

Creating a Domain Specific Language (DSL) with Xtext

Arjan Mooij, Jozef Hooman

V10.0, 2020-10-14 – Eclipse IDE 2020-09 & Xtext 2.23

1 Overview

This report explains how a Domain Specific Language (DSL) can be defined and used by means of Eclipse and the Xtext plug-ins. 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. Hints on solving problems can be found in Section 4. A number of miscellaneous topics are described in Section 5. Background information and various small topics can be found in Section 6. Section 7 contains a number of useful commands and shortcuts.

2 Install Eclipse and the required plug-ins

- Download the Eclipse Installer from <https://www.eclipse.org/downloads/> (press “Download 64 bit”) and execute it. Select “Eclipse IDE for Java and DSL Developers”, choose installation folder and accept licenses.
 - **Note:** the Eclipse Installer includes a JRE (required to run Eclipse). It is also possible to install Eclipse via a package (<https://www.eclipse.org/downloads/packages/>) but this requires manual installation of a JRE and is therefore not recommended.
- Launch Eclipse (use default workspace); Eclipse shows the Welcome screen. Close the Welcome screen (use Help > Welcome to get it back).
- Install a plugin using:
 - Help > Install new software > Work with (pull-down menu): select the plugins of the used Eclipse version, e.g. for Eclipse 2020-09, use:
2020-09 - <http://download.eclipse.org/releases/2020-09>
 - Wait till everything is loaded under “Name”
 - open “Modeling”, select “Ecore Diagram Editor (SDK)”
 - Click Next (2 times), accept the license agreement, and Finish; restart Eclipse when asked.

Note 1: it is possible to use another Eclipse version, i.e. not the DSL version. Then the Xtext and Xtend plugins have to be downloaded in a similar way. For instance, for Eclipse 2020-09 use 2020-09 - <http://download.eclipse.org/releases/2020-09>:

- open “Modeling”, select “Xtext Complete SDK”
- open “Programming Languages”, select “Xtend IDE”

Note 2: the Antlr generator library is usually downloaded automatically during the first build, but if this fails (e.g. because of security settings), it can be installed explicitly using Help > Install new software;

Click the “Add” button after “Work with”, fill in:

Name: Itemis Updates

Location: <http://download.itemis.de/updates/>

Click Add; Choose Xtext Antlr; Next (twice); accept license agreement; Finish; select “Install anyway” in pop-up about unsigned content; Restart Eclipse when asked

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.

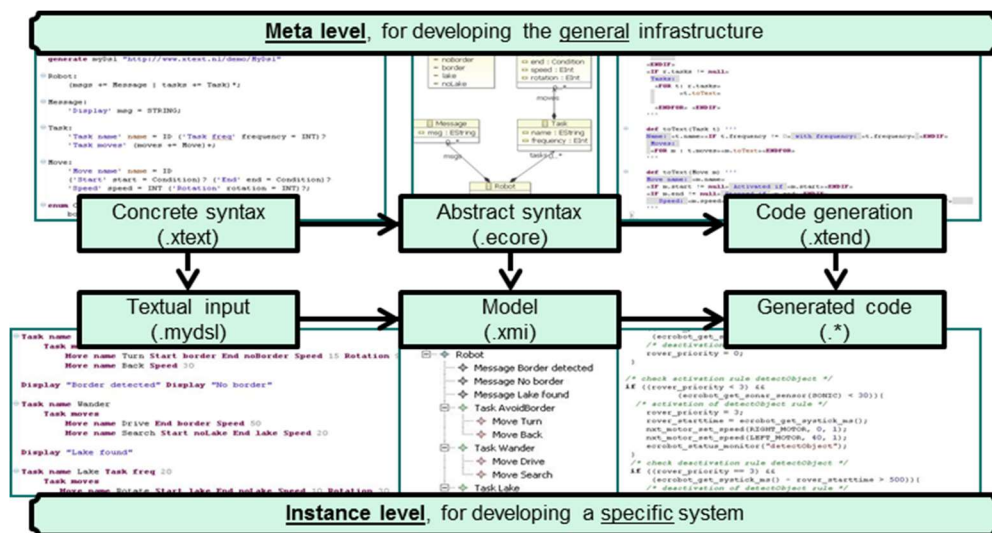
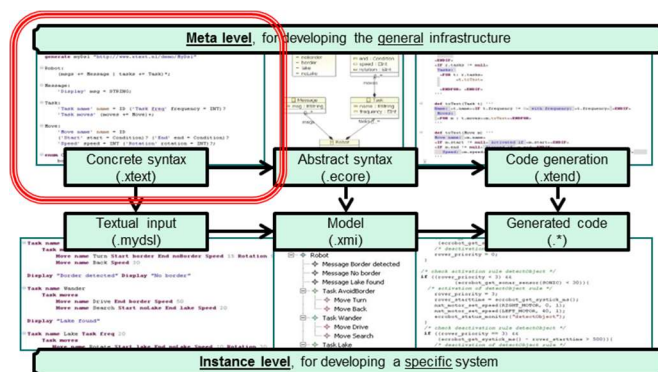


