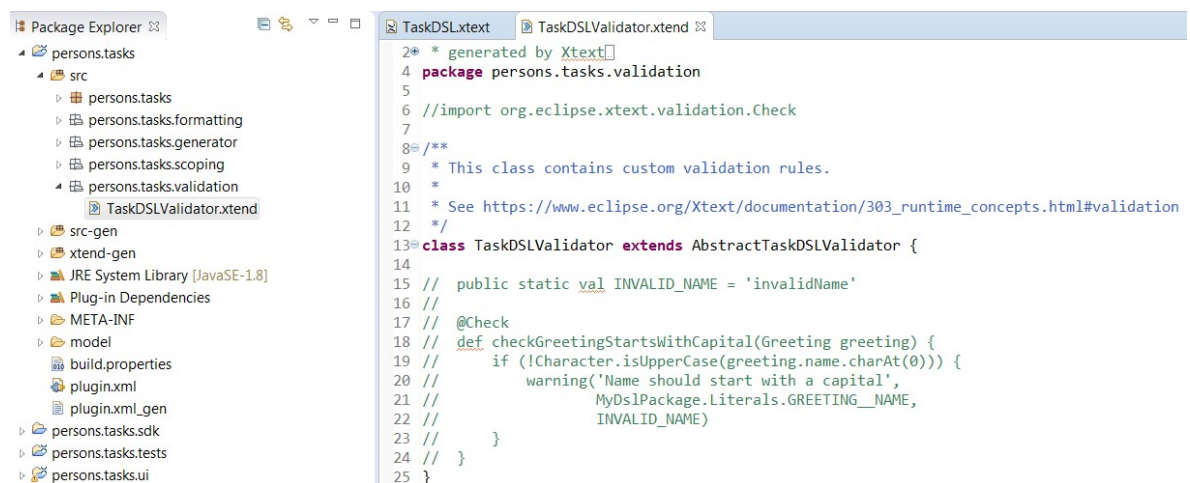


Figure 12 Initial validation file

In Xtend validation methods have the following form:

```

@Check
def anyName(SomeClassFromYourMetaModel instance) {
    ...
}

```

The method name “anyName” is completely free; the @Check takes care that the method is called on every instance of type `SomeClassFromYourMetaModel`.

It is possible to combine this with checks written in Java, which should have the form:

```

@Check
public void anyName(SomeClassFromYourMetaModel instance) {
    ...
}

```

To report [potential] faults to the user, in the body of a validation check three pre-defined methods can be called:

- `info(...)`: leads to an “i” in the margin of the instance editor;
- `warning(...)`: leads to a yellow exclamation mark in the margin and text in the instance is underlined yellow;
- `error(...)`: leads to a red cross in the margin and text in the instance editor is underlined red; this also ensures that no code can be generated.

Example

As an example, modify file “TaskDSLValidator.xtend” as follows:

```
class TaskDSLValidator extends AbstractTaskDSLValidator {

    @Check
    def checkTimeUnit(Task task) {
        if (task.duration != null){
            switch(task.duration.unit){
                case TimeUnit::MINUTE: if (task.duration.dl > 1000)
                    {warning("Rewrite to other unit",null)}
                case TimeUnit::HOUR: null
                case TimeUnit::DAY: if (task.duration.dl > 150)
                    {info("Maybe rewrite to weeks",null)}
                case TimeUnit::WEEK: if (task.duration.dl > 52)
                    {error("Deadline longer than 1 year not
allowed",null)}}
            }
        }
    }

    @Check
    def checkDoublePersons(Planning planning){
        var plist = planning.persons // lists start at position 0
        for (var i= 0; i < plist.size ; i++){
            for (var j = i+1; j < plist.size ; j++){
                if (plist.get(i).name.equals(plist.get(j).name)) {
                    error("Double name",null)
                }
            }
        }
    }
}
```

Repair the errors by adding imports, including `import org.eclipse.xtext.validation.Check`.

Launch the runtime workspace and observe the notifications to the user when changing some numbers in durations and adding a person name twice to the list of persons of a task.

More information

Concerning the parameters of the three notification methods, there are a number of patterns to indicate the location in the instance editor. We list a number of convenient parameters:

- `error("Error message", null)` : the error is reported on the full instance of the type mentioned in the corresponding `@Check` method
- `error("Error message", Literals.SOME_CLASS_FROM_YOUR_META_MODEL__SOME_FIELD)` : the error is reported for the field “someField” of the instance of the type mentioned in the corresponding `@Check` method

- `error("Error message", Literals.SOME_CLASS_FROM_YOUR_META_MODEL__SOME_FIELD, index)` : the error is reported for occurrence “index” of the field “someField” of the instance of the type mentioned in the corresponding `@Check` method (useful if “someField” has a list type)
- `error("Error message", instanceOfSomeEObject, null, -1)` : the error is reported on the full “instanceOfSomeEObject”

Combinations of the above patterns and many other possibilities are also available. All possible parameter combinations can be viewed by placing the cursor on info/warning/error and pressing <F3>.

