Language Workbench Challenge 2013
Xtext Submission

Version: 1.0 DRAFT - 27. März 2013

Karsten Thoms, Johannes Dicks, Thomas Kutz (itemis)

Abstract

The Language Workbench Challenge 2013 (LWC13) is an initiative created by a group of experts at the CodeGeneration 2010 conference[1]. The aim is to set a common task[2] for Language Workbenches[3] which is implemented with the different existing alternatives in a comparable way.

This document describes in detail how the task is solved with Xtext[4]. Xtext is one of the most well known Language Workbenches and part of the Eclipse Modeling Project[5].

Testimonial

We would like to thank:

1. *Angelo Hulshout* for initiating and organizing the Language Workbench Challenge. It is his work that allows the Language Workbench Challenge to continue now in its 3rd year.
2. *The Xtext Team* is doing a great job on developing a robust, flexible and easy to use Language Workbench.

[1] http://www.codegeneration.net/cg2010/

[2] see http://www.languageworkbenches.net/ for the detailed description of the LWC11 competition and other submissions

[3] http://martinfowler.com/articles/languageWorkbench.html, http://blog.efftinge.de/2007/11/definition-of-term-language-workbench.html

[4] http://www.xtext.org

[5] http://www.eclipse.org/modeling

Document History

Version 0.1 - 2013-01-28

- Initial creation

Table Of Contents

1 Introduction

1.1 Task Description

The LWC13 task is to implement a DSL for questionnaires (Questionnaire Language, QL), which basically allows the definition of forms with questions.

1.2 Technology Stack

This tutorial expects that you are somehow familiar with Java and Eclipse and have heard about EMF and how it works in general before. We start almost at the beginning, but not quite :-)

We will use Xtext 2.4.0, which is at the moment of writing the latest official release. Xtext 2.5 is in preparation and will be released with Eclipse Kepler in June 2013[6]. The solution approach described here would work also with any version of Xtext >= 2.0, but the API might differ slightly, so there is no guarantee that each codeline printed here would work exactly with all versions. For better reproduction it is highly recommended to use the versions mentioned above.

For Code Generation we will use the language Xtend, which itself is based on Xtext. Xtend makes use of a common expression language shipped with Xtext called Xbase. The languages developed here will also be based on Xbase, but more on this later.

The reference implementation of the Xtend generator will generate, JavaServer Faces 2.1(JSF).[7] JSF is part of the Java Enterprise Edition (Java EE). It is useful to have a basic understanding of how web applications work even if JSF provides a nice level of abstraction. The JSF reference implementation from Oracle Mojarra 2.1.6[8] is able to run within the well known Servlet container Apache Tomcat(v7.0).[9]

To get a nicely integrated developement environment we will install some components of the Web Tools Platform(WTP)[10] into an existing Eclipse installation.

[6] http://wiki.eclipse.org/Kepler/Simultaneous_Release_Plan
[7] http://www.javaserverfaces.org/
[8] http://javaserverfaces.java.net/
[9] http://tomcat.apache.org/
[10] http://www.eclipse.org/webtools/

1.3 Xtext Overview

This overview will give you a rough idea about what Xtext[11] is all about. We will then dive into the details and work on a small project.

In a nutshell, Xtext is a workbench to create and work with textual domain-specific languages (DSLs). It comes as a feature (set of plugins) to the popular Eclipse IDE.

The first thing you will want to do is to create your own domain-specific language (DSL) and specify a *grammar* for it. The grammar file is a plain text file with "`.xtext`" filename extension, and the grammar within is defined with a BNF like syntax. While you can use any text editor to modify it, Xtext gives you a specialized editor for grammar files. It is aware of the Xtext language, gives you syntax coloring, code completion, and more. To get a first impression see the screenshot of the Xtext grammar file, opened with the Xtext grammar editor, below. It is not required to fully understand the content yet, this will be discussed in the next chapter in detail.

[11] http://www.xtext.org

The aforementioned example DSL allows you to define entities like “Person”, “Car”, “Book”, and so on. An entity has properties, e.g. a Person has a name, a gender, and a date of birth. A Book has a title, one or more authors, and an ISBN number.

A textual DSL model could look like this, but you could also imagine other syntaxes:

```
entity Person {
  name : String
  gender : m
  birthday : Date
}

entity Book {
  title: String
  authors: Person[]
  isbnNumber: String
}
```

Note that the Property `authors` is of type `Person`, so there can be references between entities. In the Xtext grammar file you specify how you want to define entities and their properties.

Once you have completed your language, you can do that: define some entities, say “Book” and “Person”, together with their respective properties and with proper references between them. The nice thing is that Xtext not only gives you a syntax-driven editor for editing grammar files. Additionally it generates an editor that is specific to the language you have defined. It knows about your language’s keywords and where to place them, it knows about all the syntactical constructs you have made up in your grammar, it includes all the nice stuff like syntax coloring, code completion, validation, and more. For example, if you are at some point where a reference to another entity must be inserted, your DSL editor shows you all the references that would be valid here – according to your language rules – and lets you choose among them. All in all, using the DSL editor generated by Xtext, it is quite easy to establish a text file that adhers to your DSL.

Depending on your language’s type, you could call this text file e.g. a model, a document, a program, or whatever. We will refer to DSL files here as *models* (files).

Consider now that you have created a model. What can you do with it? A typical requirement is to generate an implementation of it in a language like Java, C++, or XML. Or a graphical representation. Or something quite different. This is where code generation comes in. Xtext creates a skeleton code generator for you. Typically you use that code generator as a starting point to produce e.g. Java source code, documentation in, say, DocBook or Wiki format, over-

view graphics using GraphViz, or any other stuff you need. Xtext offers special support for textual output formats, but it is also possible to generate binaries.

This was only a short outline of some prominent Xtext aspects. It is by far not everything Xtext can do for you, but it should suffice for now. The next chapters will show you in more detail how to work with Xtext.

1.4 Installing Eclipse and Xtext

Xtext is a SDK for the Eclipse IDE. To install it you have two options:

- You can download Xtext separately and install it in your Eclipse instance.
- You can download a specially-crafted complete Eclipse distribution which has Xtext pre-packaged already.

We will take the latter approach here and describe the individual steps:

1. Go to the Xtext download page. Here you can get Eclipse 4.2.x (Juno) including Xtext 2.3.1 along with some tools Xtext depends on. The latter are subsumed here under "Xtext" for simplicity. If you want you can download also a distribution which is already bundled with Eclipse 4.3.0 Kepler, but be aware that this is not finalized until end of June 2013.
2. The Eclipse/Xtext distribution is available for multiple platforms.
   a) Linux GTK x86 64 bit
   b) Linux GTK x86 32 bit
   c) Mac OSX x86 64 bit
   d) Windows 64 bit
   e) Windows 32 bit
3. Unpack the downloaded archive file in a directory of your choice.
   Example (Linux):

```
1 cd /opt/local
2 gzip -dc /download/eclipse-SDK-4.2-Xtext-2.3.1-linux-gtk-x86_64.tar.gz | tar
3 xvfp -
```

   The archive will be extracted to a new directory named `eclipse`. Before unpacking the archive, please ensure that there is no subdirectory named `eclipse` yet! Different operating systems may require different unpacking methods.[12]
4. Start Eclipse by running the `eclipse` executable in the newly-created `eclipse` directory.

[12] On Windows do not unpack it into a deep directory, since this might cause troubles with long path names.

1.5 Workspace Setup

Before we begin, start Eclipse and set up a fresh workspace.

Some settings should be done. Open the workspace settings:

- Windows: Window / Preferences
- Mac: Eclipse / Preferences

Workspace Encoding

File encoding is important to some type of files. It is better that the workspace is set to a common encoding to avoid any platform specific encoding. By default the workspace is using platform encoding, which is Cp1252 on Windows and MacRoman on Mac. We will use ISO-8859-1 as a common encoding here.

- Open Eclipse Preferences and go to *General / Workspace*
- Change setting *Text file encoding* to *Other / ISO-8859-1*

Launch Operation

- Open Run/Debug / Launching
- Change "Launch Operation" to "Always launch the previously launched application"

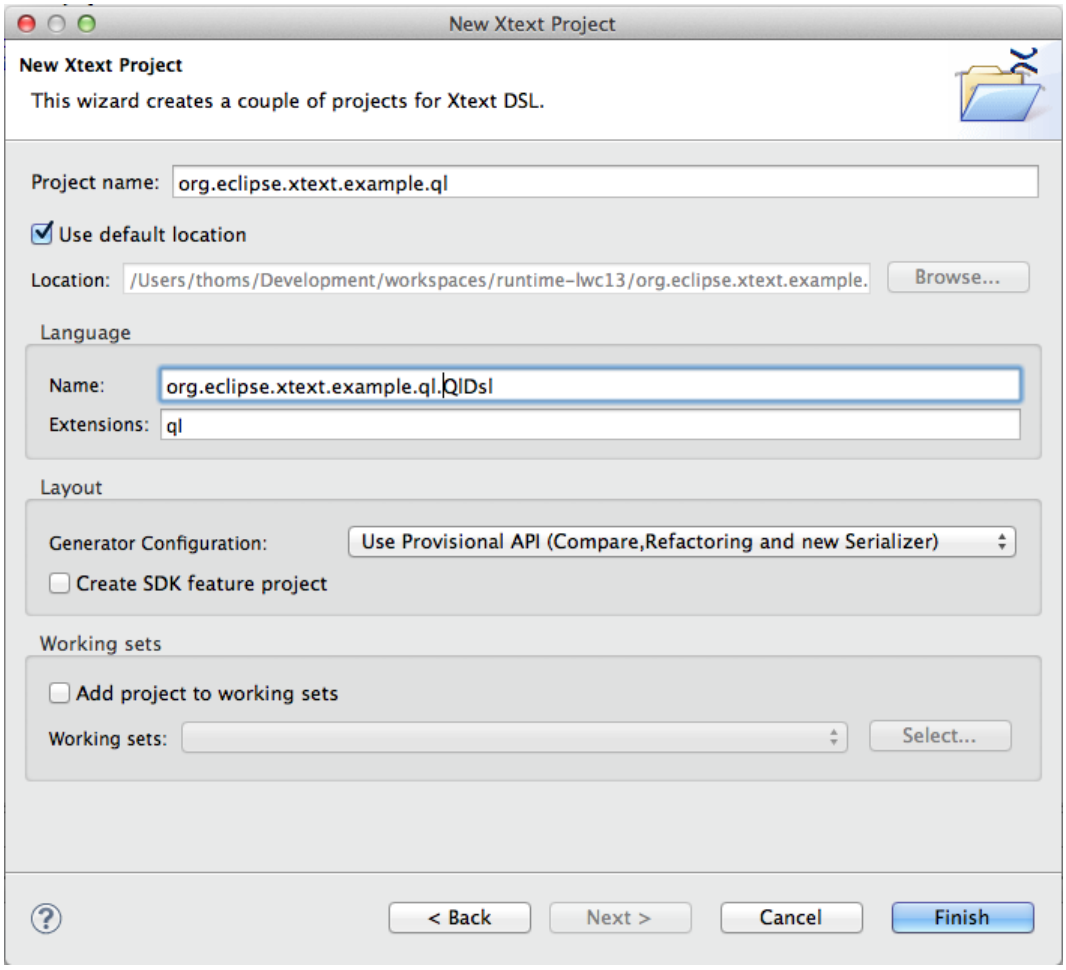
This will allow you re-running the previous launched application by just pressing the Run or Debug button in the Eclipse toolbar, or using keyboard shortcuts. The default settings does not always do what you want.

1.6 The Survey Application

2 Developing the Questionnaire Language

2.1 Create the DSL Projects

Let's start creating the projects for the Questionnaire DSL. Open the New Project Wizard with *File / New / Project*. Choose "*Xtext Project*" and press "*Next*".



On the project wizard page enter:

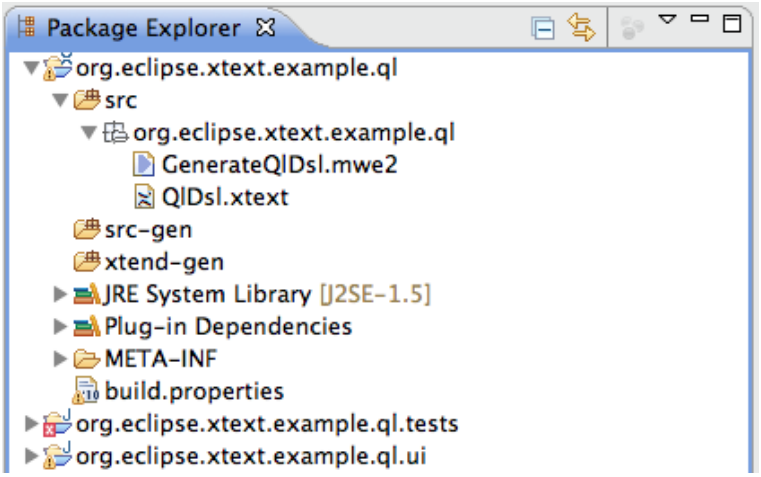
1. Project name: `org.eclipse.xtext.example.ql`. Xtext will create multiple projects, which share this prefix. It is a convention to use a lowercase, dot-separated name.