Figure 1 Overview of DSL framework

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 create a new workspace (outside Eclipse):

- Create a new directory “TasksDSL” with an empty subdirectory “TaskDSLMeta” and open Eclipse with workspace “TaskDSLMeta”.

- Switch workspace

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 Extensions: 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.

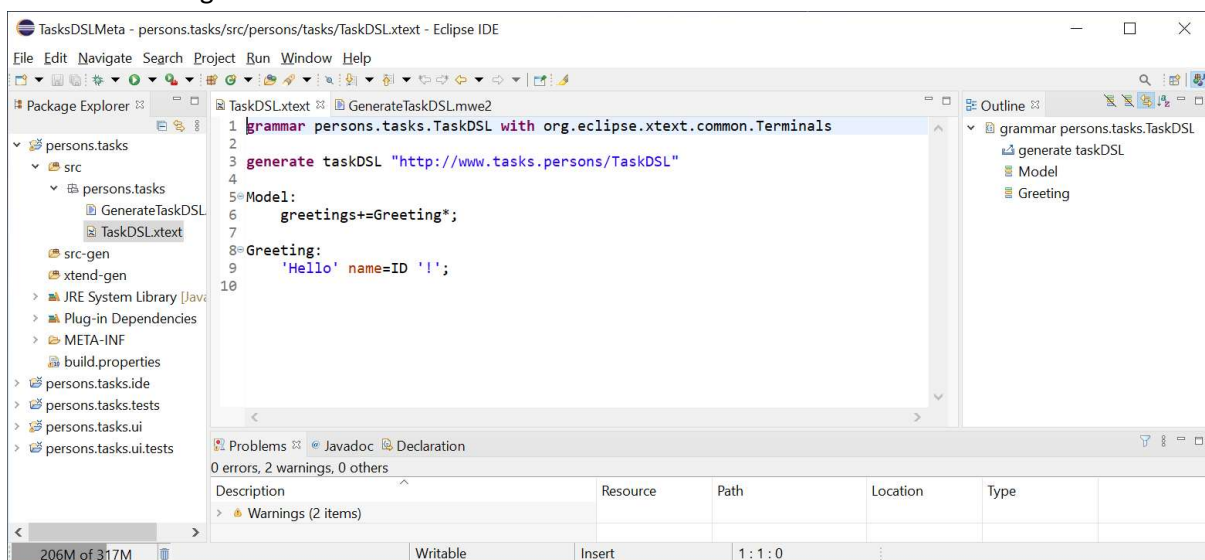


Figure 2 Initial Xtext file

The project structure of the previous step is very suitable for version management, see Section 5.3 for a brief overview on how to use with Git. 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 ID, STRING, and INT.
- 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

```

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-zA-Z_](a-zA-Z_)*';
terminal INT returns.ecore::ElInt: ('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: .;

```

Generate parser, serializer, and other infrastructure code:

- Right-click in the edit window of file TaskDSL.xtext (or on the file on the Package Explorer) and select Run As -> Generate Xtext Artifacts to generate the language infrastructure.

```
<terminated> GenerateTaskDSL.mwe2 [Mwe2 Launch] C:\Users\kanterskhjm\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64.14.0.2.v20200815-640 [main] INFO eclipse.emf.mwe.utils.GenModelHelper - Registered GenModel 'http://www.eclipse.org/xtext/common/JavaVM'
1033 [main] INFO erator.parser.antlr.AntlrToolFacade - Downloading file from 'http://download.itemis.com/antlr-generat
2227 [main] INFO erator.parser.antlr.AntlrToolFacade - Finished downloading.
2246 [main] INFO text.xtext.generator.XtextGenerator - Generating persons.tasks.TaskDSL
3406 [main] INFO nerator.ecore.EMFGeneratorFragment2 - Generating EMF model code
3444 [main] INFO clipse.emf.mwe.utils.GenModelHelper - Registered GenModel 'http://www.tasks.persons/TaskDSL' from 'pl
5795 [main] INFO text.xtext.generator.XtextGenerator - Generating common infrastructure
5826 [main] INFO .emf.mwe2.runtime.workflow.Workflow - Done.
```

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*.

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.

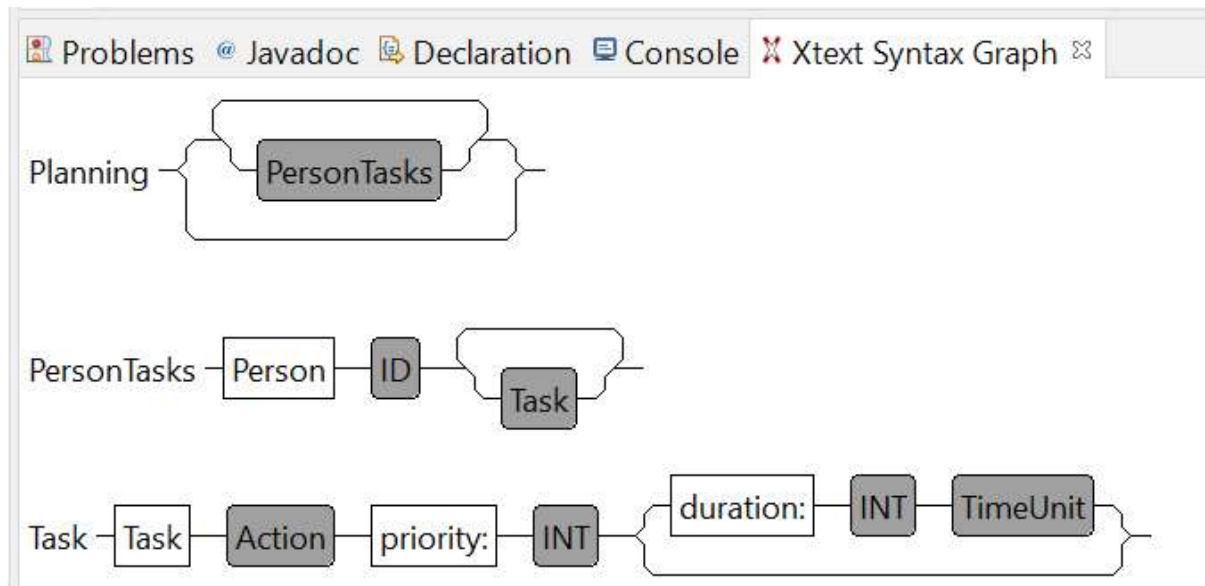
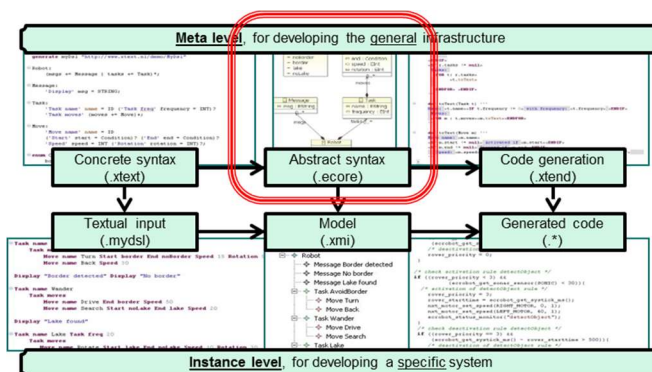