To obtain fragments such as `Literals.SOME_CLASS_FROM_YOUR_META_MODEL__SOME_FIELD`:

- Add an import, depending on the language used:
 - In an Xtend file: add an import of the form: “...<LanguageName>Package\$Literals”; in our example: “**import** persons.tasks.TaskDSL.TaskDSLPackage\$Literals”
 - In a Java file: add an import of the form: “...<LanguageName>Package.Literals”; in our example: “**import** persons.tasks.TaskDSL.TaskDSLPackage.Literals”
- On the place where the literal is needed, type “Literals.” and use content assist (<CTRL>-<SPACE>) to select one of the options of type `EReference`.

In our example, Literals can be found in `src-gen/ persons.tasks.TaskDSL /TaskDSLPackage.java/`

Since validation takes place while the user is typing a concrete model instance, it is important to take into account that most types may have the null value (this does not hold for lists). For these types, always test on null values in the validation checks to avoid exceptions on the console. Also cyclic relations (e.g., a parent relation) may give problems (call stack overflow), so it is good practice to be robust w.r.t. cyclic relations.

3.8 Expressions

We illustrate the definition of expressions by adding integer expressions and Boolean expressions to our running example. For an explanation, see [2] and [6].

Integer expressions

In file “TaskDSL.xtext”, change the `PaymentAction` to

```
PaymentAction: 'Pay' amount = IntExpression 'euro'
;
```

In the same file, define `IntExpression` as follows:

```
IntExpression: ExpressionLevel1
;
ExpressionLevel1 returns IntExpression:           // Left associativity
    ExpressionLevel2
    ( ({ExpressionAddition.left =current} "+"      right=ExpressionLevel2)
      | ({ExpressionSubtraction.left=current} "-"    right=ExpressionLevel2)
    )*
;
ExpressionLevel2 returns IntExpression:           // Left associativity
    ExpressionLevel3
```



```

    ( ({ExpressionMultiply.left=current}      "*"      right=ExpressionLevel3)
    | ({ExpressionDivision.left=current}      "/"      right=ExpressionLevel3)
    | ({ExpressionMaximum.left =current}      "max"    right=ExpressionLevel3)
    | ({ExpressionMinimum.left =current}      "min"    right=ExpressionLevel3)
    | ({ExpressionModulo.left =current}       "mod"    right=ExpressionLevel3)
    )*
;
ExpressionLevel3 returns IntExpression:      // Right associativity
    ExpressionLevel4
    ( ({ExpressionPower.left=current}         "^"
    right=ExpressionLevel3)
    )?
;
ExpressionLevel4 returns IntExpression:
    ExpressionMinus | ExpressionPlus | ExpressionLevel5
;
ExpressionMinus:
    "-" sub=ExpressionLevel5
;
ExpressionPlus:
    "+" sub=ExpressionLevel5
;
ExpressionLevel5 returns IntExpression:
    ExpressionBracket |
    ExpressionConstantInt |
    ExpressionBalance
;
ExpressionBracket:
    "(" sub=IntExpression ")"
;
ExpressionConstantInt:
    value=INT
;
ExpressionBalance:
    value=Balance
;
enum Balance:
    BALANCE = 'balance'
;

```

Regenerate the language infrastructure. Extend the file TextGenerator.xtend with methods for integer expressions:

```

def static dispatch CharSequence generateExpression(ExpressionAddition expr)
    ''' («generateExpression(expr.left)» + «generateExpression(expr.right)» )'''

def static dispatch CharSequence generateExpression(ExpressionSubtraction expr)
    ''' («generateExpression(expr.left)» - «generateExpression(expr.right)» )'''

def static dispatch CharSequence generateExpression(ExpressionMultiply expr)
    ''' («generateExpression(expr.left)» * «generateExpression(expr.right)» )'''

def static dispatch CharSequence generateExpression(ExpressionDivision expr)
    ''' («generateExpression(expr.left)» / «generateExpression(expr.right)» )'''

def static dispatch CharSequence generateExpression(ExpressionMaximum expr)
    ''' max («generateExpression(expr.left)» , «generateExpression(expr.right)» )'''

```

```

def static dispatch CharSequence generateExpression(ExpressionMinimum expr)
    '''min(«generateExpression(expr.left)», «generateExpression(expr.right)»)'''

def static dispatch CharSequence generateExpression(ExpressionModulo expr)
    '''(fmod(«generateExpression(expr.left)», «generateExpression(expr.right)») >=
0 ? fmod(«generateExpression(expr.left)», «generateExpression(expr.right)») :
fmod(«generateExpression(expr.left)», «generateExpression(expr.right)») +
«generateExpression(expr.right)»)'''

def static dispatch CharSequence generateExpression(ExpressionPower expr)
    '''pow(«generateExpression(expr.left)», «generateExpression(expr.right)»)'''

def static dispatch CharSequence generateExpression(ExpressionMinus expr)
    '''(-«generateExpression(expr.sub)»)'''

def static dispatch CharSequence generateExpression(ExpressionPlus expr)
    '''(+«generateExpression(expr.sub)»)'''

def static dispatch CharSequence generateExpression(ExpressionBracket expr)
    '''(«generateExpression(expr.sub)»)'''

def static dispatch CharSequence generateExpression(ExpressionConstantInt expr)
    '''«expr.value»'''

def static dispatch CharSequence generateExpression(ExpressionBalance expr)
    '''«expr.value»'''

```

Moreover, change `action2Text(PaymentAction action)` to

```

def static dispatch action2Text(PaymentAction action)'''
    Pay «generateExpression(action.amount)» euro'''

```

Note the feature “amount” has not been used in the html generator, so currently there is no need to extend the html generator.

