Language Workbench Challenge 2013
Xtext Submission

Version: 1.0 DRAFT - 11. 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 Houlshould* 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 2.3.1, which is at the moment of writing the latest official release. Xtext 2.4 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

An example for DSL is one that 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"

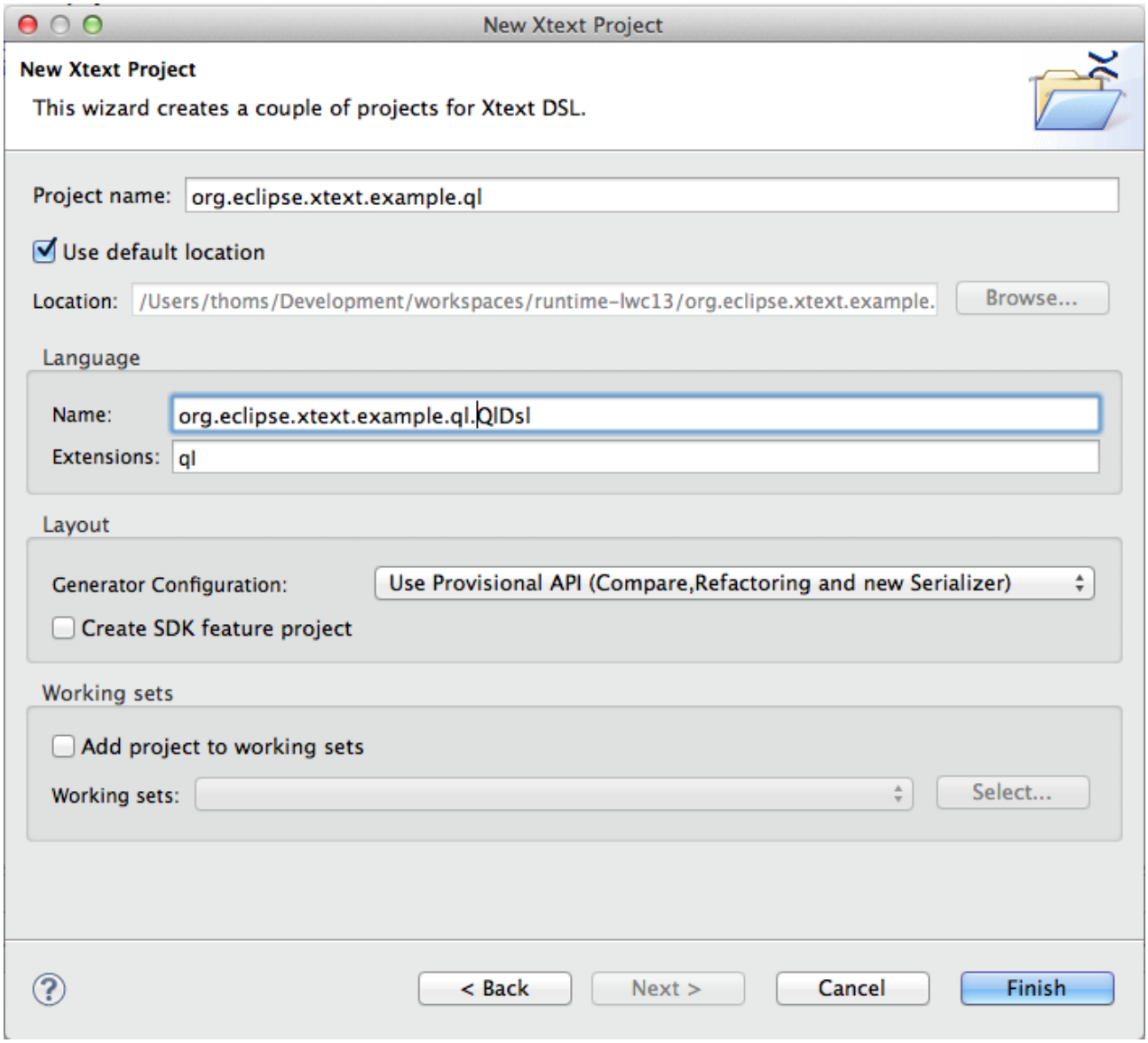
This will allow you re-running the previous launched application by just pressing the Run or Debug button in the Eclipse toolbar, or using keyboard shortcuts. The default settings does not always do what you want.