2. Language name: `org.eclipse.xtext.example.ql.QlDsl`. This is an identifier for the language, which must be unique and follows a Java full qualified identifier name pattern.

3. Language Extensions: `ql`. This will be the file extension for DSL files.

4. Uncheck the option "Create SDK feature project". It would not harm to have that checked, it would just create an additional Feature Project, which we do not handle in this tutorial any further.

Now press "*Finish*". Xtext will generate for you 3 projects into your workspace:



- `org.eclipse.xtext.example.ql`: This is the Runtime Project, which holds the language definition and any implementation which is not UI dependent. Most of the implementation details of this tutorial will be done in this project.

- `org.eclipse.xtext.example.ql.tests`: This project is intended to hold test code for the language. Tests are implemented with JUnit. Xtext will generate some infrastructure code required for tests into here. We will deal testing of DSLs at the end of this tutorial. For now, you can close this project if you want.

- `org.eclipse.xtext.example.ql.ui`: Xtext produces a language specific text editor. The editor is an Eclipse plugin. While the runtime part of the language could be used in any UI or even from command-line, the Editor is dependent on the Eclipse platform.

All projects are almost empty right now. Only the Runtime Project contains two important files in the `/src` folder.

- `GenerateQlDsl.mwe2`: This is a so-called "MWE2 Workflow". MWE is short for "Modeling Workflow Engine", which is a framework that is intended to define processes for code generation[13]. This file defines the process to generate code for the DSL implementation.

- `QlDsl.xtext`: This is the file that contains the DSL language definition itself. It is called the *Grammar* of the language.

[13]More about MWE2 see http://www.eclipse.org/Xtext/documentation.html#MWE2

2.2 Defining the Grammar

Open the Grammar file, `QlDsl.xtext`. In a first step, we will leave out the expression part in the syntax for simplicity. Enter the following text into the Grammar file[14]:

```
grammar org.eclipse.xtext.example.ql.QlDsl with org.eclipse.xtext.xbase.Xbase

generate qlDsl "http://www.eclipse.org/xtext/example/ql/QlDsl"

/* The top-most container of QL files is a Questionnaire */
Questionnaire:
  imports+=Import*
  forms+=Form*;

/* Allows importing of qualified names of types */
Import:
  'import' importedNamespace=QualifiedName;

/* QL consists of questions grouped in a top-level form construct. */
Form:
  "form" name=ID "{"
    element += FormElement*
  "}";

/* Abstract rule for elements contained in a Form */
FormElement:
  Question
;

/**
 * - Each question identified by a name that at the same time represents the result of the
     question.
 * - A question has a label that contains the actual question text presented to the user.
 * - Every question has a type.
 */
Question:
  name=ID ":" label=STRING type=JvmTypeReference
;
```

With the grammar above, the QL language won't fulfill all requirements of the LWC2013 task. We will extend the grammar later to meet all requirements. With this grammar a valid model file would look like this:

[14] https://gist.github.com/kthoms/4758255

```
import types.Money

form Box1HouseOwning {
  hasSoldHouse: "Did you sell a house in 2010?" boolean
  hasBoughtHouse: "Did you by a house in 2010?" boolean
  hasMaintLoan: "Did you enter a loan for maintenance/reconstruction?" boolean

  sellingPrice: "Price the house was sold for :" Money
  privateDebt: "Private debts for the sold house: " Money
  valueResidue: "Value residue: " Money
}
```

Now let us explain the grammar in more detail:

```
grammar org.eclipse.xtext.example.ql.QlDsl with org.eclipse.xtext.xbase.Xbase
```

The grammar has a unique identifier named `org.eclipse.xtext.example.ql.QlDsl`[15]. It is derived from another grammar, `org.eclipse.xtext.xbase.Xbase`. Xbase defines a grammar for expressions, but more on this later. Xtext supports *single inheritance* for grammars.

```
generate qlDsl "http://www.eclipse.org/xtext/example/ql/QlDsl"
```

This is an instruction for the metamodel used for the language. The `generate` statement means that Xtext generates an Ecore metamodel for this grammar[16]. The metamodel will represent the language's Abstract Syntax Tree (AST). Xtext creates the following structure in the Ecore metamodel:

- an EPackage for each `generate` statement. The name of the EPackage is the first argument (`qlDsl`), the package's `nsURI` is the second argument (`"http://www.eclipse.org/xtext/example/ql/QlDsl"`).
- an EClass
  - for each return type of a parser rule. If a parser rule does not define a return type, an implicit one with the same name as the rule itself is assumed. You can specify multiple rules that return the same type but only one EClass will be generated.
  - for each type defined in an action or a cross-reference.
- an EEnum

[15] That's what has been entered in the project wizard

[16] http://www.eclipse.org/Xtext/documentation.html#metamodelInference

  - for each return type of an enum rule.

- an EDataType
  - for each return type of a terminal rule or a data type rule.

Alternatively an Xtext grammar could be mapped to an existing Ecore metamodel [17].

```
1 Questionnaire:
2   imports+=Import*
3   forms+=Form*;
```

The top-most container rule is `Questionnaire`. Per model resource exactly one instance of this type will be contained in the root content of the resource. Any other element will be contained directly or indirectly within this instance.

Each QL model will contain zero to many `import` statements, e.g.:

```
1 import java.math.BigDecimal
2 import types.Money
```

We will use them to import types used as a question's answer type. The `"+="` operator means, that a to-many containment reference with name `imports` is added as EReference to the `Questionnaire` EClass. The `"*"` means that this rule can be repeated zero to many times.[18] After the `import` statements, the QL model can contain multiple `form` declarations.

```
1 Import:
2   'import' importedNamespace=QualifiedName;
```

The `Import` rule is defined to start with the keyword `"import"`, followed by a `QualifiedName`. The `QualifiedName` rule is not defined in the `QlDsl.xtext` grammar itself, it is inherited from the Xbase grammar. This rule defines a so-called Datatype Rule, which maps to datatype, in this case `EString`.

```
1 Import:
2   'import' importedNamespace=QualifiedName;
```

After the imports section QL forms are defined:

```
1 Form:
2   "form" name=ID "{"
3     element += FormElement*
4   "}";
```

[17] http://www.eclipse.org/Xtext/documentation.html#grammarMixins
[18] To enforce at least one rule call, the `"+"` operator would be used instead.

Forms have an attribute called `name`. `ID` is a *terminal rule*, which is defined in Xtext's root grammar `Terminals`. It allows typical Java-style identifiers (beginning with a word character followed by arbitrary many characters, numbers or underscores).

The next step is to define the rule `FormElement`. It is an abstract rule which will collect the different alternatives of elements that can be contained in a form. In our first step, the rule `Question` will be the only alternative. We will introduce a second alternative later in the grammar, and in order to reduce the refactoring effort we are introducing the `FormElement` already now.

```
1 FormElement:
2   Question
3 ;
```

Finally, the `Question` rule is defined. In a first step, Questions are identified by a `name`, followed by a `label` string and a reference to a `type`. Later we will add expressions to compute the value.

```
1 Question:
2   name=ID ":" label=STRING type=JvmTypeReference
3 ;
```

A Question's type is a reference to a JVM Type. Think of this for now that we refer to Java types. The `JvmTypeReference` rule is also inherited through Xbase, actually Xbase derives itself from another grammar, Xtype, which declares these rules.

2.3 Generate Language Implementation

Now that the initial grammar of the language has been defined it is time to test the language. Xtext ships with a code generator which generates all the glue code needed for the language implementation.

To generate the code, we need to execute the generator workflow `GenerateQlDsl.mwe2`. For this, select the workflow file, open the context menu and select *Run As / MWE2 Workflow*.

The generator will print some information to the Console, and finally it should print `"Done."`.

```
1 0 [main] INFO lipse.emf.mwe.utils.StandaloneSetup - Registering platform
2 uri ....
3    ...
4 13727 [main] INFO .emf.mwe2.runtime.workflow.Workflow - Done.
```

After successful execution the projects will be filled with implementation code. Code that will be regenerated each time the generator is executed will go to the source folder `/src-gen` (in all three projects), whereas code generated to `/src` will be generated only once as skeleton. It is safe to edit these classes.

Xtext follows the Generation Gap Pattern[19]: Generated code is based on the Xtext API. Manual code is separated from generated code. Often manual classes are derived from generated classes to allow overriding of generated code or adding functionality.

Investigate the generated code a bit. Some pieces to mention:

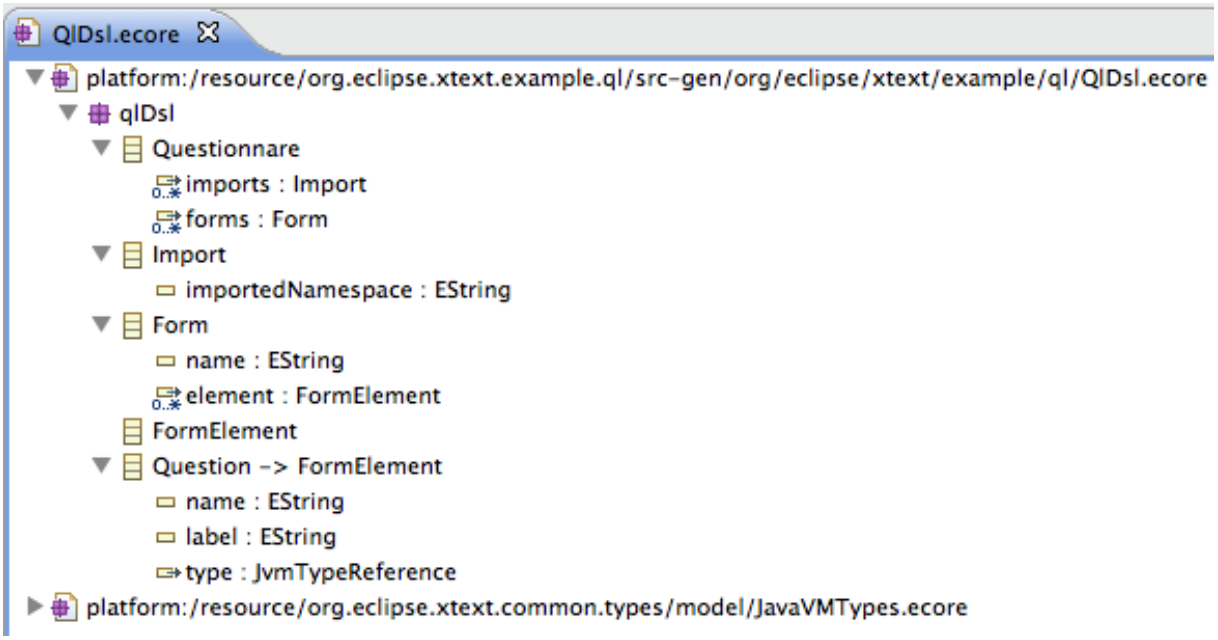
Runtime Project - folder `src`

- The class `QlDslRuntimeModule` is a Guice configuration. Guice[20] is a famous Dependency Injection[21] framework in Java. Xtext makes heavy use of Dependency Injection, which in turn allows to exchange nearly every bit of the framework for customizing or to work around limitations, if necessary, without the need to change the framework itself.
- Class `QlDslStandaloneSetup` is needed when using the language in "standalone mode", i.e. without an Eclipse environment. Eclipse plugins, like Xtext and the language plugin, usually need an OSGi container as execution environment. Xtext is designed to be executable without the need to be deployed into an OSGi container, but for this certain registrations are required which an OSGi container would usually provide automatically. This is especially useful when Xtext based languages are used in build environments or other IDEs.
- Class `QlDslFormatter` allows the implementation of a declarative code formatter for the DSL.
- File `QlDslJvmModelInferrer.xtend` is a class implemented with the Xtend language. The JVM Model Inferrer will play an important role later when we introduce expressions and code generation.
- Class QlDslJavaValidation allows the implementation of validation rules for the DSL.