Remove the errors by organizing the imports (o <CTRL>-<SHIFT>-O). Launch the runtime workspace and experiment with the use of expressions.

Boolean expressions

To illustrate Boolean expressions, we add an optional “If” clause to the payment action. Change the `PaymentAction` in file “TaskDSL.xtext” to

```

PaymentAction:      ('If' condition = BooleanExpression)?
                    'Pay' amount = IntExpression 'euro'
;

```

In the same file, define `BooleanExpression` as follows:

```

BooleanExpression: BooleanExpressionLevel1
;
BooleanExpressionLevel1 returns BooleanExpression:           // Left associativity
    BooleanExpressionLevel2
    ( {ExpressionBinOp.left=current} bop = BinaryBooleanOperator
right=BooleanExpressionLevel2
    )*

```

```

;
enum BinaryBooleanOperator:
    AND = 'AND' |
    OR  = 'OR'
;
BooleanExpressionLevel2 returns BooleanExpression:
    NotExpression |
    BooleanExpressionLevel3
;

NotExpression:
    "NOT" sub = BooleanExpressionLevel3
;

BooleanExpressionLevel3 returns BooleanExpression:
    (=> ComparisonExpression) |
    BooleanExpressionBracket |
    BooleanExpressionConstant
;

ComparisonExpression returns BooleanExpression: {ExpressionCompOp}
    left=IntExpression op = CompareOperator right = IntExpression
;
enum CompareOperator:
    EQ = '==' |
    NEQ = '!=' |
    GEQ = '>=' |
    G = '>' |
    LEQ = '<=' |
    L = '<'
;
BooleanExpressionBracket:
    "(" sub=BooleanExpression ")"
;

BooleanExpressionConstant:
    value=BOOL_LITERAL
;
terminal BOOL_LITERAL returns ecore::EBoolean: 'true' | 'false' | 'TRUE' | 'FALSE'
;

```

Add an import as follows:

```

generate taskDSL "http://www.tasks.persons/TaskDSL"

// next needed because of Ecore type EBoolean used
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

```

Note: if this leads to a warning, restart Eclipse.

Observe that “BooleanExpressionLevel3” contains the syntactic predicate “=>” which instructs the parser to try to match ComparisonExpression first, before trying BooleanExpressionBracket. For more information, see [8], [9], and [10]. This is needed here to avoid errors during language generation about an ambiguity with brackets. An alternative is to use other brackets, such as “[...]” in BooleanExpressionBracket.

Regenerate the language infrastructure. Extend the file TextGenerator.xtend with methods for Boolean expressions:

```
def static dispatch CharSequence generateExpression(ExpressionBinOp expr)
    ''' («generateExpression(expr.left)» «genBinOp(expr.bop)»
    «generateExpression(expr.right)» )'''

def static CharSequence genBinOp(BinaryBooleanOperator op){
    switch(op){
        case BinaryBooleanOperator::AND: return '''AND'''
        case BinaryBooleanOperator::OR: return '''OR'''
    }
}

def static dispatch CharSequence generateExpression(ExpressionCompOp expr)
    ''' («generateExpression(expr.left)» «genCompOp(expr.op)»
    «generateExpression(expr.right)» )'''

def static CharSequence genCompOp(CompareOperator op){
    switch(op){
        case CompareOperator::EQ: return '''='''
        case CompareOperator::NEQ: return '''!='''
        case CompareOperator::GEQ: return '''>='''
        case CompareOperator::G: return '''>'''
        case CompareOperator::LEQ: return '''<='''
        case CompareOperator::L: return '''<'''
    }
}

def static dispatch CharSequence generateExpression(BooleanExpressionBracket expr)
    ''' («generateExpression(expr.sub)» )'''
def static dispatch CharSequence generateExpression(NotExpression expr)
    '''NOT («generateExpression(expr.sub)» )'''
def static dispatch CharSequence generateExpression(BooleanExpressionConstant
expr)
    '''«expr.value»''' // use literal
```

Moreover, change the action2Text method for a PaymentAction as follows:

```
def static dispatch action2Text(PaymentAction action)'''
    «IF action.condition != null»if «generateExpression(action.condition)» then
«ENDIF»Pay «generateExpression(action.amount)» euro'''
```

More information

We list three patterns to define expressions, depending on the desired associativity.