Figure 4 Graphical representation of the grammar

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 Next; in the Create Representation Wizard select Entities in a Class Diagram; click Next and Finish; this opens the taskDSL class diagram
- As mentioned in the diagram, double-click at the indicated location inside the taskDSL class diagram. This leads to the diagram of Figure 5. Save the taskDSL class diagram.

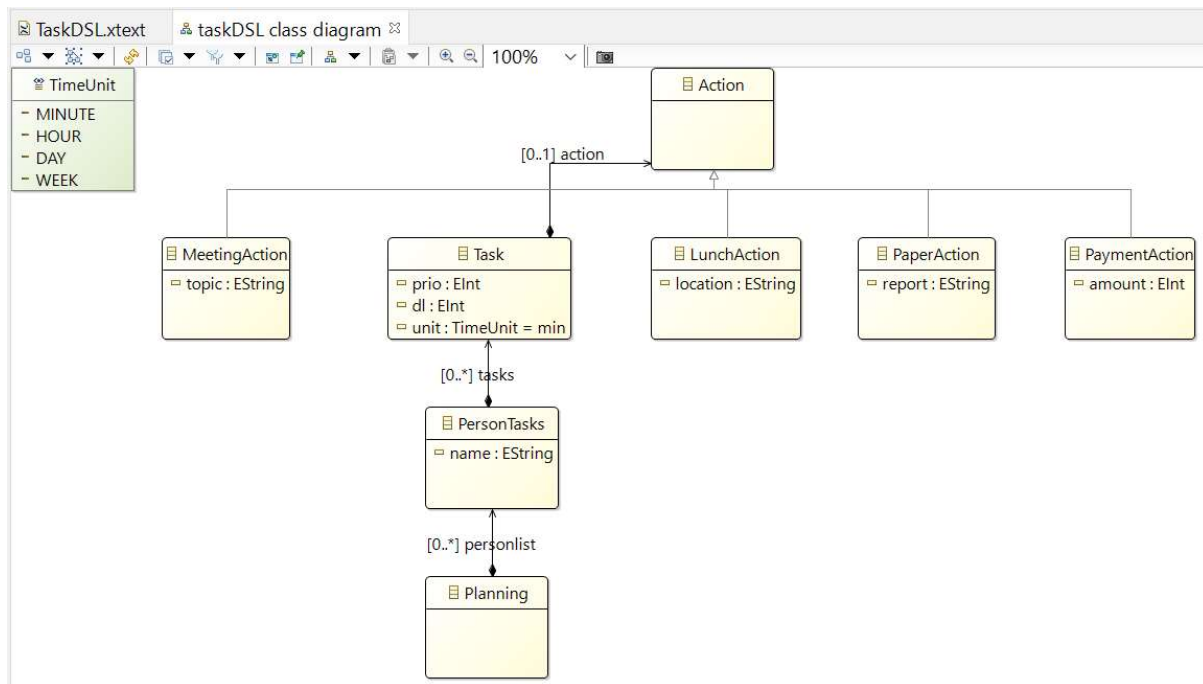


Figure 5 Ecore diagram

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

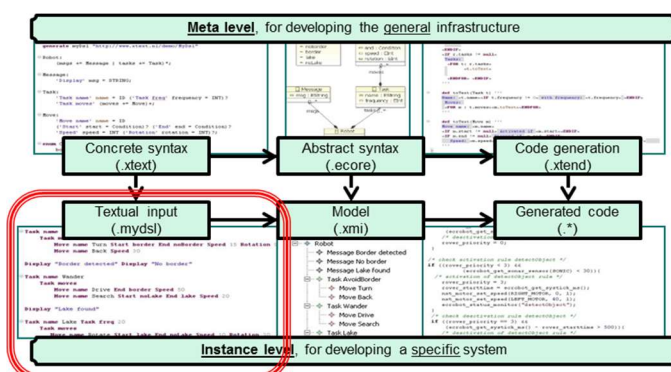
- A basic assignment “=” leads to cardinality [0..1]
- A list assignment “+=” leads to cardinality [0..*]

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

Note that the diagram can be edited to improve the lay-out if needed.