Runtime Project - folder `src-gen`

[19] http://heikobehrens.net/2009/04/23/generation-gap-pattern/
[20] http://code.google.com/p/google-guice/
[21] http://en.wikipedia.org/wiki/Dependency_injection

- The Ecore metamodel is generated to file `QlDsl.ecore`.



- The Java implementation code for the metamodel can be found in the package `org.eclipse.xtext.example.ql.qlDsl`.
- The package `org.eclipse.xtext.example.ql.parser.antlr.internal` contains an ANTLR3[22] grammar and the Lexer and Parser classes generated from it.

2.4 Testing the Questionnaire Language

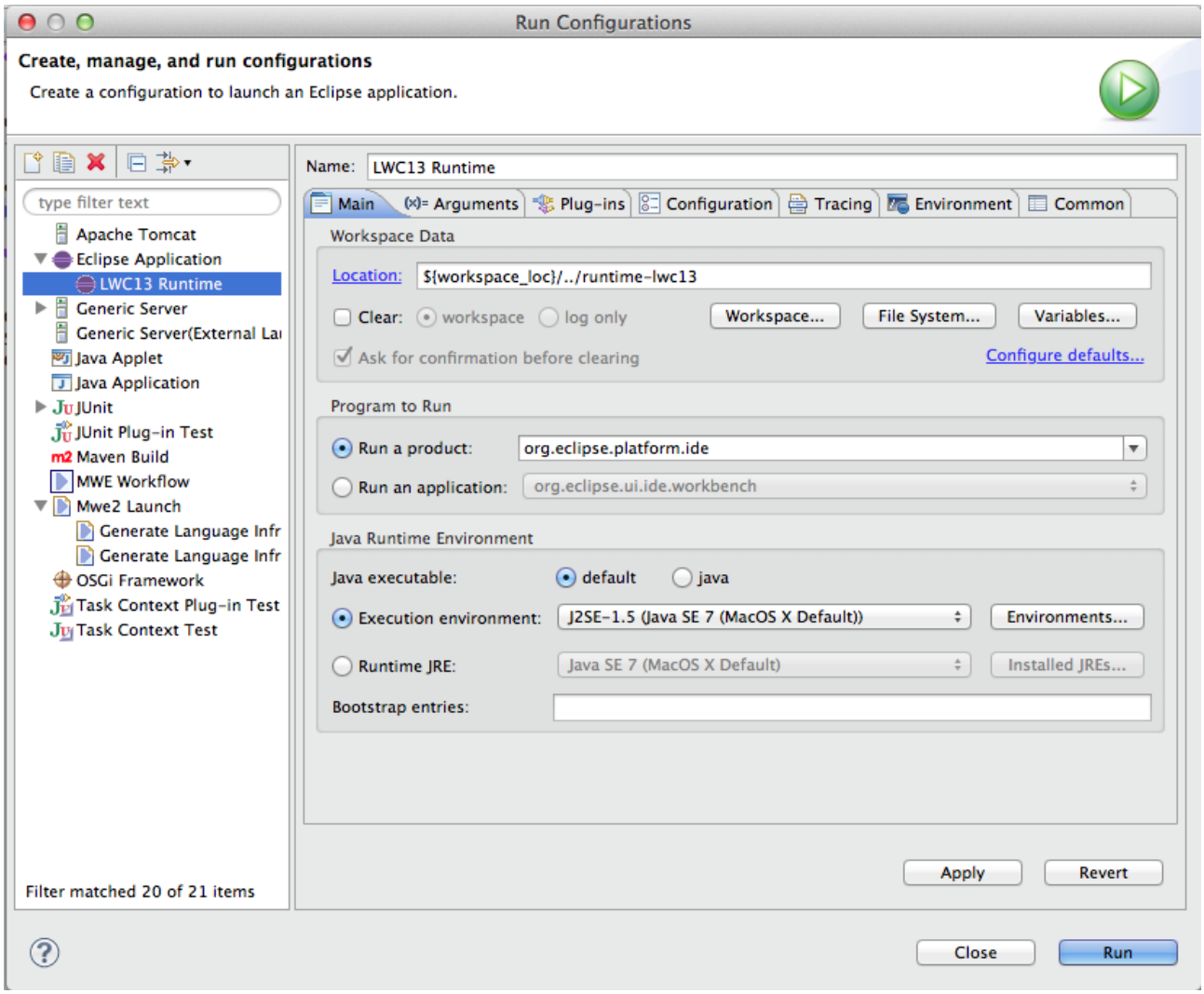
2.4.1 Creating a Launch Configuration

In order to test the language and the editor we need to deploy the developed plugins within another Eclipse instance. For testing the easiest way is start a so-called Runtime Instance.

Open the dialog *Run / Run Configurations* and select the node *Eclipse Application* from the left tree widget and press the icon with the + sign to create a new Launch Config.

You could leave the defaults here or change the name and location like in the screenshot.

[22] http://www.antlr.org



Now switch to the Arguments page and enter in the "VM arguments" text box:

```
1 -Xms40m -Xmx512m -XX:MaxPermSize=150m
```

Especially important is the MaxPermSize setting, since the default size of the PermGen space of the VM (64MB) often is not enough.

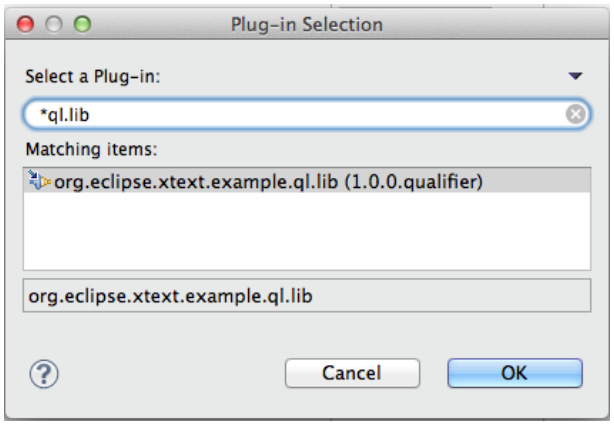
Now press the *"Run"* button. Another Eclipse instance will start with an empty workspace. Close the Welcome window.

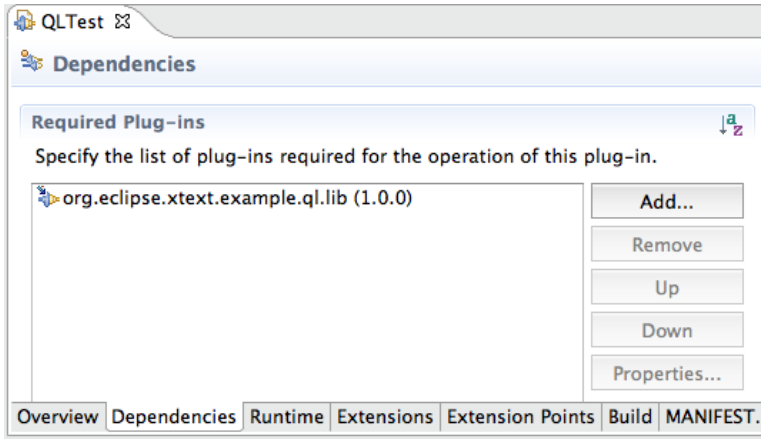
2.4.2 Create Test Project

In the Runtime Workspace create a new Plug-in Project with name "QLTest".[23]

[23] As before, uncheck the options on the second wizard page.

First, we will create a dependency to the library plugin. Open the `META-INF/MANIFEST.MF` file and switch to page Dependencies. In the *Required Plug-ins* section, add a dependency to `org.eclipse.xtext.example.ql.lib`.

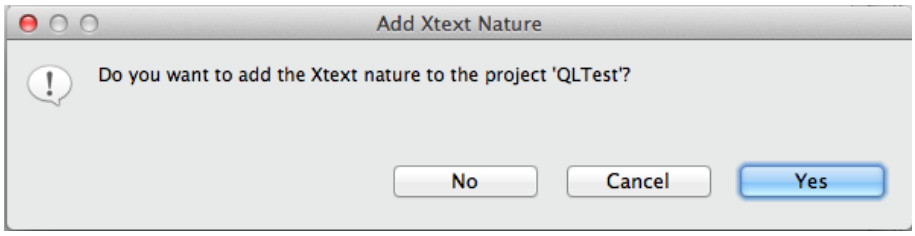




The DSL has to support custom datatypes like `Money`, which must be defined. Select the `/src` folder and create a new Java class `Money` in package `types`:

```
1  package types;
2
3  import java.math.BigDecimal;
4
5  public class Money {
6    private BigDecimal amount;
7
8    public Money(BigDecimal amount) {
9      this.amount = amount;
10   }
11
12   public BigDecimal getAmount() {
13     return amount;
14   }
15 }
```

Select the `/src` folder and create a new file `"housepurchase.ql"`. Once you have created the file a popup dialog will appear to ask, if you would like to add the Xtext nature on this project. Answer with "Yes".



From now on your project will be considered to contain files that Xtext should recognize (`.ql` files). Projects having the Xtext nature will be processed by the Xtext Builder when building projects, other projects are ignored. The Xtext Builder indexes the Xtext based resources, links the cross-references in the editor, and validates the model files. On errors, resource markers are created which can be seen in the editor and the *Problems View*.

Enter the content for `"housepurchase.ql"`[24]:

```
1  import types.Money
2
3  form Box1HouseOwning {
4    hasSoldHouse: "Did you sell a house in 2010?" boolean
5    hasBoughtHouse: "Did you by a house in 2010?" boolean
6    hasMaintLoan: "Did you enter a loan for maintenance/reconstruction?" boolean
7
8    sellingPrice: "Price the house was sold for :" Money
9    privateDebt: "Private debts for the sold house: " Money
10   valueResidue: "Value residue: " Money
11 }
```

You see now that the editor has recognized our DSL. The language's keywords are highlighted. Xtext offers far more than just syntax coloring for the language, it created a fully integrated editor. You may explore some of the features now.

- If you make errors, error markers are created and resolved while you type.
- Content assist is offered with CTRL+SPACE.
- The Outline view [25] presents the structure of the document, and allows quick navigation.
- F3 allows jumping to the definition of an element, which is defined somewhere else. You

[24] https://gist.github.com/kthoms/5036304
[25] if not present, open with *Window / Show View / Outline*

could try this by selecting “Money” and press F3. At the moment, only the type information of questions is cross-referenced.

From now on, we will extend the DSL a bit further. This usually requires to restart the test environment. So close it and proceed reading.

2.5 Xbase

The language developed in section 2.1 does not yet meet all demands on the LWC2013 task. Two core features are missing: First, a question’s answer can be computed, i.e. its answer can be derived from an expression referring to previous questions’ answers. Second, questions can be optional depending on the previous answers. For this, also the possibility to define expressions is needed. This is where Xbase comes into play.

Xbase is an expression language which can be reused in your own Xtext DSL. Its language concepts are similar to Java, but with some syntactical derivations improving readability. The Xbase grammar is defined in Xtext, thus its elements can be used in any other Xtext grammar by importing or directly extending Xbase via Xtext’s possibility for grammar inheritance. In addition to the grammar, Xbase ships with further infrastructural parts like a compiler, interpreter, linker or static analyzer which all can be adapted to your own needs. In the background, Xbase produces plain Java code which is run on the JVM. Like other DSLs defined with Xtext, Xbase provides also editor features like syntax highlighting, content assistance and navigation via hyperlinks. In the following we will first introduce some language concepts of Xbase, and afterwards we will describe how to integrate Xbase into the Questionnaire DSL.

In Xbase everything is an expression which always has a return type which might be `null` for some expressions. Variables are defined with the `var` keyword, whereas for constant values the `val` keyword is used. Types are derived automatically, so they don’t need to be defined explicitly:

```
1 var myVariable = 'some modifiable value'
2 val Integer myConstant = 42
```

Xbase ships with a library extending existing Java types like String or Integer with further functionality. So besides the already known String operations from Java like `toUpperCase` or `toLowerCase`, in Xbase expressions you can also use `toFirstUpper` and `toFirstLower` changing only the first letter’s case which might come in handy in some situations. Large numbers can be written more readable by using underscores to separate digits:

```
"a day has ".toFirstUpper() + 86_400_000 + " milliseconds."
// results in: A day has 86400000 milliseconds.
```

As in Java, Xbase provides `if-else`-expressions for defining conditions. Since each expression has a return type, it is valid to use `if-else`-blocks similar to the ternary operator in Java:

```
var x = if (condition) 42 else 43
```

There are further concepts in Xbase which we will not cover here in more detail, since they have not much relevance for the Questionnaire language. So e.g. it is possible to use loops for iterating over a collection of element; there is a `switch-case`-expression with type guards allowing for defining behavior depending on the type of a parameter; and last but not least, Xbase allows the definition of closures. For more details, please look up the reference documentation[26] or the Xbase tutorials directly in Eclipse (*File / New / Other.. / Xbase Tutorial*).