Left Associativity (left-to-right evaluation): $a \ \&\& \ b \ \&\& \ c = (a \ \&\& \ b) \ \&\& \ c$

ExpressionLevel1 returns Expression:

```
ExpressionLevel2
( ({ExpressionAnd.left=current}          "&&" right=ExpressionLevel2)
| ({ExpressionOr.left=current}           "||" right=ExpressionLevel2)
)*
;
```

Right Associativity (right-to-left evaluation): $a \ \&\& \ b \ \&\& \ c = a \ \&\& \ (b \ \&\& \ c)$

ExpressionLevel1 returns Expression:

```

    ExpressionLevel2
    ( ({ExpressionAnd.left=current}           "&&"   right=ExpressionLevel1)
      | ({ExpressionOr.left=current}          "||"    right=ExpressionLevel1)
    )?
;

```

No Associativity: a && b && c is not allowed

ExpressionLevel1 returns Expression:

```

    ExpressionLevel2
    ( ({ExpressionAnd.left=current}           "&&"   right=ExpressionLevel2)
      | ({ExpressionOr.left=current}          "||"    right=ExpressionLevel2)
    )?
;

```

3.9 Scoping

To investigate scoping issues, change the grammar to

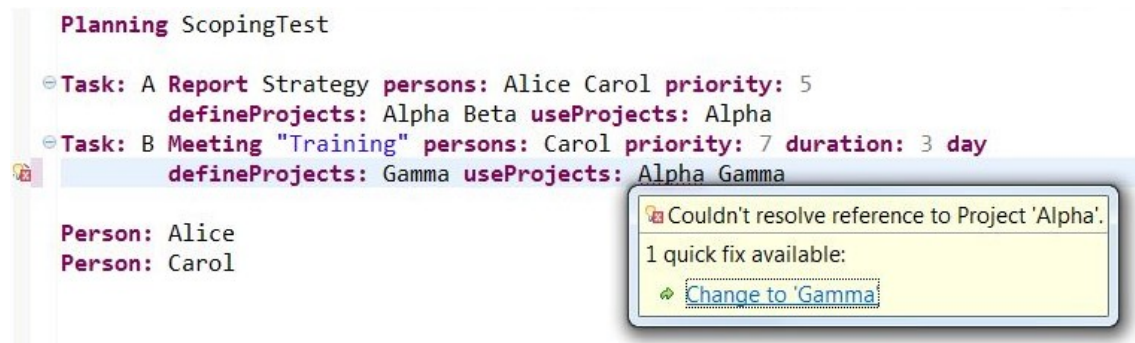
```

Task: 'Task:' name = ID action = Action // added name
      'persons:' persons += [Person] +
      'priority:' prio = INT
      ('duration:' duration = Duration)?
      'defineProjects:' (projects += Project)*
      'useProjects:' (use += ProjectUse)*
;
Project: name = ID
;
ProjectUse: project = [Project]
;

```

Regenerate the language infrastructure and open the runtime workspace.

Then the runtime shows a problem if we try to use a project name defined in another task:



Note that the problem does not appear if the name feature of Task is removed.

If we want to keep the task name, a solution is to extend the scoping rules, see also [11].

First step is to add the possibility for a task to refer to another task. As an example, we add the optional “extends” feature to make explicit to which other task(s) a task may refer:

```

Task: 'Task:' name = ID action = Action // added name
      'persons:' persons += [Person] +
      'priority:' prio = INT

```

```

('duration:' duration = Duration)?
'defineProjects:' (projects += Project)*
'useProjects:' (use += ProjectUse)*
("extends" extends=[Task|ID])?
;

```

Close the runtime workspace. Regenerate the language infrastructure.

Next we extend the scoping rules (which now only resolve local references) in the meta-level workspace. Open file TaskDSLScopeProvider.xtend in persons.tasks/src/persons.tasks.scoping/ and edit it as follows:

```

package persons.tasks.scoping

import java.util.Collection
import java.util.LinkedList
import org.eclipse.emf.ecore.EObject
import org.eclipse.emf.ecore.EReference
import org.eclipse.xtext.scoping.IScope
import org.eclipse.xtext.scoping.Scopes
import persons.tasks.taskDSL.ProjectUse
import persons.tasks.taskDSL.Task
import persons.tasks.taskDSL.TaskDSLPackage.Literals

/**
 * This class contains custom scoping description.
 *
 * See
 * https://www.eclipse.org/Xtext/documentation/303\_runtime\_concepts.html#scoping
 * on how and when to use it.
 */
class TaskDSLScopeProvider extends AbstractTaskDSLScopeProvider {
    override getScope(EObject context, EReference reference) {
        if (context instanceof ProjectUse &&
            reference == Literals.PROJECT_USE__PROJECT
        ) {
            return getScopeProjectUseProject(context as ProjectUse);
        }
        return super.getScope(context, reference);
    }

    private def getScopeProjectUseProject(ProjectUse projectUse) {
        var EObject context = projectUse;
        while (!(context == null || context instanceof Task)) {
            context = context.eContainer();
        }
        var runningTask = context as Task;
        var IScope scope = IScope.NULLSCOPE;
        var Collection<Task> visited = new LinkedList<Task>();
        while (runningTask != null && !visited.contains(runningTask)) {
            visited.add(runningTask);
            scope = Scopes.scopeFor(runningTask.getProjects(), scope);
            runningTask = runningTask.getExtends();
        }
        return scope;
    }
}

```

The added implementation of `TaskDSLScopeProvider` computes the scope of projects using the “extends” relation in the DSL. The overridden method “`getScope`” first checks if it concerns an instance of type `ProjectUse` and that it concerns the scope of “project” in the “`ProjectUse`” rule (using the literal `PROJECT_USE__PROJECT`). Then the scope is computed using auxiliary method “`getScopeProjectUseProject`”. Otherwise the default scoping mechanism (`super.getScope`) is used. Method “`getScopeProjectUseProject`” first searches for a containing object of type `Task` higher in the hierarchy (see also the note in Section 3.4.1). Next the “extends” feature is used to extend the scope. Variable “visited” ensures termination if the “extends” relation contains a cycle.