1.6 The Survey Application

2 Developing the Questionnaire Language

2.1 Create the DSL Projects

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



On the project wizard page enter:

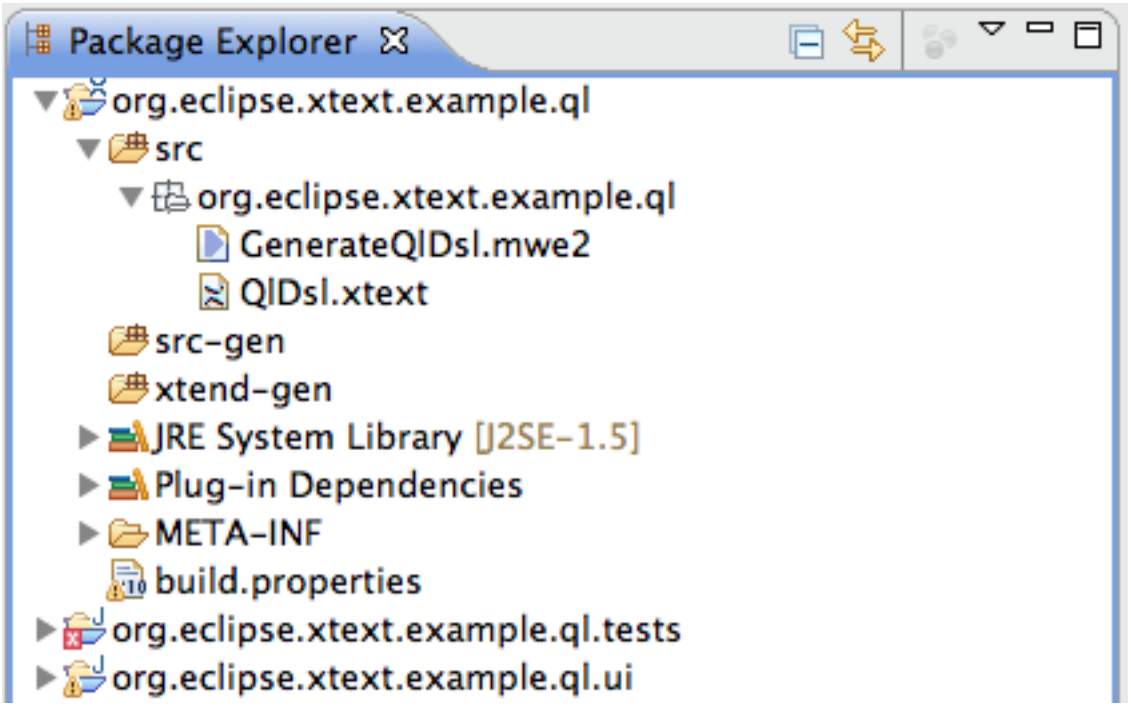
1. Project name: `org.eclipse.xtext.example.ql`. Xtext will create multiple projects, which share this prefix. It is a convention to use a lowercase, dot-separated name.

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 won’t deal testing of DSLs in this tutorial any further. You can close or remove 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. 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.

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[13]:

```
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:

[13] 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`[14]. 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[15]. 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

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

[15] 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 [16].