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 6.

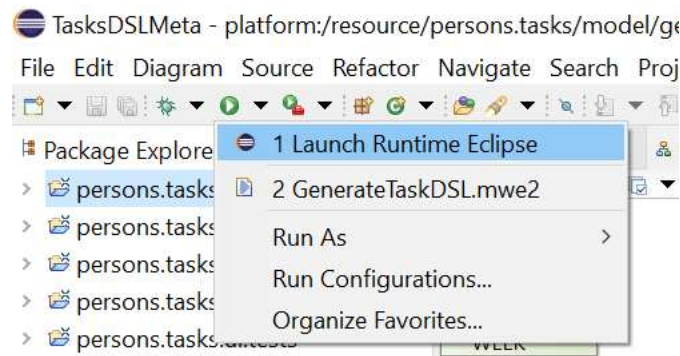


Figure 6 Run button

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: right-click on project:
 - 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:
 - 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

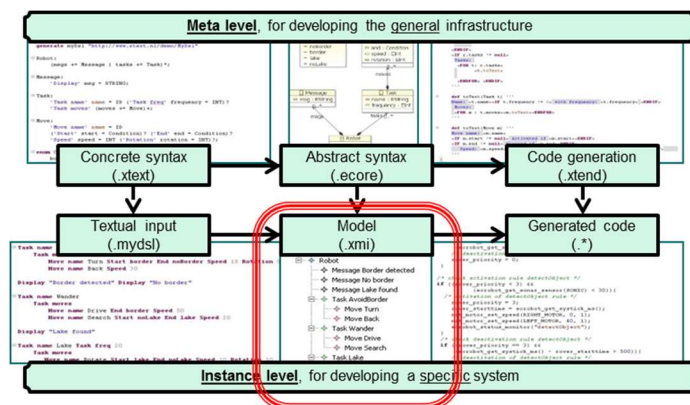
Person Alice Task Meeting "Demo" priority: 4 duration: 90 min

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 file spec1.tdsl, select “Open With / Other ...”, choose (from Internal editors): Sample Ecore Model Editor, click OK. Unfold the model and 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.tdsl 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 to 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 selecting 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 some manual editing, the diagram of Figure 7 can be obtained.

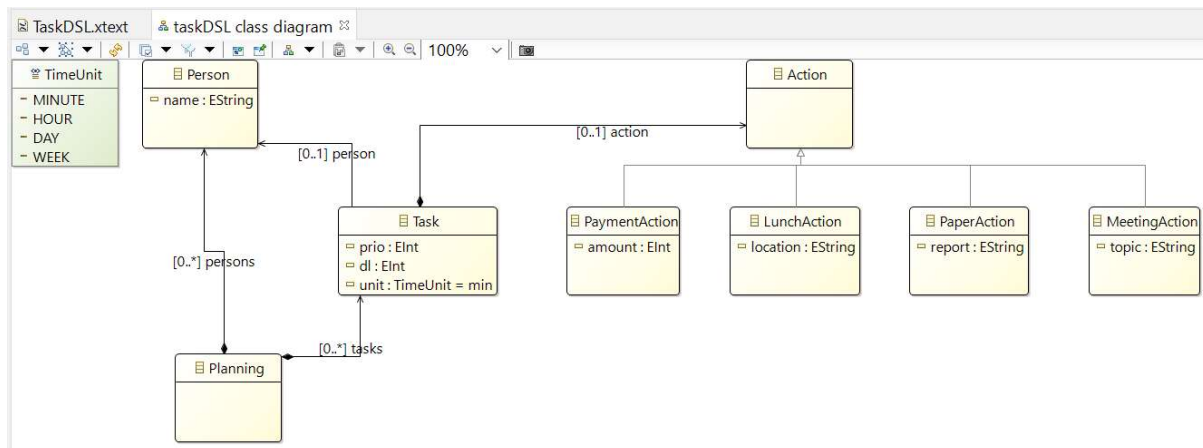


Figure 7 New.ecore diagram

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 6 and selecting Launch Runtime Eclipse.