Open the runtime workspace and add an “extends” relation to the second task:

```
Task: B Meeting "Training" persons: Carol priority: 7 duration: 3 day
      defineProjects: Gamma useProjects: Alpha Gamma extends A
```

Note that the error disappeared. Note that a task C which extends B can also use Alpha:

```
Task: C Lunch Snackbar persons: Alice priority: 1
      defineProjects: Delta useProjects: Alpha extends B
```

With these definitions, task B cannot use Delta.

Alternative solution:

In the solution above the “extends” feature defines which projects are in the scope of a task. An alternative is to extend the scope with the projects of all tasks in the planning. This can, for instance, be achieved by changing method “`getScopeProjectUseProject`” to:

```
private def getScopeProjectUseProject(ProjectUse projectUse) {
  var EObject context = projectUse;
  while (!(context == null || context instanceof Planning)) {
    context = context.eContainer();
  }
  var runningPlanning = context as Planning;
  var IScope scope = IScope.NULLSCOPE;
  for (Task t : runningPlanning.tasks){
    scope = Scopes.scopeFor(t.getProjects(), scope)
  }
  return scope;
}
```

Add `import persons.tasks.taskDSL.Planning` to resolve the error. Next observe in the runtime workspace that all projects are in scope:

```
Task: A Report Strategy persons: Alice Carol priority: 5
      defineProjects: Alpha Beta useProjects: Alpha
Task: B Meeting "Training" persons:Carol priority: 7 duration: 3 day
      defineProjects: Gamma useProjects: Alpha Gamma Delta
Task: C Lunch Snackbar persons: Alice priority: 1
      defineProjects: Delta useProjects: Alpha
```

4 Miscellaneous

In the section we collect a number of useful topics. Section 4.1 contains help facilities and tools. A few notes about possible problems can be found in Section 4.2. Support to debug parsing problems

is described in Section 4.4. Importing and exporting is explained in Section 4.5. Section 4.6 discusses version management. Options in the “.mwe2” files are presented in Section 4.7. Section 4.8 contains a few notes about downloading the Antlr parser generator manually. Section 4.9 explains how to repair a broken runtime workspace.

4.1 Useful help facilities and tools

- To compare Ecore metamodels (diff, merge, etc.), there is a convenient tool, see: <http://www.eclipse.org/emf/compare/> The download page contains installation instructions.
- UML-based visualizations can be obtained using PlantUML, see <http://plantuml.sourceforge.net/> and download a plugin from <http://plantuml.sourceforge.net/updatesite/>

4.2 Grammar errors and warnings

A rule such as

```
Planning: Persons*
```

leads to two error messages: “An unassigned rule call is not allowed, when the ‘current’ was already created.” And “The entry rule ‘Planning’ may consume non empty input without object instantiation. Add an action to ensure object creation, e.g. ‘{Planning}’.”. The problem is that in parser rules it is not allowed to call other parser rules without assigning the outcome to something. Note that each rule leads to class in the meta model, and classes have to be linked somehow; in the rule above no link is specified between Planning and the class Person.

Note that

```
Planning: Person
```

is allowed and leads to inheritance.

Observe that removing “name = ID” feature from the Planning rule

```
8 Planning: 'Planning'
9   (persons += Person | tasks += Task)*
10 ;
```

leads to a warning: “The entry rule ‘Planning’ may consume non empty input without object instantiation. Add an action to ensure object creation, e.g. ‘{Planning}’.”

In general, without an explicit object creation action, object creation take place when the first assignment is executed, which may not occur in this example.

The warning disappears indeed by adding the action (or removing the key word):

```
Planning: {Planning} 'Planning'
        (persons += Person | tasks += Task)*
;
```

4.3 Problem with generation

When in the runtime workspace nothing is generated for DSL instances that do not contain errors, it might be possible that by accident “Build Automatically” has been disabled (see also Section 3.4.2). This can easily be repaired by selecting Project > Build Automatically.

4.4 Debugging parsing errors

Behind the scenes, the Xtext grammar is transformed into an ANTLR grammar (see Section 5.3). Within Eclipse, grammatical errors and ambiguities are reported in terms of the ANTLR grammar. To diagnose them, the program ANTLRWorks is useful:

- Download ANTLRWorks 1.5.2 from <http://www.antlr3.org/works/>
- Start ANTLRWorks by double clicking on the downloaded jar file
 - The first time ANTLRWorks is started, it asks to choose a document to create; we do not want to create such a file, but click OK and then Cancel on the New Grammar Wizard.
 - File -> Open... ->
 - Navigate to a folder like MetaLevelWorkspace\org.xtext.example.mydsl\src-gen\org\xtext\example\mydsl\parser\antlr\internal
 - Select InternalMyDsl.g (or DebugInternalMyDsl.g if present, see below)
 - Grammar -> Check grammar
 - This will result in a pop-up with the error; press OK
 - At the left side of the screen, click on any red rule
 - At the bottom of the screen, select Syntax Diagram. At the right side of the Syntax Diagram select both Alternatives 1 and 2.
 - Now the Syntax Diagram graphically displays two possible parsings: one in green, and one in red.