With these capabilities integrated in the Questionnaire language it is feasible to define complex domain logic e.g. for the result of a questionnaire directly in its definition. For example, when designing a questionnaire for a test, let's say to define a person's stress level, you can write some Xbase code as expression for the last "result" question:

```
stressLevelResult: "Your Stress-Level: " String (
  {
    var Integer stressPoints = if (hasTimePressureAtWork) 30 else 0
    stressPoints = stressPoints + daysSleepingBadPerWeek * 3
    stressPoints = stressPoints + glassesOfAlcoholPerDay * 12
    stressPoints = stressPoints - daysWithSportPerWeek * 2
    if (stressPoints>80) "High" else if (stressPoints>40) "Medium" else "Low"
  }
)
```

2.6 Including Expressions into the QL Language

Recall the example of a housowning questionnaire as mentioned in the LWC13 task:

```
import types.Money

form Box1HouseOwning {
  hasSoldHouse: "Did you sell a house in 2010?" boolean
  hasBoughtHouse: "Did you by a house in 2010?" boolean
  hasMaintLoan: "Did you enter a loan for maintenance/reconstruction?" boolean
```

[26] http://www.eclipse.org/Xtext/documentation.html#xbaseLanguageRef_Introduction

```
7
8   if (hasSoldHouse) {
9     sellingPrice: "Price the house was sold for: " Money
10    privateDebt: "Private debts for the sold house: " Money
11    valueResidue: "Value residue: " Money (sellingPrice - privateDebt)
12  }
13 }
```

Compared to the language developed in section 2.1, we need to add (1) a condition statement to express optional questions (see line 8) and (2) the capability for automatically deriving a question's answer from previously answered questions (see line 11). As explained in section 2.1, our grammar inherits from Xbase making its rules reusable in the questionnaire language. To fulfill the missing requirements our grammar needs to be extended to the following: [27]

```
1  grammar org.eclipse.xtext.example.ql.QlDsl with org.eclipse.xtext.xbase.Xbase
2
3  generate qlDsl "http://www.eclipse.org/xtext/example/ql/QlDsl"
4
5  /* The top-most container of QL files is a Questionnaire */
6  Questionnaire:
7    imports+=Import*
8    forms+=Form*;
9
10 /* Allows importing of qualified names of types */
11 Import:
12   'import' importedNamespace=QualifiedName;
13
14 /* QL consists of questions grouped in a top-level form construct. */
15 Form:
16   "form" name=ID "{"
17     element += FormElement*
18   "}";
19
20 /* Abstract rule for elements contained in a Form */
21 FormElement:
22   Question | ConditionalQuestionGroup
23 ;
24
25 /**
26  * - Each question identified by a name that at the same time represents the result of the
      question.
27  * - A question has a label that contains the actual question text presented to the user.
28  * - Every question has a type.
```

[27] https://gist.github.com/kthoms/5114439

```
29  * - A question can optionally be associated to an expression:
30  * this makes the question computed
31  */
32 Question:
33   name=ID ":" label=STRING type=JvmTypeReference expression=XParenthesizedExpression?
34 ;
35
36 /**
37  * Groups questions within a block, optionally made conditional with an if-condition.
38  */
39 ConditionalQuestionGroup: {ConditionalQuestionGroup}
40   ("if" condition=XParenthesizedExpression)? "{"
41     element += FormElement*
42   "}"
43 ;
```

Compared to the grammar defined in section 2.1, the follwoing points have changed:

```
1 FormElement:
2   Question | ConditionalQuestionGroup
3 ;
```

A `FormElement` is now either a normal question or a `ConditionalQuestionGroup`. Conditional question groups are groups of form elements embraced by an optional `if`-condition:

```
1 ConditionalQuestionGroup: {ConditionalQuestionGroup}
2   ("if" condition=XParenthesizedExpression)? "{"
3     element += FormElement*
4   "}"
5 ;
```

For the condition of the `if`-statement the grammar rule `XParenthesizedExpression` inherited from Xbase is used. An `XParenthesizedExpression` is simply an expression in parenthesis. The `if`-statement is optional (as defined by the question mark '?' symbol) which allows for just grouping questions without the necessarity for a condition. The inner elements are again `FormElement`s, making it possible to nest groups within groups and so on. The last part that has changed is the `Question` rule. Here again the rule `XParenthesizedExpression` is used to optionally embed Xbase expressions:

```
1 Question:
2   name=ID ":" label=STRING type=JvmTypeReference expression=XParenthesizedExpression?
3 ;
```

After changing the grammar, the implementation has to be regenerated. Run the `GenerateQlDsl.mwe2` workflow again. Then restart the runtime workbench. [28]

Xbase comes out of the box with the support for standard Java types like Strings or Integers inside expressions. However, in the questionnaire language own data types, like the Money type from the example, need also to be integrated. Such data types will be typically defined in a Java class. When importing such a type via the `import` statement, it will be available in the questionnaire definition. Xbase needs to know how to handle these types when they are used in expressions with operators like '+'. '-', '*' and '/'. The logic for these operators need to be implemented in special methods in the data type itself. As example, let's see how this is achieved for the Money type:[29]

```
1  package types;
2
3  import java.math.BigDecimal;
4
5  public class Money {
6    private BigDecimal amount;
7
8    public Money (BigDecimal amount) {
9      this.amount = amount;
10   }
11   public BigDecimal getAmount() {
12     return amount;
13   }
14
15   // Implement operators
16   public Money operator_minus (Money other) {
17     return new Money(this.amount.subtract(other.amount));
18   }
19   public Money operator_plus (Money other) {
20     return new Money(this.amount.add(other.amount));
21   }
22   public Money operator_multiply (Money other) {
23     return new Money(this.amount.multiply(other.amount));
24   }
25   public Money operator_divide (Money other) {
26     return new Money(this.amount.divide(other.amount));
27   }
28 }
```

[28] Select it from the Run / Run Configurations dialog or from the drop down menu next to the green "play" button in the tool bar.

[29] http://code.google.com/a/eclipselabs.org/p/lwc13-xtext/source/browse/examples/QLTest/src/types/Money.java

The data type `Money` simply holds the amount as a value of type `BigDecimal`. For each operator a special method, e.g. `operator_minus(Money other)`, defines how to procede when this operator is used two values of type `Money`. In this simple example, a new Money object is created and its value is computed corresponding to the operator type. When evaluating an expression, Xbase searches for these methods inside the used types to compute the result.

In order to test the new version of the questionnaire language, the MWE workflow needs to be executed again (*Right click on GenerateQlDsl.mwe2 / Run As.. / MWE2 Workflow*). The questionnaire language now supports expressions, but there is still one point missing: Questions cannot be referenced within an expression. For this, we need to derive a JVM model from the questionnaire model which we will discuss in the next section.

2.7 JVM Model Inference

For languages using Xbase it is necessary to tell Xtext, how to map concepts of a language to a Java model. In our example, a Form could be mapped to the Type concept, while Questions are the fields of a class. By doing this, elements of the language can be made available in expressions. Further, it allows that model elements are linkable where Java types are expected, without necessarily generate a Java class.

The derivation of the Java model for language concepts is the responsibility of the JVM Model Inferrer, which is a class that implements the `IJvmModelInferrer` interface. A skeleton has already been generated into package `org.eclipse.xtext.example.ql.jvmmodel`. The file `QlDslJvmModelInferrer.xtend` is a class written with Xtend.

The mapping that has to be implemented for the Questionnaire DSL should be as follows:

1. Each `Form` instance is mapped to a `JvmDeclaredType` (which is the common concept for Java classes and interfaces). The type's name is simply the form name, and the target package is forms.

2. Each `Question` of a `Form` is mapped to a `JvmField`, which is added as member of the declared type

3. For each `Question` accessor methods for the field are generated. The field gets only a setter if the value of the Question is not computed by an expression. If the field is computed, the content of the getter has to compute the result.

4. For each `Question` a method `is<QUESTIONNAME>Enabled()` is inferred. Questions with computed values are not enabled.

5. For each `ConditionalQuestionGroup` a method is produced that computes whether the group is visible or not.

Now place the content into the inferrer class[30] :

```
1  package org.eclipse.xtext.example.ql.jvmmodel
2
3  import com.google.inject.Inject
4  import java.io.Serializable
5  import org.eclipse.xtext.common.types.JvmOperation
6  import org.eclipse.xtext.common.types.util.TypeReferences
7  import org.eclipse.xtext.example.ql.qlDsl.ConditionalQuestionGroup
8  import org.eclipse.xtext.example.ql.qlDsl.Question
9  import org.eclipse.xtext.example.ql.qlDsl.Questionnaire
10 import org.eclipse.xtext.xbase.XExpression
11 import org.eclipse.xtext.xbase.XbaseFactory
12 import org.eclipse.xtext.xbase.jvmmodel.AbstractModelInferrer
13 import org.eclipse.xtext.xbase.jvmmodel.IJvmDeclaredTypeAcceptor
14 import org.eclipse.xtext.xbase.jvmmodel.JvmTypesBuilder
15
16 class QlDslJvmModelInferrer extends AbstractModelInferrer {
17   @Inject extension JvmTypesBuilder
18   @Inject TypeReferences typeReferences
19
20 def dispatch void infer(Questionnaire element, IJvmDeclaredTypeAcceptor acceptor, boolean
       isPreIndexingPhase) {
21    for (form: element.forms) {
22      acceptor.accept(form.toClass("forms."+form.name))
23      .initializeLater[
24        //implements Serializable
25        it.superTypes +=typeReferences.getTypeForName(typeof(Serializable),element,null)
26
27        members += toField("serialVersionUID",typeReferences.getTypeForName("long",element)
              ,[final = true ^static = true
28          setInitializer([it.append("1L")])
29        ])
30
31        val allQuestions = form.eAllContents.filter(typeof(Question)).toList
32
33        for (question: allQuestions) {
34          members += question.toField(question.name, question.type)
```

[30] https://gist.github.com/kthoms/5132153

```
35        }
36
37        for (question: allQuestions) {
38          if (question.expression == null) {
39            members += question.toGetter(question.name, question.type)
40            members += question.toSetter(question.name, question.type)
41          } else {
42            val getter = question.toGetter(question.name, question.type)
43            getter.body = question.expression
44            members += getter
45          }
46          members += question.createIsEnabledMethod
47        }
48
49        val allQuestionGroups = form.eAllContents.filter(typeof(ConditionalQuestionGroup)).
              toList
50        var groupIndex=0;
51        for (questionGroup: allQuestionGroups) {
52          members += questionGroup.createIsGroupVisibleMethod(groupIndex)
53          groupIndex = groupIndex+1
54        }
55
56      ]
57    }
58  }
59
60   def JvmOperation createIsEnabledMethod (Question question) {
61     question.toMethod("is"+question.name.toFirstUpper+"Enabled", typeReferences.
          getTypeForName("boolean", question, null)) [
62       body = [it.append('''return Â«question.expression == nullÂ»;''')]
63     ]
64   }
65
66   /** Create a method <code>public boolean isGroup[groupIndex]Visible ()</code>.*/
67   def JvmOperation createIsGroupVisibleMethod (ConditionalQuestionGroup group, int
         groupIndex) {
68     group.toMethod("isGroup"+groupIndex+"Visible", typeReferences.getTypeForName("boolean",
           group, null)) [
69       if(group.condition != null) {
70         body = group.condition
71       } else {
72         body = [it.append('''return true;''')]
73       }
74     ]
75   }
```

```
76
77 }
```

Now lets take a deeper look at the implementation:

```
1 class QlDslJvmModelInferrer extends AbstractModelInferrer {
2   @Inject extension JvmTypesBuilder
3   @Inject TypeReferences typeReferences
4   def dispatch void infer(Questionnaire element, IJvmDeclaredTypeAcceptor acceptor, boolean
        isPreIndexingPhase) {
5     ...
6   }
7 }
```

The inferrer class implements `IJvmModelInferrer`, but for convenience we derive from its abstract implementation `AbstractModelInferrer`. The main method to implement is `infer()`. In the case of QL models, the root element of model resources is a `Questionnaire`. The base implementation uses polymorphic dispatching on the root element of a model resource, and the `infer()` method of our implementation hooks into the dispatching by using the dispatch keyword. That is also why the first argument can be of type `Questionnaire`, and not of the base type `EObject`, like defined in the `infer()` method that is definied in `IJvmModelInferrer`.