In the runtime workspace, observe that the previous instance “spec1.tdsl” now contains many errors. If the errors are not visible, perform *Project > Clean* – click Clean. 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

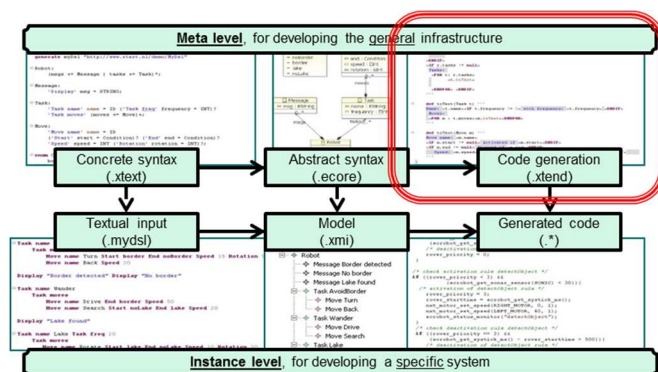
Note that it is possible to add comments to an instance using // and /* */, see the predefined Xtend terminals SL_COMMENT (Single Line Comment) and ML_COMMENT (Multi Line Comment) mentioned in Section 3.1.2.

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 8. 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.

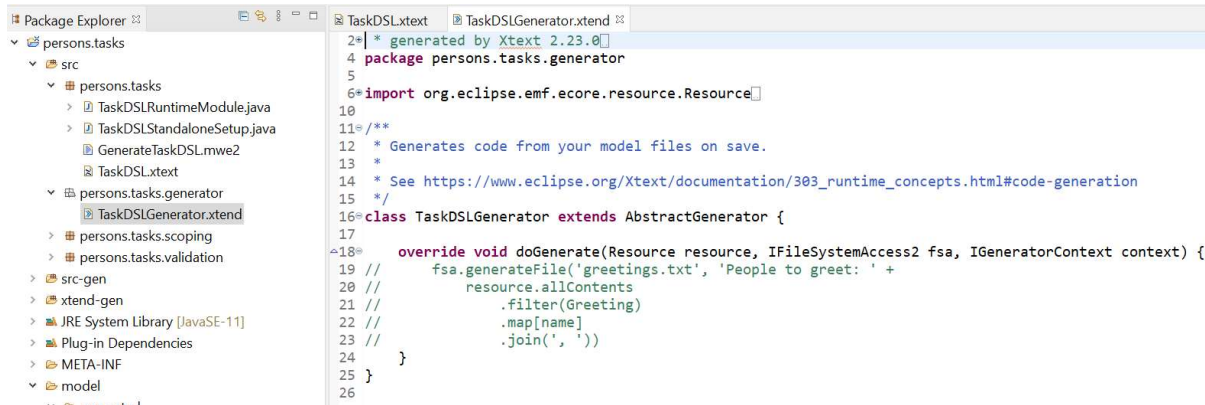


Figure 8 Generated part of model transformation

- 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 for comparisons with “null” it is more efficient to use == and != instead of == and != (this is indicated by a warning for which a quick-fix is available).

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 CancellableIndicator.