See also <http://blog.dietmar-stoll.de/2011/08/xtext-grammar-visualization.html>

4.5 Import and Export

We describe how to create and import workspaces, e.g., via archive file.

4.5.1 Create archive file

We list the steps to create an archive (zip) file at the meta level:

- File > Export
- Select General > Archive File; Next
- Select All (or choose a selection);
- Browse to specify archive file at appropriate location; Save
- Finish

Similarly, an archive of the runtime workspace can be generated.

4.5.2 Import archive

1. Open Eclipse (e.g., using a new workspace, or add the project to an existing workspace)
2. File > Import > General > Existing Projects into Workspace ; Next
3. Select an archive file: choose an archive file of a workspace
4. Use the default selection of all projects or choose a selection; Finish
5. Wait till "Building workspace" has finished (see indicator at bottom right corner).
Note that the created project may contain various warnings

6. Generate the language infrastructure

If a pop-up complaining about errors appears: Proceed and continue as follows:

7. Restart Eclipse with the same workspace
8. Project > Clean... ; Clean the imported project
9. Wait till "Building workspace" has finished (see indicator at bottom right corner)
10. All errors should be removed

Note: if not all errors are removed (e.g., errors in the "ui" project are reported), remove the ".antlr-generator.....jar" file in the directory of the main project and generate the language infrastructure again. Answer 'y' as in Figure 13.

4.6 Version management

When archiving your Xtext projects, not all files have to be archived. Before archiving, the Xtext projects can be reduced as follows:

- Folder "bin" can be removed
- Folder "src-gen" can be made empty (but should not be removed)
- Folder "xtend-gen" can be made empty (but should not be removed)
- Folder "model/generated" can be made empty (but should not be removed)

Note that the (meta-level and run-time) workspaces (in particular their large .metadata folders) do not need to be archived. The metadata can be reconstructed later on based on the projects alone.

When using a version management system like SVN, it is useful to set for the folders "src-gen", "xtend-gen" and "model/generated" the ignore property such that SVN clients don't give warnings for any changes in these folders.

In the following two subsections we sketch the use of two popular SVN clients: Subclipse

(<http://subclipse.tigris.org>) and Subversive

(<http://www.eclipse.org/subversive/documentation/gettingStarted/aboutSubversive/install.php>).

4.6.1 Subclipse

To use subversion management, install the next plug-in:

Install Subclipse (Help > Install new software > Work with:

http://subclipse.tigris.org/update_1.8.x

Click on Add, give name, for instance, Subclipse.

Wait till everything is loaded under "Name", select "Subclipse".

Click Next (2 times), accept license agreement, and Finish; restart Eclipse when asked.

To check out an existing project from an SVN:

- File > Import > SVN > Checkout Projects from SVN
- If the SVN location is already listed: Use existing repository location, select required part, etc.; otherwise, Create a new repository location and select URL

To update project from existing SVN location:

- right-click on project, select Team > Update to HEAD
- when needed, regenerate files as described in later sections

To create an SVN repository for an existing project:

- Right-click on project, select Team > Share Project ...

To upload project updates to SVN:

- right-click on project, select Team > Commit...

4.6.2 Subversive

To use subversion management, install the Subversive plug-in:

Help > Install new software > Work with: Neon - <http://download.eclipse.org/releases/neon>

Wait till everything is loaded under "Name", select in package "Collaboration" the non-optimal elements starting with "Subversive".

Click Next (2 times), accept license agreement, and Finish; restart Eclipse when asked.

Activate the Subversive perspective by menu item

'Window > Perspective > Open Perspective > Other... -> SVN Repository Exploring'.

When opening this for the first time, select connectors, e.g. SVN Kit, and install the software.

To check out an existing project from an SVN:

- Open "SVN Repository Exploring" perspective
- Right-click in "SVN repositories" -> New... -> Repository location -> ... and follow instructions
- Right-click in "SVN repositories" on the projects to be checked out -> Check Out
- Open another perspective (such as Resource or Java)

To update project from existing SVN location:

- right-click on project, select Team > Update
- when needed, regenerate files as described in later sections

To create an SVN repository for an existing project:

- Right-click on project, select Team > Share Project ...

To upload project updates to SVN:

- right-click on project, select Team > Commit...

An ignore property can be set for certain folders, to avoid that an SVN client keeps sending warnings after changes in these folders. In Subversive this can be done as follows:

- Right-click on the folder, select Team -> Set Property...
- Name (use the drop-down-box): svn:ignore
- Property value: *
- Click O

To ensure Subversive uses these properties, the option "Subversive SVN JDT Ignore Extensions (Optional)" has to be selected when installing Subversive.