The implementation uses two services, which are injected as members into the class:

- The `JvmTypesBuilder` offers factory and builder functions to create instances of JVM Model types. The additional keyword `extension` has the effect, that the methods of the `JvmTypesBuilder` become so-called **extension methods**. This means, the functions become implicitly available as additional methods on the first argument of the function. We will see extensive use of this nice feature of Xtend in the implementation of the Xtend based code generator in the next chapter.

- `TypeReferences` is used to retrieve the respective JVM Model instances for given qualified Java class names through its `getTypeForName()` methods.

```
1   for (form: element.forms) {
2     acceptor.accept(form.toClass("forms."+form.name))
3     .initializeLater[
4       ...
5     ]
6   }
```

Let's take a deeper look on the `infer()` method. The outer loop simply iterates over the `Form`

instances of the `Questionnaire` element. Inside the loop we first derive a Class instance for each `Questionnaire` element in package `forms`. JVM Model Inference is executed in two phases: In the first phase all types are derived, without any content. In the second phase, the content of the types is derived. This is done by the closure passed to `initializeLater()`. The reason why this has to happen this way is that during inference of type members, they could refer again to types that are derived by the inferrer. The two phases prevent circular calls.

```
1 it.superTypes +=typeReferences.getTypeForName(typeof(Serializable),element,null)
2
3 members += toField("serialVersionUID",typeReferences.getTypeForName("long",element),
4 [final = true ^static = true
5   setInitializer([it.append("1L")])
6 ])
```

We want to make the resulting Java class serializable. This is optional, but better style. Therefore the class has to implement the `java.io.Serializable` interface, whose JVM Model representative is retrieved from the `TypeReferences` instance and added to the `superTypes` collection. The identifier `it` denotes the implicit variable of type Form of the closure. It is not necessary to qualify it here, it could be left out. The closure passed to the `setInitializer()` method initializes the field with the value "1" of type long.

```
1 val allQuestions = form.eAllContents.filter(typeof(Question)).toList
2
3 for (question: allQuestions) {
4   members += question.toField(question.name, question.type)
5 }
```

All `Question` instances from the resource are bound to the final variable `allQuestions`. Since Questions can be nested into groups, the content has to be searched recursively. `eAllContents` will traverse over all elements.

Next, for each `Question` a `JvmField` instance is inferred. Here the `JvmTypesBuilder` is helping us with the method `toField`, which gets the name and type of the derived field. Here we see the effect of the extension keyword: It seems that `toField` is actually a method of type Question, but it is a method of the `JvmTypesBuilder` class.

```
1 for (question: allQuestions) {
2   if (question.expression == null) {
3     members += question.toGetter(question.name, question.type)
4     members += question.toSetter(question.name, question.type)
5   } else {
6     val getter = question.toGetter(question.name, question.type)
7     getter.body = question.expression
```

```
8     members += getter
9   }
10  ...
11 }
```

The next loop creates the accessor methods for the fields. We could have done this in the previous loop also, but it is better style to declare the fields first, and methods next in the class. The inferred `JvmDeclaredType` will be translated to Java later, so it is better to have that clean from the beginning.

Within the loop, we decide if the question has a computation expression or not. If it hasn't one, it is a simple field with getter and setter, where we call the `toGetter()`/`toSetter()` builder functions. If the question value is computed by an expression, it does not make sense to offer a setter method. The field needs to be read-only. The getter method does not simply return the value of a field. Instead, the method has to evaluate the expression. Thus, we assign the expression as body of the method.

```
1  for (question: allQuestions) {
2    ...
3    members += question.createIsEnabledMethod
4  }
5
6  ...
7  def JvmOperation createIsEnabledMethod (Question question) {
8    question.toMethod("is"+question.name.toFirstUpper+"Enabled",
9   typeReferences.getTypeForName("boolean", question, null)) [ body = [it.append('''return
       Â«question.expression == nullÂ»;''')]
10   ]
11 }
```

For each `Question` a method `boolean is<QUESTIONNAME>Enabled()` is inferred. The body of the method does simply return `true` if the Question does not have an computation expression assigned, or `false` otherwise.

In this case we assign to the body a closure that computes the method implementation text. This is the first example where we make use of Xtend's *Rich String* feature (the text between the three single quotes '''), which is later heavily used in the code generator templates.

```
1  val allQuestionGroups = form.eAllContents.filter(typeof(ConditionalQuestionGroup)).toList
2  var groupIndex=0;
3  for (questionGroup: allQuestionGroups) {
4    members += questionGroup.createIsGroupVisibleMethod(groupIndex)
5    groupIndex = groupIndex+1
```

```
6  }
7
8  def JvmOperation createIsGroupVisibleMethod (ConditionalQuestionGroup group, int groupIndex)
       {
9   group.toMethod("isGroup"+groupIndex+"Visible", typeReferences.getTypeForName("boolean",
       group, null)) [
10    if(group.condition != null) {
11      body = group.condition
12    } else {
13      body = [it.append('''return true;''')]
14    }
15  ]
16 }
```

We now filter all `ConditionalQuestionGroup` instances from the `Questionnaire` and loop over them. For each of them, a method `is<QUESTIONGROUPINDEX>Visible()` is produced. Unfortunately, question groups are anonymous, thus we maintain an index counter and name the methods `isGroup<IDX>Visible()`.

Since condition expressions for groups are optional, the method body has to return simply `true` in the case that no expression is assigned. When groups have a condition, the condition expression is assigned as the method body.

2.8 Scoping

Scoping is, roughly said, the computation of referrable names in a given context. It is a quite complex topic, and we won't cover it here into deep. The topic itself is heavily documented by the Xtext user manual[31], several articles[32] and implementation examples [33].

The Xtext framework already provides default implementations to solve scoping and linking. In the case of Xbase based languages the `XbaseScopeProvider` is configured by default as implementation of the `IScopeProvider` interface. We have to extend its behavior to recognize the current `Form` instance as variable "`this`"within an expression. By this, Question fields are noticed as variables in expressions, which allows us to use the required expressions.

Create the class `QlScopeProvider` in package `org.eclipse.xtext.example.ql.customizing`:[34]

[31] Xtext manual:Scoping
[32] e.g. http://blogs.itemis.de/stundzig/archives/776
[33] e.g. https://github.com/LorenzoBettini/xtext-scoping
[34] urlhttp://tinyurl.com/bd7kggk

```
1  package org.eclipse.xtext.example.ql.customizing;
2
3  import java.util.Iterator;
4
5  import javax.inject.Inject;
6
7  import org.eclipse.emf.ecore.EObject;
8  import org.eclipse.xtext.EcoreUtil2;
9  import org.eclipse.xtext.common.types.JvmType;
10 import org.eclipse.xtext.example.ql.qlDsl.Form;
11 import org.eclipse.xtext.resource.EObjectDescription;
12 import org.eclipse.xtext.scoping.IScope;
13 import org.eclipse.xtext.scoping.impl.SingletonScope;
14 import org.eclipse.xtext.xbase.jvmmodel.IJvmModelAssociations;
15 import org.eclipse.xtext.xbase.scoping.LocalVariableScopeContext;
16 import org.eclipse.xtext.xbase.scoping.XbaseScopeProvider;
17
18 import com.google.common.collect.Iterables;
19
20 /**
21  * Adds local variable bindings for expressions.
22  */
23 @SuppressWarnings("restriction")
24 public class QlScopeProvider extends XbaseScopeProvider {
25   @Inject
26   private IJvmModelAssociations associations;
27
28   @Override
29   protected IScope createLocalVarScope(IScope parentScope,
30       LocalVariableScopeContext scopeContext) {
31     // search the Form instance of the current context by traversing
32     // the containment hierarchy
33     Form qlForm = EcoreUtil2.getContainerOfType(scopeContext.getContext(),
34         Form.class);
35     if (qlForm != null) {
36       // form found => get the JvmType representative of the Form
37       JvmType jvmTypeOfForm = getJvmType(qlForm);
38       // bind the EClass' JvmType as variable 'this' by creating
39       // a scope that consists of only one element, and delegate
40       // to the default implementation as outer scope
41       IScope result = new SingletonScope(EObjectDescription.create(
42           XbaseScopeProvider.THIS, jvmTypeOfForm), super.createLocalVarScope(
43           parentScope, scopeContext));
44       return result;
45
```

```
46      }
47      return super.createLocalVarScope(parentScope, scopeContext);
48    }
49
50    /**
51     * Find the JvmType associated with an object
52     *
53     * @param context
54     * Some model object
55     * @return The JvmType associated with the object, usually derived by the
56     * IJvmModelInferrer. Returns <code>null</code> if the object is not
57     * associated to an JvmType.
58     */
59    private final JvmType getJvmType(EObject context) {
60      Iterable<JvmType> jvmTypes = Iterables.filter(
61          associations.getJvmElements(context), JvmType.class);
62      Iterator<JvmType> it = jvmTypes.iterator();
63      JvmType result = it.hasNext() ? it.next() : null;
64      return result;
65    }
66
67  }
```

Whenever a default implementation must be exchanged by a custom one, this has to be added or overridden in the respective Guice module. In the case of the scope provider, this has to be added to the `QlDslRuntimeModule` class. Open this class and add the binding:

```
1  public class QlDslRuntimeModule extends
2      org.eclipse.xtext.example.ql.AbstractQlDslRuntimeModule {
3    @Override
4    public Class<? extends IScopeProvider> bindIScopeProvider() {
5      return QlScopeProvider.class;
6    }
7    ...
8  }
```

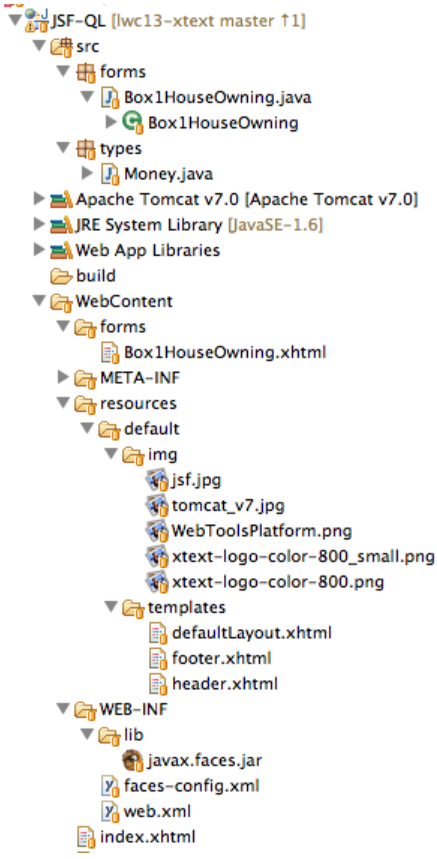
3 Developing the Code Generator

3.1 Reference Implementation

Before implementing a code generator one has to know what the target code is. Therefore a reference implementation has been developed which was coded to large degree manually. From this reference code the templates can be derived. Also this is a manual step.

We use a Java Server Faces (JSF) based application, which can be deployed on any Java Web container (also known as a Servlet container) like Glassfish, JBoss and Apache Tomcat.

The following screenshot shows the structure of the web application project. The application is available for download from the project homepage (JSF-QL-1.0.zip)



Large parts of the application are not derivable from the model, they build the skeleton of the project. This is:

- Custom types (`src/types/*`)
- Custom type converter (`src/converter/*`)

- Images (`WebContent/resources/default/img/*`)
- Page Templates (`WebContent/resources/default/templates/*`)
- Libraries (`WebContent/WEB-INF/lib/*`)
- Web Application Descriptor (`WebContent/WEB-INF/web.xml`)
- Faces configuration (`WebContent/WEB-INF/faces-config.xml`)

We will focus on the parts which are dependent on the QL model and thus subject of code generation. These artifacts are:

- Java Bean classes representing the state of a Form (`src/forms`)
- JSF enabled XHTML pages representing the presentation of a Form (`WebContent/forms/*`)

... to be continued

- setup webtools
- import reference apllication
- install tomcat
- run reference application

To integrate the generated artifacts into our web application we have to make manually changes in the `welcome-file` declared in `WebContent/WEB-INF/web.xml` later. (`WebContent/index.xhtml`)

3.2 Xtend

Since we will use Xtend to write the code generator, this chapter describes some basic concepts of the Xtend language. However, we will not cover all aspects of Xtend in this section, for more information see also the official documentation[35].

