Language Workbench Challenge 2013
Xtext Submission

Version: 1.0 DRAFT - 12. Februar 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 **5**
- 1.1 Task Description . . . 5
- 1.2 Technology Stack . . . 5
- 1.3 Xtext Overview . . . 5
- 1.4 Installing Eclipse and Xtext . . . 8
- 1.5 Workspace Setup . . . 8
- 1.6 The Survey Application . . . 9

2 Developing the Questionnaire Language **9**
- 2.1 Create the DSL Projects . . . 9
- 2.2 Defining the Grammar . . . 12

3 Testing the Questionnaire Language **15**

4 Xbase **15**

5 Including Expressions into the QL Language **15**

6 JVM Model Inferrence **15**

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 developped here will also be based on Xbase, but more on this later.

1.3 Xtext Overview

This overview will give you a rough idea about what Xtext[7] 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

[6] http://wiki.eclipse.org/Kepler/Simultaneous_Release_Plan
[7] http://www.xtext.org

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.

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:

```
1 entity Person {
2   name : String
3   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, overview 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.[8]
4. Start Eclipse by running the `eclipse` executable in the newly-created `eclipse` directory.

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

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

- 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"

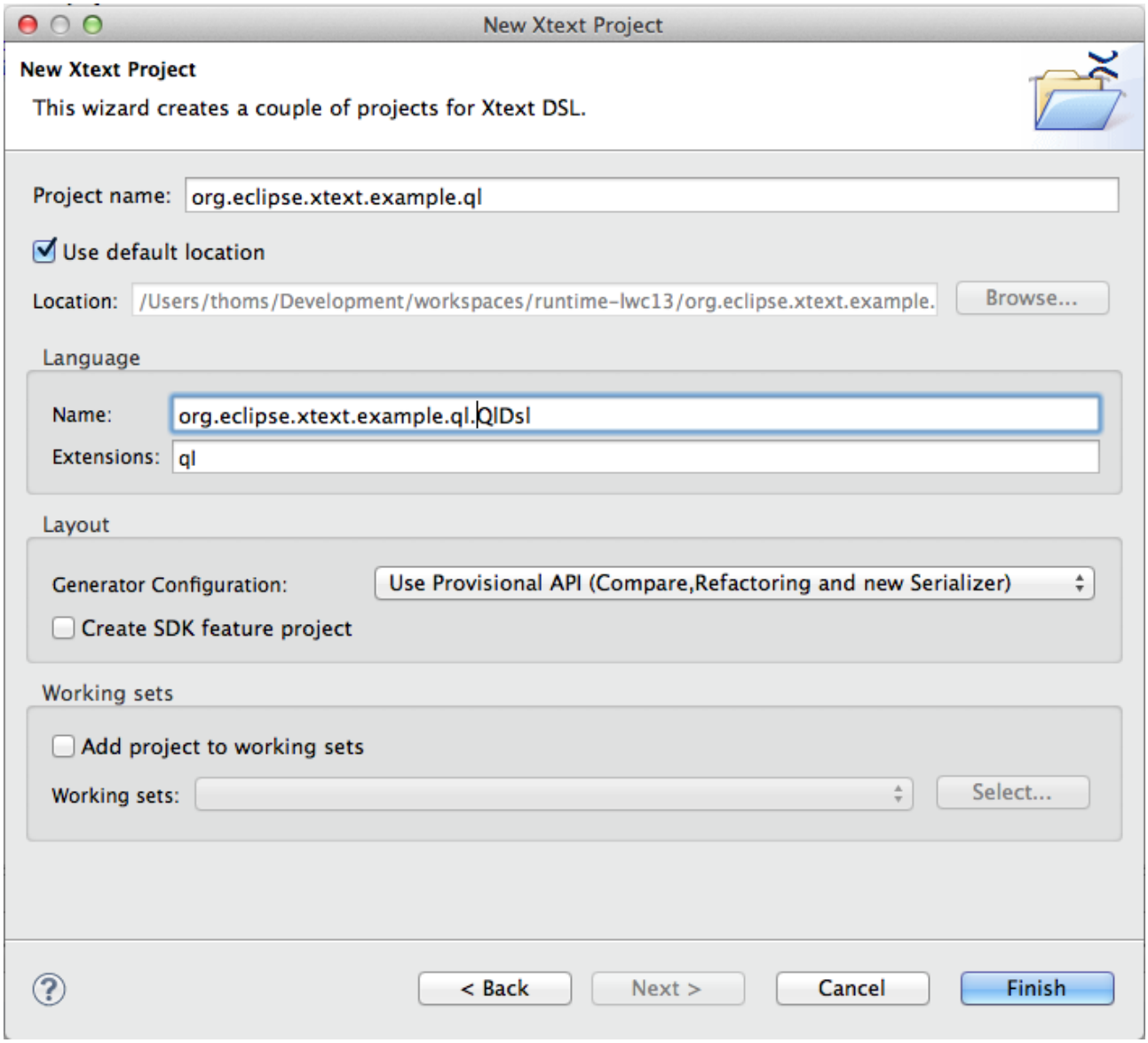
This will allow you re-running the previous launched application by just pressing the Run or Debug button in the Eclipse toolbar, or using keyboard shortcuts. The default settings does not always do what you want.

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:

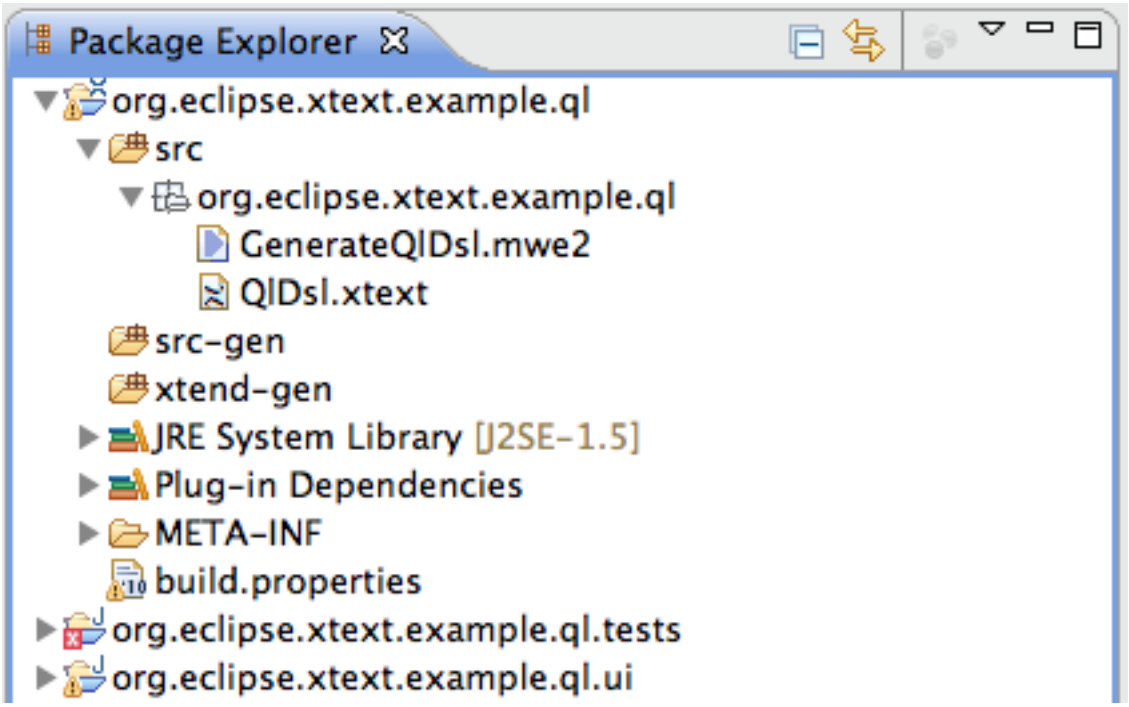
1. Project name: `org.eclipse.xtext.example.ql`. Xtext will create multiple projects, which share this prefix. It is a convention to use a lowercase, dot-separated name.

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 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 dependend. 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[9]:

```
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 */
Questionnare:
  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
;
```

Now let us explain the grammar in more detail:

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

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

The grammar has a unique identifier named `org.eclipse.xtext.example.ql.QlDsl`[10]. 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.

```
1 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[11]. 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 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 more than one rule 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
  - 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 [12].

```
1 Questionnare:
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 in this instance.

[10] That's what has been entered in the project wizard
[11] http://www.eclipse.org/Xtext/documentation.html#metamodelInference
[12] http://www.eclipse.org/Xtext/documentation.htmlgrammarMixins

Each QL model will contain zero to many Import statements:

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

We will use them to import types used as a Question's 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.[13] After `import` statements the QL model can contain multiple `Form` declarations.

```
Import:
  '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.

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

After the imports section QL forms are defined:

```
Form:
  "form" name=ID "{"
    element += FormElement*
  "}";
```

Forms have an attribute called `name`. ID is a `terminal rule`, which is defined in Xtext's root grammar `Terminals`.

```
FormElement:
  Question
;

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

[13] To enforce at least one rule call, the `"+"` operator would be used instead.

3 Testing the Questionnaire Language

- Create Launch Config - Start Runtime - Create Test Project

4 Xbase

- Describe Xbase - Examples for Expressions - Hint: Xbase Tutorial Projects

Extending Xbase: http://koehnlein.blogspot.de/2011/07/extending-xbase.html

5 Including Expressions into the QL Language

- Derive Grammar from Xbase - Generate Implementation

6 JVM Model Inferrence

- Describe the role of JVM Types - Describe the role of the JVM Model Inferrer - Implement JVM Model Inferrer