4.7 Options in the ".mwe2" file

A number of options can be enabled/disables in the ".mwe2" file which can be found in the "src" folder, in the package where also the ".xtext" file is located. As an example, a validator to check for duplicated exported object has been disabled by default:

```
validator = {
    // composedCheck = "org.eclipse.xtext.validation.NamesAreUniqueValidator"
```

It can be enabled by uncommenting the corresponding line. After a change, regenerate the language infrastructure..

4.8 Downloading parser generator

If the Antlr generator library is not installed (e.g., step 3 of Section 2 is omitted), then the first time when a language infrastructure is generated, the user is asked to download it – see Figure 13.

Answer “y” to the question about download of ANTLR 3 parser generator.

If downloading the file gives problems with the proxy server:

- download the file <http://download.itemis.com/antlr-generator-3.2.0-patch.jar>
- store it as “.antlr-generator-3.2.0-patch.jar” in directory *persons.tasks* of workspace TaskDSLMeta (note the dot before the name).
- Rerun Generate Language Infrastructure

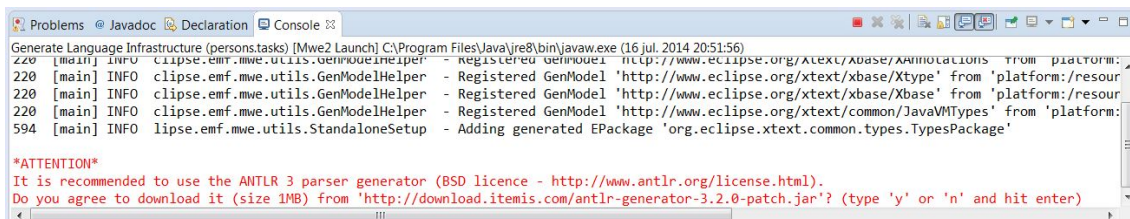


Figure 13 Generate infrastructure

4.9 Repair broken runtime workspace

When starting a runtime workspace it might occur that the progress bar hangs half-way. A few possible causes:

- The runtime workspace has been closed by clicking in the meta-level workspace on the red terminate button in the Console view.
- While using the runtime workspace in debug mode, a pop-up appears mentioning that changes cannot be propagated and choices are offered such as Continue and Terminate. Clicking on Terminate might lead to problems; it is safer to click on Continue and next close the runtime workspace manually.

A solution to this problem:

- Stop the hanging start-up by clicking on the red button in the Console view of meta-level workspace.
- Outside Eclipse, remove the “.metadata” folder in the runtime workspace.
- Start the runtime workspace again; this leads to an empty runtime workspace.
- In the empty runtime workspace, select File->Import...->General->Existing Projects into Workspace
Behind “Select root directory” click on Browse and next on OK to accept the proposed directory.
Under “projects” the existing projects are selected automatically. Finally, click Finish.

5 Background information

For more info about the Eclipse Modeling Framework (EMF) see for instance [7].

5.1 Xtext information

Xtext defines the grammar of the concrete syntax of the DSL language, i.e., the legal sequences of terminal symbols (alphanumeric characters, punctuation marks, and white space characters). These sequences are defined by means of a set of production rules for non-terminal symbols, using Extended Backus-Naur Form-like (EBNF) expressions (e.g., using “|” for the choice between alternatives). There are four different possible cardinalities in these expressions:

- exactly one (the default, no operator)
- one or none (operator ?)
- any (zero or more, operator *)
- one or more (operator +)

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.

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

- The simple equal sign = is the straight forward assignment, and used for features which take only one element.
- The += sign (the add operator) is used for a multi-valued feature (i.e., a list of values) and adds the value on the right hand to that feature. Since “*attribute_name* += ...” is translated in the meta model to a list, it is convenient to use a plural term (e.g., “*values*”) for *attribute_name*. Similarly, it is a good convention to use a singular term (e.g., “*name*”) for the *attribute_name* in grammar fragment “*attribute_name* = ...”.
- The ?= sign (boolean assignment operator) is used for a feature of type Boolean.

Note that multiple assignments to a feature are allowed, for instance:

```
'{ tasks += Task ( ' , tasks += Task ) * ' }
```

5.2 Xtend information

Xtend can be considered a simplified version of the Java programming language. For instance, many type information can be omitted. It also has some additional features such as rich strings which allow for readable string concatenation. They are denoted between triple quotes “” ... “” and may contain expressions between angle brackets << ... >>. The return type of a rich string is CharSequence.

We list a few useful constructs:

- Access an element of an enumeration type: Color::BLACK
- Check whether an optional attribute is set: (attr != null)
- String operations (the full Java API can be used):
 - Concatenation: "hello" + " " + "world"
 - Equality check: "<=".equals(name)
- List operations:

```
«FOR element: someList» Element: «element» «ENDFOR»
«FOR i : (0.. someList.size-1)» Element «i»: «someList.get(i)» «ENDFOR»
```

- Print to console: (for debugging, shown at meta-level console)
println("The value of this variable is now: " + someVariable)

Breakpoints can be set as follows:

- Double click in the grey border at the left side of the statement
- A blue dot will appear in the grey border
- Only effective when running the instance workspace in debug mode

5.3 Parsing

Xtext uses ANTLR (<http://www.antlr.org/>) which is a LL(*) recursive-descent parser generator. See, e.g., http://en.wikipedia.org/wiki/LL_parser. LL parsers do not allow left recursive grammars, see also the Xtext documentation.