Xtend is a statically-typed general purpose language similar to Java. Xtend uses Xbase as core language which was already roughly explained in section 2.5. Hence, the presented concepts of Xbase are also valid for Xtend. The main focus of Xtend lies in providing a language that is more readable than Java in certain situations. In the background, Xtend compiles to Java code. Thus, it plays perfectly together with Java, e.g. methods declared in Java classes can be called

[35] http://www.eclipse.org/xtend/documentation.html

in Xtend and vice versa. Several concepts of Xtend are especially beneficial when writing code generators.

```
1 def someMethod() {
2   var myQuestion = 'where do we go?'
3   myQuestion = myQuestion.toFirstLower
4   val myAnswer = 42
5   'The answer for question '+myQuestion+' is: '+myAnswer
6 }
```

In Xtend, a method is defined with the `def` keyword. The return type is optional and will be automatically inferred. Only when a method is recursively invoking itself, a return type needs to be specified explicitly. The keyword `var` defines a variable; constant values are defined with `val`. In Xtend - since it is based on Xbase - everything is an expression, meaning that it has a return type. The last expression of a method defines the return value and also the return type of the method.

In line 3 a so-called extension method is used. Recall that the String class in Java does not provide a method `toFirstUpper`. Xtend allows for extending closed types without changing (maybe you already guess where Xtend got its name from). You can easily write your own extension for the String type for example:

```
1   def square(Integer input) {
2     input * input
3   }
4
5   def useExtension() {
6     16.square // returns 256
7   }
```

What Xtend basically does is changing the syntax of how methods are called. Instead of writing `toLastUpper("Hello")`, Xtend always offers the alternative to use the first input parameter as receiver of the method call (`"Hello".toLastUpper`). This results in the syntax being more chained than nested which improves readability. To use methods from another class as extension methods in your class, the field defining an object of the other class needs to be marked with the `extension` keyword. In our scenario we will use dependency injection like the following:

```
1 class MyGenerator implements IGenerator{
2   @Inject extension IJvmModelAssociations
3   ...
4 }
```

This statement simply allows to use the methods declared by the interface `IJvmModelAssociations` in our generator class as extension methods on our objects. Which concrete implementation of the interface is later actually called, is configured in the Guice module.

A further useful feature of Xtend is *polymorphic dispatching*. Using the `dispatch` keyword on multiple methods with identical signatures has the effect that the decision on which method should be called is based on the runtime type of the target object. In contrast, Java binds methods at compile time based on the static type of the target object. Since Xtend compiles to Java, polymorphic dispatching is internally realized by a dispatcher method using a cascade of `instanceof` constructs.

```
1  def dispatch doSomething(Integer input) {
2    // do something with integer
3  }
4
5  def dispatch doSomething(Float input) {
6    // do something with float
7  }
8
9  def useDispatching() {
10   val Number x = 123
11   val Number y = 0.1
12   x.doSomething + y.doSomething
13 }
```

// TODO gescheites beispiel Ã¼berlegen und zwei sÃtze zu schreiben.

Xtend offers the possibility to define *Rich Strings* (also called *templates*) which is especially useful when writing code generators. Rich Strings allow for writing complex Strings with line breaks and indentations without the need for concatenating special characters like `'\n'` or `'\t'`. A rich String construct starts and ends with triple single quotes (´´´). Within such a String code pieces which themself return a String can be inserted (surrounded by guillemots `Â«Â»`). There are also logical structures like `for` loops or `if-else` statements supported:

```
1  def htmlContent(List<String> contents) '''
2    <html>
3      <body>
4        Â«FOR content: contents BEFORE '<p>' SEPARATOR '<br/>' AFTER '</p>'Â»
5          Â«IF content.length > 10Â»
6            Large Content: Â«content.toFirstUpperÂ»
7          Â«ELSEÂ»
8            Small Content: Â«content.toFirstUpperÂ»
```

```
9           Â«ENDIFÂ»
10        Â«ENDFORÂ»
11      </body>
12    </html>'''
```

Note the special keywords in the `FOR` loop declaration: `BEFORE` and `AFTER` will be called once before and after the iteration, but only if the loop will be iterated at least once. With the `SEPARATOR` keyword a string which will be inserted between two iteratins can be specified. Calling the example method with ['hello', 'Some more text'] results in:

```
<html>
  <body>
    <p>
    Small Content: Hello<br/>
    Large Content: Some more text
    </p>
  </body>
</html>
```

Last but not least, Xtend offers a more sophisticated `switch-case` statement than Java does. The `break` statement as known from Java is implicit in Xtend. Furthermore, switching based on Strings and even types is possible:

```
def getTypeName(Number input) {
  switch (input) {
    Integer: "It is an Integer!"
    Float: "It is a Float!"
    default: "It is some other number type."
  }
}
```

Now as you know the basic concepts of Xtend, let's finally start writing the code generator.

3.3 Code Generator

In this section you will learn how to implement the code generator for the target application. For simplicity, the code generator templates are placed in the `org.eclipse.xtext.example.ql` project in a sub-package `generator`. Usually it would be better to create a separate project which contains the generator, since the language is independent from a single target platform. It would be possible to create different code generators for different target platforms, and it would be better to implement each of them as separate projects.

Generator templates in Xtend are implementations of the `IGenerator` interface:

```
package org.eclipse.xtext.generator;

public interface IGenerator {
  /**
   * @param input - the input for which to generate resources
   * @param fsa - file system access to be used to generate files
   */
  public void doGenerate(Resource input, IFileSystemAccess fsa);
}
```

3.3.1 Dispatcher template

The code generator is invoked with a `Resource` instance, which holds a `Questionnaire` instance. We have to generate multiple artifacts for each resource, so it is a common pattern to create a template class which serves as entry point and dispatches to other template classes to create the artifacts. Usually one template per artifact is created.

Create the class `Root.java` in package `org.eclipse.xtext.example.ql.generator`:

```
package org.eclipse.xtext.example.ql.generator;

import javax.inject.Inject;

import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.xtext.generator.IFileSystemAccess;
import org.eclipse.xtext.generator.IGenerator;
import org.eclipse.xtext.xbase.compiler.JvmModelGenerator;

@SuppressWarnings("restriction")
public class Root implements IGenerator {
  @Inject
  JvmModelGenerator jvmModelGenerator;

  public void doGenerate(Resource input, IFileSystemAccess fsa) {
    // dispatch to other generators
    jvmModelGenerator.doGenerate(input, fsa);
  }
}
```

As a first generator to which is dispatched, we inject an instance of `JvmModelGenerator`. This is a standard generator shipped with Xtext which translates types inferred by the Jvm

Model Inferrer to Java classes. In our case, the Java class for Forms are generated by the `JvmModelGenerator`. In JSF terms, we speek of the *Backing Bean*.

Next, Xtext has to know that `Root` is the template that has to be invoked as generator implementation. This has to be configured - you guessed right! - with Guice again. We need to add a configuration that binds the `IGenerator` interface to the `Root` class.

Open class `QlDslRuntimeModule` and add this method:

```
1  @Override
2  public Class<? extends IGenerator> bindIGenerator() {
3    return Root.class;
4  }
```

Now we are ready to add additional templates and register them in the `Root` class.

3.3.2 JSF Generator

After creation of the `Root` class where we easily can add new Generators, we use the *New Xtend Class Wizard* to create a new Xtend class called `JSFGenerator.xtend` in package `org.eclipse.xtext.example.ql.generator`.

This class will be our entry point to generate JSF related artifacts.

The *New Xtend Class Wizard* provides the possibility to bind interfaces to the new class by use of the `Add` button near the interface section. As we want to create a new generator we add the interface `org.eclipse.xtext.generator.IGenerator` to our new Xtend class. After typing in the package, the name and the interface of our new Xtend class as shown in figure 1, we can finish the wizard so that the class shown in listing 1 will be created in our project.

```
1  package org.eclipse.xtext.example.ql.generator
2
3  import org.eclipse.xtext.generator.IGenerator
4  import org.eclipse.emf.ecore.resource.Resource
5  import org.eclipse.xtext.generator.IFileSystemAccess
6
7  class JSFGenerator implements IGenerator {
8
9    override doGenerate(Resource input, IFileSystemAccess fsa) {
10     throw new UnsupportedOperationException("TODO: auto-generated method stub")
11   }
12 }
```

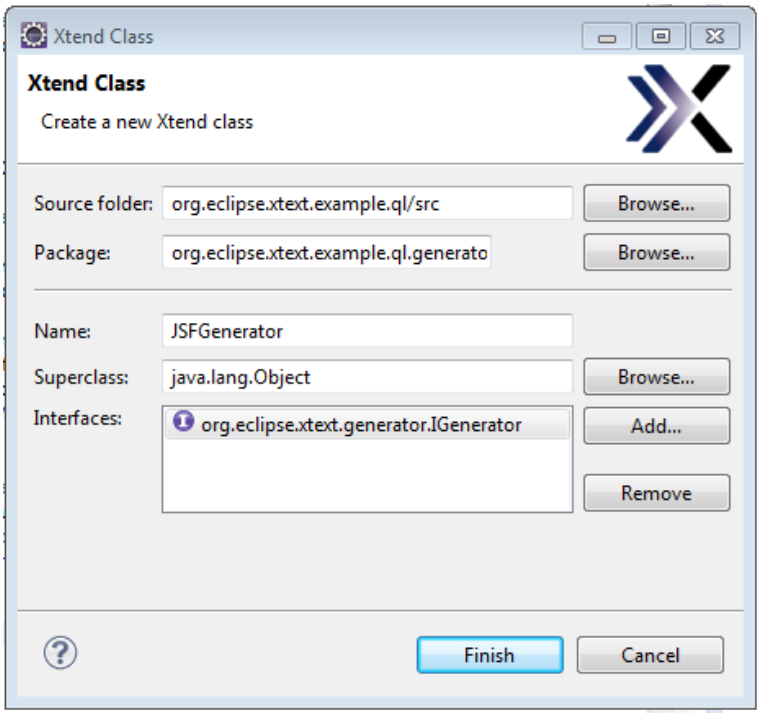


Abbildung 1: New Xtend Class Wizard

Listing 1: New Xtend Class

- show whole generater - explain step by step

3.3.3 OutputConfigurationProvider

//TODO explain OutputConfigurationProvider

```
1  package org.eclipse.xtext.example.ql.generator;
2
3  import java.util.Set;
4
5  import org.eclipse.xtext.generator.OutputConfiguration;
6  import org.eclipse.xtext.generator.OutputConfigurationProvider;
7
8  public class JsfOutputConfigurationProvider extends OutputConfigurationProvider {
9
10   public final String WEB_CONTENT = "WebContent";
11
12   /**
13    * @return a set of {@link OutputConfiguration} available for the generator
14    */
```

```
15   public Set<OutputConfiguration> getOutputConfigurations() {
16     Set<OutputConfiguration> outputConfigurations = super
17         .getOutputConfigurations();
18
19     OutputConfiguration webContent = new OutputConfiguration(WEB_CONTENT);
20     webContent.setDescription("Read-only Output Folder for forms");
21     webContent.setOutputDirectory("./WebContent");
22     webContent.setOverrideExistingResources(true);
23     webContent.setCreateOutputDirectory(true);
24     webContent.setCleanUpDerivedResources(true);
25     webContent.setSetDerivedProperty(true);
26     outputConfigurations.add(webContent);
27
28     return outputConfigurations;
29   }
30 }
```

Listing 2: JSFOutputConfigurationProvider

//TODO binding the OutputConfigurationProvider in QlDslRuntimeModule

```
1 @Override public void configure(Binder binder) {
2     super.configure(binder);
3     binder.bind(IOutputConfigurationProvider.class)
4         .to(JSFOutputConfigurationProvider.class).in(Singleton.class);
5   }
```

Listing 3: Binding JSFOutputConfigurationProvider

3.4 Testing the Questionnaire Application

- describe how the application can be started, show some screenshots

4 Layout and Styling Language (QLS)

4.1 The Language QLS

In this chapter we will describe how the optional task to define a language for styling and layout information can be accomplished in our language workbench. The language QLS should allow for defining the following information:

- Grouping of question forms into pages, sections and subsections
- Navigation links between pages
- Styling of question texts by defining font style, font type and color
- Defining the appearance of a question by specifying the widget type to render

The following shows an example model that we want to be able to define with QLS:

```
1  page HouseOwningPage {
2    section house uses Box1HouseOwning {
3      question hasBoughtHouse [widget: CheckBox]
4      question valueResidue [
5        font-style: "italic"
6        font-weight: "bold"
7        font-color: "#2233FF"
8        font-family: "Arial"
9      ]
10   }
11   section garage uses GarageOwning {
12     question hasBoughtGarage [widget: Radio["Yepp", "Nope"]]
13   }
14   navigation {
15     CarOwningPage
16   }
17 }
18
19 page CarOwningPage uses CarOwning {
20   // ...
21 }
```

This example model defines two pages. The first page consists of two sections, one for the question form `Box1HouseOwning` which was defined in chapter 2 and one for a further form called `GarageOwning`. The form to be rendered inside a page or a section is specified by the `uses` keyword. This definition must be unambiguously, e.g. if there is a form included by a page, it is not allowed to refer to another form in one of its containing sections. This restriction is ensured by implementing a corresponding validation which will be covered in chapter (TODO). In a section (or page) the styling information for the questions of the included form can be defined as in lines 3-9 and 13. The `navigation` keyword allows to define the order in which the pages are to be displayed by specifying which page should appear next.