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;
    }
}
```

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.ArrayList
import java.util.List
import persons.tasks.taskDSL.Action
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
}
```

Note that the definition of *getActions* 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]
}
```

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.

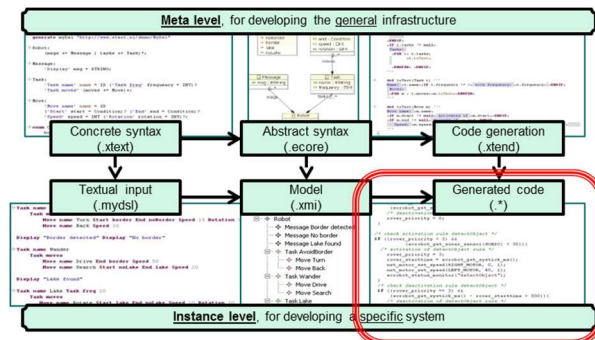
For more info about Xtend, see [4] and for instance [5]. Note that a wrong use of the angle brackets << ... >> might lead to strange error messages.

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.



Launch 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 9.

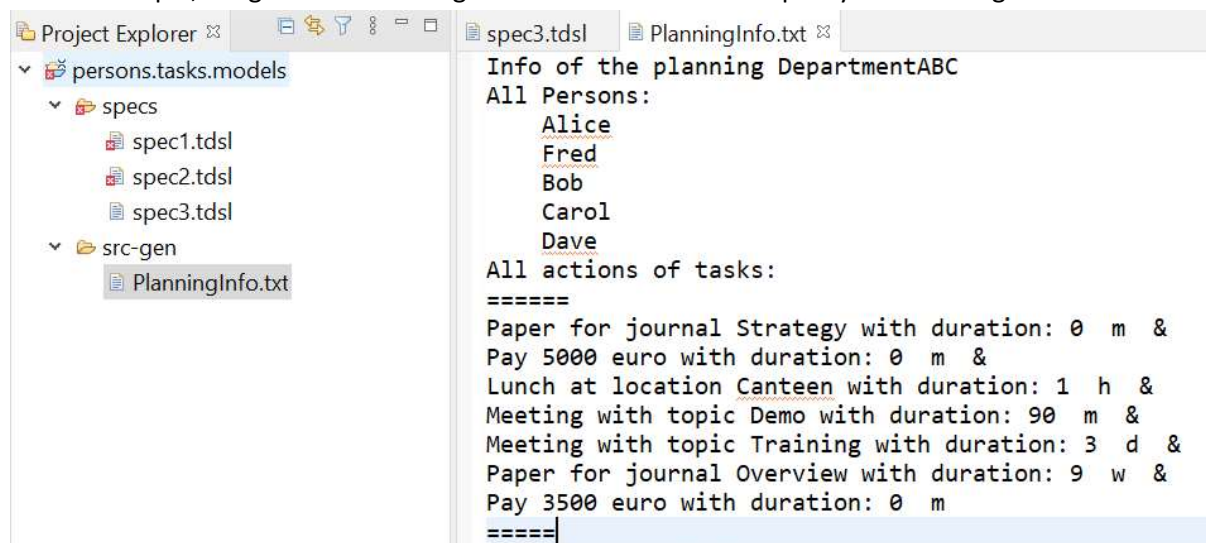


Figure 9 Part of the generated text file

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 an 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 (to empty the cache).

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;}
        </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

6), 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 and selecting Launch Runtime Eclipse.

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 a 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>'''
```



```

        break;
    case DAY:
        if (duration.getDl() > 150)
            info("Maybe rewrite to weeks",null);
        break;
    case WEEK:
        if (duration.getDl() > 52)
            error("Deadline longer than 1 year not
allowed",null);
        break;
    }
}
}

@Check
void checkDoublePersons(Planning planning){
    EList<Person> plist = planning.getPersons(); // lists start at
position 0
    for (int i = 0; i < plist.size() ; i++){
        for (int j = i+1; j < plist.size() ; j++){
            if
(plist.get(i).getName().equals(plist.get(j).getName())) {
                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.

Validation in Xtend

It is also possible

In Xtend validation methods have the following form:

```

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

```

As an example, instead of “TaskDSLValidator.java” one might use the following file “TaskDSLValidator.xtend”:

```

package persons.tasks.validation

import org.eclipse.xtext.validation.Check
import persons.tasks.taskDSL.Planning
import persons.tasks.taskDSL.Task
import persons.tasks.taskDSL.TimeUnit

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)
            }
        }
    }
}
}

```

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 (<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)
)?
;

```


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 Problem solving

This section contains a few hints about solving problems. A general approach is to regenerate the language infrastructure in the meta-level workspace, to perform a Project > Clean in the runtime workspace, or to restart Eclipse. Section 4.1 addresses a problem with the generation. A few notes about possible grammar errors can be found in Section 4.2. Section 4.3 explains how to repair a broken runtime workspace.