Syntactic analysis takes place in three phases:

1. Lexing: a sequence of characters (the text input) is transformed into a sequence of so called tokens (such as ID, INTEGER, REAL, etc.);
2. Parsing: a sequence of tokens is transformed into a parse tree;
3. Linking: the parse tree is converted into a graph based on the cross-references.

These phases are executed sequentially and, hence, earlier phases cannot use knowledge of later phases:

- lexing has no information about the tokens that are expected by the parser;
- parsing cannot use knowledge about possible cross references.

An example of an ambiguity in lexing, consider a grammar contains the following:

- terminal INTEGER returns ecore::EInt: ('0'..'9')+;
- terminal REAL returns ecore::EFloat: ('0'..'9')+ '.' ('0'..'9')+;
- the grammar uses '.' in another rule (which means that this also becomes a token)

This is ambiguous, because input 3.4 can be tokenized in two ways (even if the grammar rules out certain combinations): as one token (REAL) or as three tokens (INTEGER; '.'; INTEGER). Xtext does not give a warning in this case; it will be tokenized as a REAL. More problematic is the input "3.", which will be parsed as a REAL (no backtracking) and hence yields an error.

As an example of an ambiguity in the parsing, consider:

```
Expression: ExpressionConstant | ExpressionVariable;
```

```
ExpressionConstant: variable = [Constant|ID];
```

```
ExpressionVariable: method = [Variable|ID];
```

This is ambiguous because token input ID can be parsed in two ways, even if the declaration of the ID clearly indicates, for instance, that it is a constant (so cross-referencing would exclude one of the possibilities). In general, ambiguity of a grammar is undecidable. If backtracking is not enabled (see

section 4.7), there are limitations on the grammars and often Xtext reports ambiguities after performing “Generate Language Infrastructure”. Since backtracking can have a negative impact on performance and the clarity of error messages, it is advisable to avoid backtracking as much as possible.

6 Commands and shortcuts

6.1 *Useful basic Eclipse shortcuts*

- Switch between tabs:
 - <CTRL>-<Page Up>
 - <CTRL>-<Page Down>
- Maximize current editor (tab):
 - <CTRL>-<M>
 - Double click on tab
- Jump to position of last edit
 - <CTRL>-<Q>
- Save file
 - <CTRL>-S
- Split screen: open two tabs, click on one tab and drag it to the text area; then a vertical or horizontal screen outline will become visible; release the mouse for the desired lay-out.

6.2 *Navigation*

- Outline view:
 - Window -> Show view -> Outline
 - <CTRL>-O opens a quick outline that is searchable (including wildcards)
- Hyper-linking to referenced elements:
 - <F3> while the cursor position is at a cross-reference
 - <CTRL>-click on a cross-reference
- Find references:
 - <CTRL>-<SHIFT>-<G>
- Label (with extra information):
 - Point at a cross-reference

6.3 *Editing*

- Content assist: (including templates for system and data/process/cluster classes)
 - <CTRL>-<SPACE>
- Rename element: (including any cross-references)
 - <ALT>-<SHIFT>-R
- Automatic formatting: (pretty printing)
 - <CTRL>-<SHIFT>-F
- Organize imports:
 - <CTRL>-<SHIFT>-O

6.4 *Fault detection and correction*

- Validation: Automatically upon changes and upon saving
- Revalidate all files:
 - Project -> Clean...
- Validation results:
 - error: icon: cross in a red box
 - warning: icon: exclamation mark in a yellow triangle
- Quick fix (available if a light bulb is displayed next to the validation icon)
 - <CTRL>-<1>

References

- [1] Xtext overview <http://www.eclipse.org/Xtext/>
- [2] Xtext documentation & short tutorials <http://www.eclipse.org/Xtext/documentation/>
- [3] Xtend overview <http://www.eclipse.org/xtend/>
- [4] Xtend documentation <http://www.eclipse.org/xtend/documentation/>
- [5] 20 Facts about Xtend
<http://inario.org/org/inario/inario/documentation/20FactsAboutXtendSpec.html>
- [6] Parsing Expressions with Xtext, Sven Efftinge, <https://typefox.io/parsing-expressions-with-xtext>
- [7] EMF Tutorial, Lars Vogel: <http://www.vogella.com/tutorials/EclipseEMF/article.html>
- [8] How do Syntactic Predicates Work? <http://stackoverflow.com/questions/25341789/how-do-syntactic-predicates-work>
- [9] Using syntactic predicates in Xtext, part 1
<https://dslmeinte.wordpress.com/2011/12/05/using-syntactic-predicates-in-xtext-part-1/>
- [10] Using syntactic predicates in Xtext, part 2
<https://dslmeinte.wordpress.com/2011/12/20/using-syntactic-predicates-in-xtext-part-2/>
- [11] Xtext documentation, Scoping
https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#scoping