Before we can write the corresponding Xtext grammar, we need to create the DSL projects for QLS as we have done for the Questionnaire language. For this, refer again to chapter 2.1 and

replace all occurences of `ql` with `qls`. The project `org.eclipse.xtext.example.qls` should then contain the grammar file `QlsDsl.xtext`. We define the content of this file as the following:

```
grammar org.eclipse.xtext.example.qls.QlsDsl with org.eclipse.xtext.common.Terminals

generate qlsDsl "http://www.eclipse.org/xtext/example/qls/QlsDsl"
import "http://www.eclipse.org/xtext/example/ql/QlDsl" as ql

QuestionnaireStyleModel:
  pages+=Page*;

Page:
  "page" name=ID ("uses" form=[ql::Form|ID])? "{"
    element+=PageElement*
    navigation=Navigation?
  "}"
;

PageElement:
  QuestionStyling | Section
;

QuestionStyling:
  "question" question=[ql::Question] styling+=StyleInformation?
;

StyleInformation: {StyleInformation}
  "["(
    ("font-style:" fontStyle=STRING)? &
    ("font-weight:" fontWeight=STRING)? &
    ("font-color:" fontColor=STRING)? &
    ("font-family:" fontFamily=STRING)? &
    ("widget:" widget=Widget)?
  )"]"
;

Widget: {Widget}
  widgetType=("Radio"|"DropDown"|"CheckBox"|"Text"|"Slider") ("[" labels+=STRING ("," labels
      +=STRING)* "]")?
;

Section:
  "section" name=ID ("uses" form=[ql::Form|ID])? "{"
    element+=PageElement*
  "}"
```

```
;

Navigation: {Navigation}
  "navigation" "{" (nextPage+=[Page|ID])+ "}"
;
```

After reading chapter 2.1 you should be familiar with the concepts of Xtext grammar definitions and understand most parts of the QLS grammar. One new concept represented in the QLS grammar is the one of unordered lists:

```
StyleInformation: {StyleInformation}
  "["(
    ("font-style:" fontStyle=STRING)? &
    ("font-weight:" fontWeight=STRING)? &
    ("font-color:" fontColor=STRING)? &
    ("font-family:" fontFamily=STRING)? &
    ("widget:" widget=Widget)?
  )"]"
;
```

The styling information can be defined in an arbitrary order in which each kind of element (font style, font color and so on) may only occur at most once. The arbitrariness of the order is expressed by the `'&'` between the style elements. The optionality of each style information is again declared by the question mark `'?'`.

A further noteworthy aspect is the handling of references. In Xtext, references to language elements are expressed by using squared brackets. An example for this are the references to existing pages in the navigation section:

```
Navigation: {Navigation}
  "navigation" "{" (nextPage+=[Page|ID])+ "}"
;
```

Here, `nextPage` is a reference to an existing page description which may even be defined in a different file. `ID` defines which attribute should be used for the reference's name. Xtext automatically creates hyperlinks to the referenced elements which can be enabled by holding `Ctrl` and moving the curser over the reference or by pressing `F3`.

To define references to elements defined in other langauges than the own, the reference needs to be qualified. An example in QLS is the reference to questions or forms defined in a QL model. To express this on the grammar level, the QL DSL needs first to be imported:

```
import "http://www.eclipse.org/xtext/example/ql/QlDsl" as ql
```

Since the QL DSL is imported under the name `ql`, references can now be qualified by using a double colon (`::`):

```
QuestionStyling:
  "question" question=[ql::Question] styling+=StyleInformation?
;
```

This grammar rule allows for refering all question elements defined in any QL model in our test project. However, since there is always an unambiguous form specified for a page or a section, only the questions defined within this form should be referable. This is a typical scoping issue which needs to be handled in the scope provider of the QLS language. We already learned about scoping in section 2.8. The scope provider for the QLS language looks like the following:

```
public class QlsDslScopeProvider extends AbstractDeclarativeScopeProvider {

  public IScope scope_QuestionStyling_question(EObject context, EReference reference) {
    Form usedForm = getUsedForm(context);
    List<Question> allQuestions = EcoreUtil2.getAllContentsOfType(usedForm,
        Question.class);
    return Scopes.scopeFor(allQuestions);
  }

  private Form getUsedForm(EObject context) {
    if (context instanceof Section) {
      Section section = (Section) context;
      if (section.getForm() != null) {
        return section.getForm();
      }
    } else if (context instanceof Page) {
      Page page = (Page) context;
      if (page.getForm() != null) {
        return page.getForm();
      }
    }
    if (context != null) {
      return getUsedForm(context.eContainer());
    }
    return null;
  }
}
```

The method `scope_QuestionStyling_question` is always invoked whenever the visible elements for the attribute `question` in the grammar rule `QuestionStyling` need to be computed. Here, we just use a recursive algorithm to get the form declaration of the parent section or page

which is mandatory. With the helper method `getAllContentsOfType` all questions defined in this form are collected and a scope containing these questions is returned.

Now it's time to play around with the QLS language and to test its features. For this, switch to the test project in the runtime environment (see also section 2.4) and create a file with the file extension `.qls`.

4.2 QLS Code Generator

So far, our QLS models have no effect on the generated output. The first step is to think about which artifacts are to be generated from a qls model. The QLS model contains two kinds of information. The first is layout information: A page consists of one or more forms. Recall that for each form two xhtml files are already generated, one for the base content of the form site and a wrapper referencing this base xhtml. Hence a page maps to an xhtml file which is composed of all specified forms. The second information in QLS is styling information. As usual in web development, styling is expressed in the CSS format in our tool. To sum up:

For each page:

- Generate one CSS file
- Generate one Xhtml file wiring all form Xhtmls together and reference css file

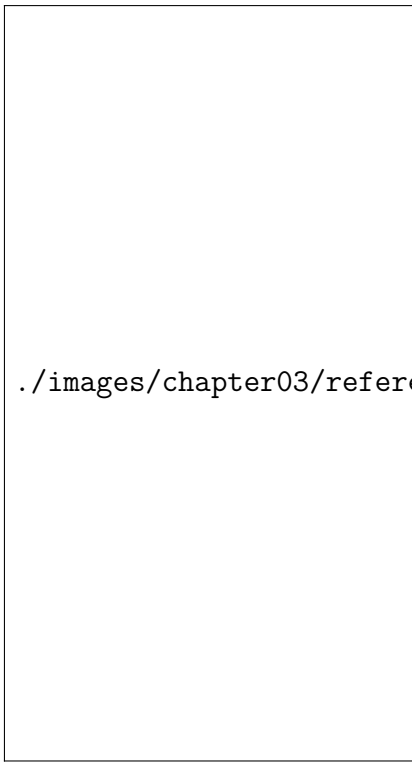
Let's take the following QLS model as reference example:

```
page HouseOwningPage {
  section house uses HouseOwning {
    question hasBoughtHouse [font-style:"italic"]
    question valueResidue [
      font-weight: "bold"
      font-color: "#2233FF"
      font-family: "Verdana"
    ]
  }
  section garage uses GarageOwning {
    question hasBoughtGarage [widget: Radio["Yepp", "Nope"]]
  }
  navigation {
    CarOwningPage
  }
}

page CarOwningPage uses CarOwning {
```

```
  question hasSoldCar [font-color: "green"]
  question hasBoughtCar [font-color: "red"]
}
```

The intended generated file structure is visualized in the following screenshot:



// TODO screenshot, visualize what is generated by QL and what by QLS

The page *HouseOwningPage* contains two sections using the forms *HouseOwning* and *GarageOwning*, thus the generated file *pages/HouseOwningPage.xhtml* composes the two base files *forms/HouseOwningBase.xhtml* and *forms/GarageOwningBase.xhtml* (generated by the QL code generator) together:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <h:head></h:head>
  <ui:composition template="/index.xhtml">
    <ui:define name="content">
      <h:outputStylesheet library="default/css/generated" name="HouseOwningPage.css" />
      <div><ui:include src="/generated/forms/HouseOwningBase.xhtml" /></div><p/>
      <div><ui:include src="/generated/forms/GarageOwningBase.xhtml" /></div>

      <div>
      <h:outputLink value="CarOwningPage.jsf">CarOwningPage</h:outputLink>
    </div>
```

```
15      </ui:define>
16    </ui:composition>
17  </html>
```

The two pages are referenced in lines 9 and 10. Line 13 defines the link to the next page as defined in the `navigation` section of the QLS model. Line 8 specifies which CSS file is to be used to render the site's elements. The CSS file for the house owning page contains all corresponding styling information which are linked to the corresponding label elements by their ids:

```
1  #houseowning\:lblHasBoughtHouse {
2    font-style: italic;
3  }
4  #houseowning\:lblValueResidue {
5    color: #2233FF;
6    font-family: Verdana;
7    font-weight: bold;
8  }
9  #garageowning\:lblHasBoughtGarage {
10 }
```

When JSF converts from Xhtml each element gets a unique (full qualified) id. In out scenario this is always the id of the parent form concatenated with the id of the elment itself. As separator JSF uses a colon. However, colons are special characters in CSS, hence they need to be escaped.

Now that the intended artifacts to be generated are clearified, the code generator itself can be written. Here again we use Xtend and some of its language concepts:

```
1  class QlsDslGenerator implements IGenerator {
2    @Inject extension JsfOutputConfigurationProvider
3    @Inject extension JSFGenerator
4
5    override void doGenerate(Resource input, IFileSystemAccess fsa) {
6    if (input.URI.fileExtension!="qls")
7        return
8
9      val styleModel = input.contents.head as QuestionnaireStyleModel
10     for (page: styleModel.pages) {
11       val cssContent = generateCssFile(page);
12       val cssFileName = "resources/default/css/generated/"+page.name+".css"
13       fsa.generateFile(cssFileName, WEB_CONTENT, cssContent)
14
15       val xhtmlContent = generateXhtmlFile(page);
16       val xhtmlFileName = "generated/pages/"+page.name+".xhtml"
```

```
17      fsa.generateFile(xhtmlFileName, WEB_CONTENT, xhtmlContent)
18    }
19    val contentIndex = generateIndexPage(styleModel.pages.get(0))
20    fsa.generateFile("generated/pages/index.xhtml",WEB_CONTENT, contentIndex)
21  }
22
23  def generateCssFile(Page page) '''
24    <!-- @generated -->
25    Â«FOR styleInfo: page.eAllContents.filter(typeof(StyleInformation)).toListÂ»
26    Â«styleInfo.idÂ» {
27      Â«IF styleInfo.fontColor != nullÂ»color: Â«styleInfo.fontColorÂ»;Â«ENDIFÂ»
28      Â«IF styleInfo.fontFamily != nullÂ»font-family: Â«styleInfo.fontFamilyÂ»;Â«ENDIFÂ»
29      Â«IF styleInfo.fontStyle != nullÂ»font-style: Â«styleInfo.fontStyleÂ»;Â«ENDIFÂ»
30      Â«IF styleInfo.fontWeight != nullÂ»font-weight: Â«styleInfo.fontWeightÂ»;Â«ENDIFÂ»
31    }
32   Â«ENDFORÂ»
33  '''
34
35  def generateXhtmlFile(Page page) '''
36  <!-- @generated -->
37  <html xmlns="http://www.w3.org/1999/xhtml"
38        xmlns:ui="http://java.sun.com/jsf/facelets"
39        xmlns:h="http://java.sun.com/jsf/html"
40        xmlns:f="http://java.sun.com/jsf/core">
41    <h:head></h:head>
42    <ui:composition template="/index.xhtml">
43      <ui:define name="content">
44        <h:outputStylesheet library="default/css/generated" name="Â«page.nameÂ».css" />
45        Â«IF page.form != nullÂ»
46        <div><ui:include src="/generated/forms/Â«page.form.nameÂ»Base.xhtml" /></div>
47        Â«ELSEÂ»
48          Â«FOR section: page.eAllContents.toList.filter(typeof(Section)).toList SEPARATOR '<
              p/>'Â»
49          <div><ui:include src="/generated/forms/Â«section.form.nameÂ»Base.xhtml" /></div>
50          Â«ENDFORÂ»
51        Â«ENDIFÂ»
52
53        <div>
54        Â«IF page.navigation != nullÂ»
55          Â«FOR nextPage: page.navigation.nextPageÂ»
56          <h:outputLink value="Â«nextPage.nameÂ».jsf">Â«nextPage.nameÂ»</h:outputLink>
57          Â«ENDFORÂ»
58        Â«ENDIFÂ»
59      </div>
60      </ui:define>
```