```
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.[17] 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   "}";
```

[16] http://www.eclipse.org/Xtext/documentation.html#grammarMixins
[17] 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[18]: 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:

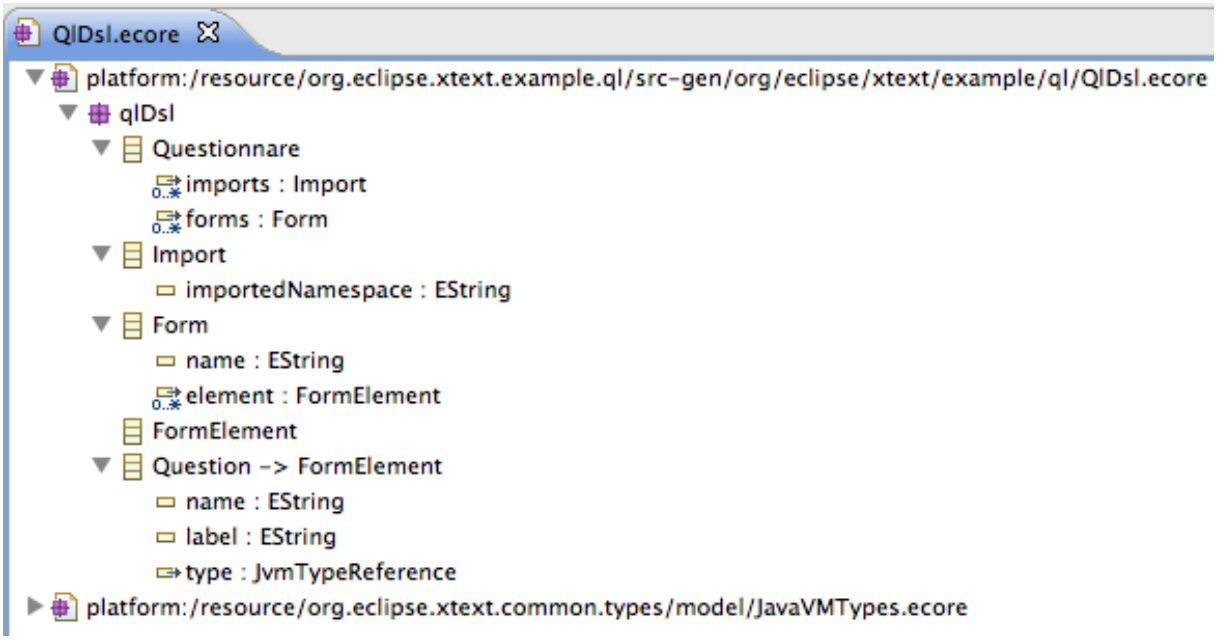
Runtime Project - folder `src`

- The class `QlDslRuntimeModule` is a Guice configuration. Guice[19] is a famous Dependency Injection[20] 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`

[18] http://heikobehrens.net/2009/04/23/generation-gap-pattern/
[19] http://code.google.com/p/google-guice/
[20] 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`
- The package `org.eclipse.xtext.example.ql.parser.antlr.internal` contains an ANTLR3[21] grammar and the Lexer and Parser classes generated from it.