4.1 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.

4.2 Grammar errors and warnings

A rule such as

```
Planning: Person* ;
```

leads to an error message: “An unassigned rule call is not allowed, when the 'current' was already created.” and a warning which will be explained later. 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 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 Miscellaneous

In the section we collect a number of useful topics. Section 5.1 contains help facilities and tools. Importing and exporting is explained in Section 5.2. Section 5.3 discusses version management. Options in the “.mwe2” files are presented in Section 5.4.

5.1 Useful help facilities and tools

- To compare Ecore metamodels (diff, merge, etc.), there is a convenient tool, see: <http://www.eclipse.org/emf/compare/> It can be downloaded from the Eclipse Market place via Help > Eclipse Marketplace.
- UML-based visualizations can be obtained using PlantUML, see <http://plantuml.sourceforge.net/> and download a plugin (see <http://plantuml.com/eclipse>) from <http://files.idi.ntnu.no/publish/plantuml/repository>.

5.2 Import and Export

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

5.2.1 Create archive file

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

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

5.2.2 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

5.3 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 “test-bin” can be removed
- Folder “model/generated” 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)

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 Git as version management system the .gitignore file below is recommended. This file must be placed alongside the projects folder (under TaskDSLMeta if you followed Section 3.1)

```

**/src-gen/*
**/xtend-gen/*
**/bin/
**/test-bin/
**/model/generated/
!.gitkeep
.antlr-generator-*-patch.jar
# Below is only needed when Git repo is inside the Eclipse workspace
.metadata/

```

Important: before committing to Git, create an empty file called .gitkeep in the src, src-gen and xtend-gen directories. In this way Git will preserve those empty directories which are required for Eclipse to build the projects. If you do not do this your projects will not build after a fresh checkout of the Git repository.

By default Eclipse creates the Xtext projects (Section 3.1.1) in the Eclipse workspace. If you want to move these projects to an existing Git repository first copy the projects from your Eclipse workspace to your Git repository (in this case copy the projects from the TaskDSLMeta directory). Next select all projects in Eclipse, right mouse click and press “Delete”. Check “Delete projects contents on disk (cannot be undone)” and press “Ok”. Now follow the steps described in Section 5.3.2 to import the projects from an existing Git repository.

A popular Git client which is bundled with Eclipse is EGit. It is also possible to use a generic Git client like the Git command line or Sourcetree. We sketch the use of both in the following subsections.

5.3.1 EGit

To check out an existing project from Git:

- File > Import > Git > Projects from Git -> Next
- If you already have a checkout of the Git repository locally, select “Existing local repository”; otherwise select “Clone URI”.
- Regenerate serializers, parsers and other infrastructure code as described in Section 3.1.2

To retrieve latest changes from Git:

- right-click on project, select Team > Pull
- when needed, regenerate files as described in Section 3.1.2

To create a Git repository for an existing project:

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

To upload project updates to Git:

- right-click on project, select Team > Commit

5.3.2 Generic Git client

When using a generic Git client like the Git command line or Sourcetree, first make a checkout of your Git repository. Load the projects in Eclipse by navigating to File, Import, select “Existing Projects into Workspace”. Check “Select root directory”, press “Browse...” and select your Git repository. The projects will now show up under “Projects:”, make sure all are checked. Next press “Finish” and regenerate serializers, parsers and other infrastructure code as described in Section 3.1.2

5.4 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.

6 Background information

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

6.1 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 ) * '}
```

6.2 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

6.3 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 excludes one of the possibilities). In general, ambiguity of a grammar is undecidable. If backtracking is not enabled (see section 5.4), 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.

7 Commands and shortcuts

7.1 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.

7.2 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

7.3 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

7.4 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://jnario.org/org/jnario/jnario/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