```
61      </ui:composition>
62    </html>
63    '''
64
65    def generateIndexPage(Page page)'''
66    <?xml version='1.0' encoding='UTF-8' ?>
67    <!-- @generated -->
68    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
          xhtml1/DTD/xhtml1-transitional.dtd">
69    <html xmlns="http://www.w3.org/1999/xhtml"
70          xmlns:h="http://java.sun.com/jsf/html"
71          xmlns:ui="http://java.sun.com/jsf/facelets">
72      <ui:composition template="/index.xhtml">
73        <ui:define name="content">
74          <h:outputLink value="Â«page.nameÂ».jsf">Â«page.nameÂ»</h:outputLink>
75        </ui:define>
76      </ui:composition>
77    </html>
78    '''
79
80    def getId(StyleInformation styleInfo) {
81    val question = (styleInfo.eContainer as QuestionStyling).question
82    val form = EcoreUtil2::getContainerOfType(question, typeof(Form))
83    "#"+form.id+ "\\:lbl"+question.id.toFirstUpper
84    }
85
86    def dispatch getForm(Section section) {
87    if (section.form != null) {
88      section.form
89    }
90    else {
91      section.eContainer.form
92    }
93    }
94
95    def dispatch getForm(Page page) {
96    page.form
97    }
98  }
```

5 Additional Concepts

5.1 Validation

As an optional task the LWC13 task requires the implementation of analysis rules. Xtext provides a validation framework which integrates into the EMF Validation framework[36]. The Xtext User Manual contains a Validation chapter that is worth reading additionally[37].

We will show in this section how the constraints defined in the task description can be realized with Xtext.

5.1.1 Extending the Java Validator class

With the first translation of the Xtext grammar, the generator has already created the necessary infrastructure to implement custom validation rules. Look into the package `org.eclipse.xtext.exampl` you will find a Java class `QlDslJavaValidator`. The class extends the `AbstractQlDslJavaValidator` class, which is regenerated each time the grammar is translated. Thus, the generation gap pattern is applied here again. It is safe to extend the `QlDslJavaValidator` class manually.

But instead of implementing the constraints in Java, we will use Xtend again. For easier integration, our Xtend based validator class will be inserted into the class hierarchy of QlDslJavaValidator.

Create an Xtend class `QlDslXtendValidator`, and extend it from `AbstractQlDslJavaValidator`.

```
1  package org.eclipse.xtext.example.ql.validation
2
3  import javax.inject.Inject
4  import org.eclipse.xtext.validation.Check
5  import org.eclipse.xtext.xbase.XFeatureCall
6  import org.eclipse.xtext.xbase.XbasePackage
7  import org.eclipse.xtext.xbase.jvmmodel.IJvmModelAssociations
8
9  import static extension org.eclipse.xtext.nodemodel.util.NodeModelUtils.*
10 class QlDslXtendValidator extends AbstractQlDslJavaValidator {
11   @Inject extension IJvmModelAssociations
12 }
```

[36] http://www.eclipse.org/modeling/emf/?project=validation
[37] see http://www.eclipse.org/Xtext/documentation.html#validation

Derive QlDslJavaValidator from the the Xtend validator:

```
public class QlDslJavaValidator extends QlDslXtendValidator {
  // do nothing here, rules are implemented in Xtend
}
```

5.1.2 Constraint: Ensure order of questions

The first constraint in the LWC13 task is defined so:

> *Test for cyclic dependencies. For instance, the following snippet should be rejected:*
>
> ```
> if (x) { y: "Y?" boolean }
> if (y) { x: "X?" boolean }
> ```
>
> *The reason is that y will only be asked for when x is true, but x will only get a value when y is true. Of course such cyclic dependencies could occur transitively and nested in expressions. Another way of stating this check is: the ordering of questions should be consistent with how the question variables are used in conditions and computed values.*

The task allows two approaches to achieve the goal. We will choose the second approach: Check that elements referred in expressions have been declared before their usage.

Xtext does not enforce that elements are declared before they are used somewhere. The referred names simply must be in the scope of the context, which the current scope implementation already accomplishes. We could implement this constraint also in the scope provider of the language by restricting the scope to elements that have been declared before. However, we will implement this as a semantic constraint in the validator class.

To implement the constraint we need to know the location in the model where a Question element is declared and where it is called in an expression. Besides the Abstract Syntax Tree Xtext maintains a second model, which represents the document. This is the so-called *Node Model*, and the class `NodeModelUtils` provides some utility functions to navigate from an AST element to the node model. Since Questions that are referred in expressions are local to the Form, we can simply compare the offset in the document of the declared Question and the feature call in an expression.

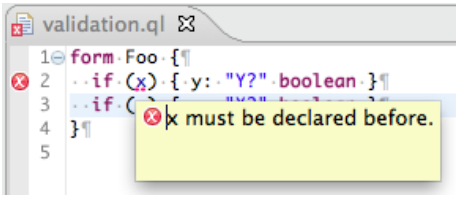
Xtext validation rules are implemented as methods which are annotated with `@Check`. The method name does not matter. Check methods are expected to have exactly one parameter,

which is of the type of element that has to be checked. The base class provides methods to create error messages. For the case of this constraint, the necessary context object type is `XFeatureCall`.

Now open the `QlDslXtendValidator` class again and add this method[38]:

```
1  @Check
2  def void check_featureDeclaredBeforeCall (XFeatureCall featureCall) {
3    val featureSource = featureCall.feature.sourceElements.head
4    val nodeFeature = if (featureSource != null) featureSource.node else featureCall.feature
         .node
5    val nodeCall = featureCall.node
6    if (nodeFeature != null) {
7      if (nodeFeature.offset > nodeCall.offset) {
8        error(featureCall.feature.simpleName+" must be declared before.",featureCall,
9        XbasePackage::eINSTANCE.XAbstractFeatureCall_Feature, "
             ERR_FEATURE_CALL_BEFORE_DECLARATION", null)
10     }
11   }
12 }
```

After restarting the workbench the situation will be recognized as an error:



5.1.3 Constraint: Type conformance check

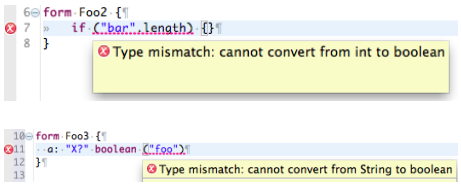
Next, the LWC13 task requires checking the type conformance in expressions:

> *Type check conditions and variables: the expressions in conditions should be type correct and should ultimately be booleans. The assigned variables should be assigned consistently: each assignment should use the same type.*

Here we have to do nothing, since this constraint is already implemented in Xbase. This works thanks to Xbase's type inference mechanism. The Xtend language makes heavy use of this nice feature, which makes it almost unneccessary to declare types anywhere. For dynamic languages

[38] https://gist.github.com/kthoms/5240455

this is natural, since the actual type is known and evaluated at runtime. Static typed language often lack this feature.



5.1.4 Testing validation rules

It is easy to test validation rules in a runtime environment. However, we will show how these rules can also be unit tested. Also herefore the Xtext framework already contains the necessary infrastructure. Remember that Xtext has already created a test plugin with the initial generator run? Now it is time to make use of it.

Again, it is easier to create the test with Xtend. Xtend allows us to create simple models inline with Rich Strings, pass the result to a parser, and validate the result. With Xtend this is a one-liner.

```
1  package org.eclipse.xtext.example.ql.validation.test
2
3  import javax.inject.Inject
4  import org.eclipse.xtext.example.ql.QlDslInjectorProvider
5  import org.eclipse.xtext.example.ql.qlDsl.Questionnaire
6  import org.eclipse.xtext.junit4.InjectWith
7  import org.eclipse.xtext.junit4.XtextRunner
8  import org.eclipse.xtext.junit4.util.ParseHelper
9  import org.eclipse.xtext.junit4.validation.ValidationTestHelper
10 import org.eclipse.xtext.xbase.XbasePackage
11 import org.junit.Before
12 import org.junit.Test
13 import org.junit.runner.RunWith
14
15 @RunWith(typeof(XtextRunner))
16 @InjectWith(typeof(QlDslInjectorProvider))
17 class QlDslValidationTest {
18   @Inject extension ParseHelper<Questionnaire> parseHelper
19   @Inject extension ValidationTestHelper
20
21   @Before
22   def void setUp () {
23     parseHelper.fileExtension="ql"
24   }
```

```
25
26  @Test
27  def void testValidation_CallBeforeDeclaration_expectError () {
28    '''
29    form Foo {
30      if (x) { y: "Y?" boolean }
31      if (y) { x: "X?" boolean }
32    }
33    '''.parse.assertError(XbasePackage::eINSTANCE.XFeatureCall, "
          ERR_FEATURE_CALL_BEFORE_DECLARATION", "must be declared before")
34  }
35
36  @Test
37  def void testValidation_CallBeforeDeclaration_expectSuccess () {
38    '''
39    form Foo {
40      x: "foo" boolean
41      if (x) { a: "X?" boolean }
42    }
43    '''.parse.assertNoErrors
44  }
45
46
47  // Type check conditions and variables: the expressions in conditions should be type
        correct and should ultimately be booleans.
48  // The assigned variables should be assigned consistently: each assignment should use the
        same type.
49  @Test
50  def void testValidation_ConditionTypeCheck_expectError () {
51    '''
52    form Foo {
53      if ("foo".length) { a: "X?" boolean }
54    }
55    '''.parse.assertError(XbasePackage::eINSTANCE.XMemberFeatureCall,
56       "org.eclipse.xtext.xbase.validation.IssueCodes.incompatible_types", "Type mismatch")
57  }
58
59  @Test
60  def void testValidation_ConditionTypeCheck_expectSuccess () {
61    '''
62    form Foo {
63      if ("foo".length>1) { a: "X?" boolean }
64    }
65    '''.parse.assertNoErrors
66  }
```

```
67
68    @Test
69    def void testValidation_AssignmentTypeCheck_expectFailure () {
70      '''
71      form Foo {
72        a: "X?" boolean ("foo".length)
73      }
74      '''.parse.assertError(XbasePackage::eINSTANCE.XMemberFeatureCall,
75         "org.eclipse.xtext.xbase.validation.IssueCodes.incompatible_types", "Type mismatch")
76    }
77
78    @Test
79    def void testValidation_AssignmentTypeCheck_expectSuccess () {
80      '''
81      form Foo {
82        a: "X?" boolean ("foo".length>1)
83      }
84      '''.parse.assertNoErrors
85    }
86  }
```

Basically, the class is a plain JUnit 4 class. We have to use a special JUnit execution class, `XtextRunner`, and provide a language specific initializer class, `QlDslInjectorProvider`.

Next, the class adds two extension classes. ParseHelper provides a parse() method for char sequences. This will parse and validate the model. Afterwards the observed issues can be asserted with the methods from the

```
1  @RunWith(typeof(XtextRunner))
2  @InjectWith(typeof(QlDslInjectorProvider))
3  class QlDslValidationTest {
4    @Inject extension ParseHelper<Questionnaire> parseHelper
5    @Inject extension ValidationTestHelper
6    ...
7  }
```

Now the test methods can be implemented. They are super simple:

```
1    @Test
2    def void testValidation_CallBeforeDeclaration_expectSuccess () {
3      '''
4      form Foo {
5        x: "foo" boolean
6        if (x) { a: "X?" boolean }
7      }
```

```
8     '''.parse.assertNoErrors
9   }
```

Due to the tight Java integration, the unit tests of the Xtend class can be executed by running them through the context menu (*Run As / JUnit Test*).