2.4 Testing the Questionnaire Language

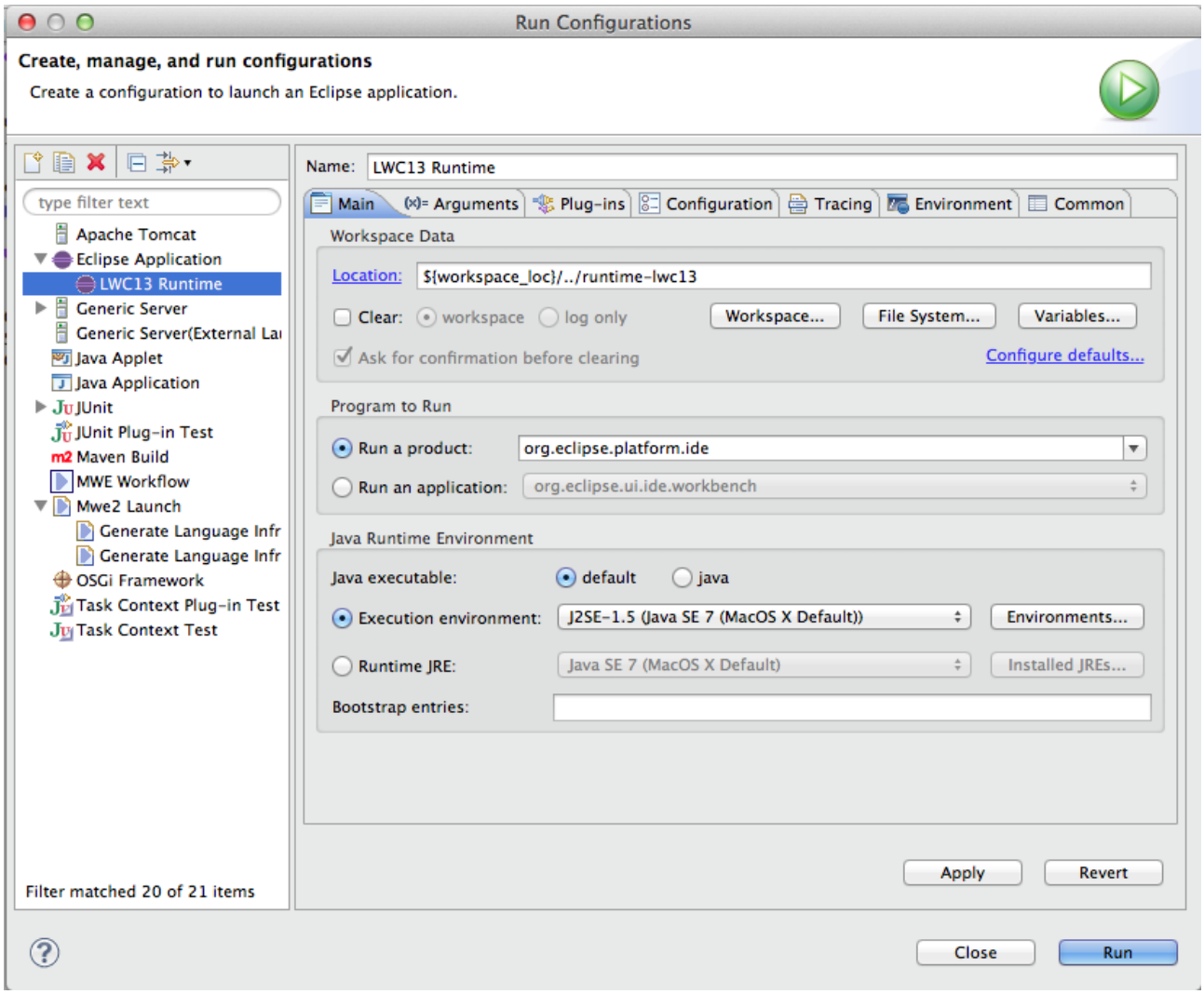
2.4.1 Creating a Launch Configuration

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.

[21] 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.

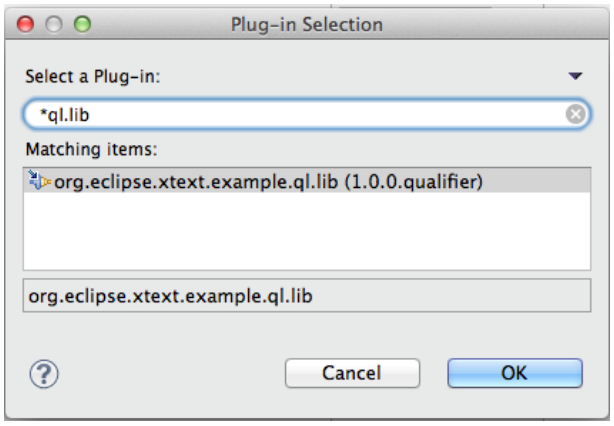
Now press the *"Run"* button. Another Eclipse instance will start with an empty workspace. Close the Welcome window.

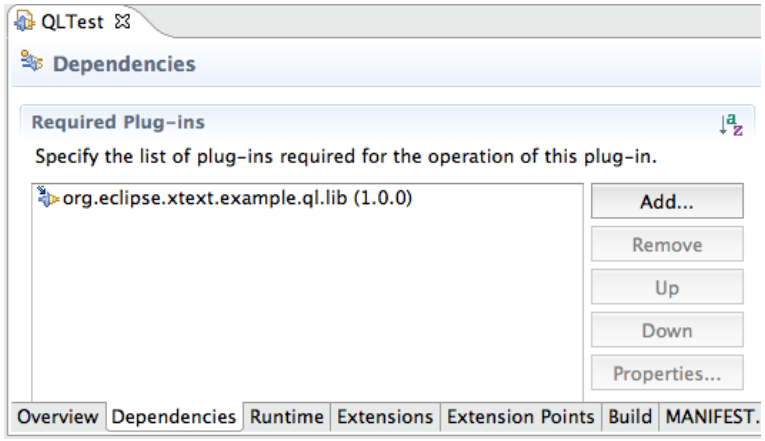
2.4.2 Create Test Project

In the Runtime Workspace create a new Plug-in Project with name "QLTest".[22]

[22] 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`.

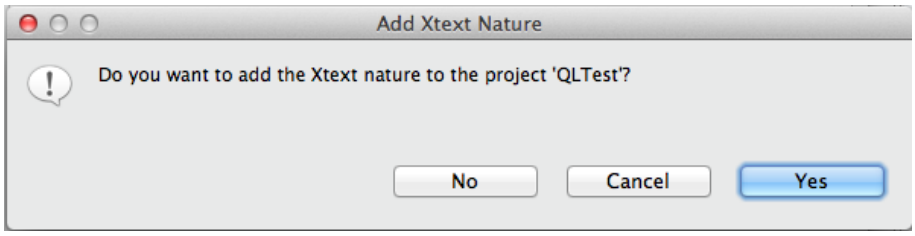




The DSL has to support custom datatypes like `Money`, which must be defined. Select the `/src` folder and create a new Java class `Money` in package `types`:

```
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"`[23]:

```
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 [24] 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

[23] https://gist.github.com/kthoms/5036304
[24] 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[25] 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
```

[25] 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: [26]

```
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.
```

[26] 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 element are again `FormElements`, 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. [27]

Xbase comes out of the box with the support for standard Java types like 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:[28]

```
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 }
```

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

[28] 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, when 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 Question Group a method is produced that computes whether the group is visible.

Now place the content into the inferrer class[29] :

```
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        it.members += toField("serialVersionUID",typeReferences.getTypeForName("long",
              element),[final = false ^static = false ])
28
29        val allQuestions = form.eAllContents.filter(typeof(Question)).toList
30
31        for (question: allQuestions) {
32          members += question.toField(question.name, question.type)
33        }
34
```

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

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

Now lets take a deeper look at the implementation:

```
class QlDslJvmModelInferrer extends AbstractModelInferrer {
  @Inject extension JvmTypesBuilder
  @Inject TypeReferences typeReferences
  def dispatch void infer(Questionnaire element, IJvmDeclaredTypeAcceptor acceptor, boolean
      isPreIndexingPhase) {
    ...
  }
}
```

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 available 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 JvmModel instances for given qualified Java class names through its getTypeForName() methods.

```
  for (form: element.forms) {
    acceptor.accept(form.toClass("forms."+form.name))
    .initializeLater[
      ...
    ]
  }
```

We loop over the Form instances of the Questionnaire elem and derive a Class instance for them 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 it.members += toField("serialVersionUID",typeReferences.getTypeForName("long",element),[
    final = false ^static = false ])
```

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.

```
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 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 simpley 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 (the text between the three single quotes "') feature, 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() method 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.

3 Developing the Code Generator

3.1 Reference Implementation

- describe how the final application looks like - of which files does it consist, how do the play together, and what needs to be generated

3.2 Xtend

- describe relevant concepts of xtend, like polymorphic dispatching, rich strings etc.

3.3 Code Generator

- describe step by step how the code generator is developed

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

- QLS example - QLS grammar

4.2 Adapt the Code Generator

- describe which parts of the code generator need to be adapted to use the QLS features