



Language Workbench Challenge 2013
Xtext Submission

Version: 1.0 - DRAFT 5. April 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 conference¹. The aim is to set a common task² for Language Workbenches³ which is implemented with the different existing alternatives in a comparable way.

This document describes in detail how the task is solved with Xtext⁴. Xtext is one of the most well known Language Workbenches and part of the Eclipse Modeling Project⁵.

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/>

²see <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

Version 1.0 - 2013-04-08

- Final version for the Language Workbench Challenge workshop in Cambridge

Table Of Contents

1	Introduction	6
1.1	Task Description	6
1.2	Technology Stack	6
1.3	Xtext Overview	7
1.4	Installing Eclipse and Xtext	9
1.5	Workspace Setup	10
	Workspace Encoding	10
	Launch Operation	10
1.6	The Survey Application	10
2	Developing the Questionnaire Language	10
2.1	Create the DSL Projects	10
2.2	Defining the Grammar	13
2.3	Generate Language Implementation	16
	Runtime Project - folder <code>src</code>	17
	Runtime Project - folder <code>src-gen</code>	17
2.4	Testing the Questionnaire Language	18
	2.4.1 Creating a Launch Configuration	18
	2.4.2 Create Test Project	19
2.5	Xbase	22
2.6	Including Expressions into the QL Language	23
2.7	JVM Model Inference	27
2.8	Scoping	33
3	Developing the Code Generator	34
3.1	Reference Implementation	34
	3.1.1 IDE Configuration	35
	Webtools	35
	Tomcat installation	35
	3.1.2 Import and run reference	35
	3.1.3 Main Layout	36
	Default CSS	38
	3.1.4 Reference Forms	38
	Bean	38
	Form	38

3.2	Xtend	38
3.3	Code Generator	42
3.3.1	Dispatcher template	42
3.3.2	Output Configuration Provider	43
3.3.3	JSF Generator	45
	JSF Form index	47
	JSF Form	49
	QlDslExtensions	49
3.4	Testing the Questionnaire Application	49
4	Layout and Styling Language (QLS)	49
4.1	The Language QLS	49
4.2	QLS Code Generator	54
5	Additional Concepts	58
5.1	Validation	58
5.1.1	Extending the Java Validator class	59
5.1.2	Constraint: Ensure order of questions	59
5.1.3	Constraint: Type conformance check	61
5.1.4	Testing validation rules	61
5.2	Build	65
5.2.1	settings.xml	65
5.2.2	Parent POM	67
5.2.3	Reactor POM	69
5.2.4	QL Runtime Project POM	70
5.2.5	QL UI Project POM	74
5.2.6	SDK Feature POM	74
5.2.7	p2 Repository	75
5.2.8	Continuous Integration	76
6	Closing Words	77

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.

We assume that you have read the LWC13 assignment document carefully before continuing reading this document.

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 2013⁶. 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 2.1.6⁸ is able to run within the well known Servlet container Apache Tomcat(v7.0).⁹

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

⁶http://wiki.eclipse.org/Kepler/Simultaneous_Release_Plan

⁷<http://www.java-serverfaces.org/>

⁸<http://java-serverfaces.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 Xtext¹¹ 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.



```
2+ * Copyright (c) 2009 itemis AG (http://www.itemis.eu) and others.
8 grammar org.eclipse.xtext.example.domainmodel.Domainmodel with org.eclipse.xtext.xbase.Xbase
9
10 generate domainmodel "http://www.xtext.org/example/Domainmodel"
11
12 DomainModel:
13   » elements+=AbstractElement*;
14
15 AbstractElement:
16   » PackageDeclaration | Entity | Import;
17
18 Import:
19   » 'import' importedNamespace=QualifiedNameWithWildcard;
20
21 PackageDeclaration:
22   » 'package' name=QualifiedName '{'
23   »   » elements+=AbstractElement*
24   » '}' ;
25
26 Entity:
27   » 'entity' name=ValidID ('extends' superType=JvmParameterizedTypeReference)? '{'
28   »   » features+=Feature*
29   » '}' ;
30
31 Feature:
32   » Property | Operation;
33
34 Property:
35   » name=ValidID ':' type=JvmTypeReference;
36
37 Operation:
38   » 'op' name=ValidID '(' (params+=FullJvmFormalParameter (',' params+=FullJvmFormalParameter)*)? ')'
39   »   » body=XBlockExpression;
40
41 QualifiedNameWithWildcard:
42   » QualifiedName ('.' '*')?;
```

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:

```
1 entity Person {
2   name : String
3   gender : m
4   birthday : Date
5 }
6
7 entity Book {
8   title: String
9   authors: Person[]
10  isbnNumber: String
11 }
```

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 adheres 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 xvf -
```

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.

¹²On 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”

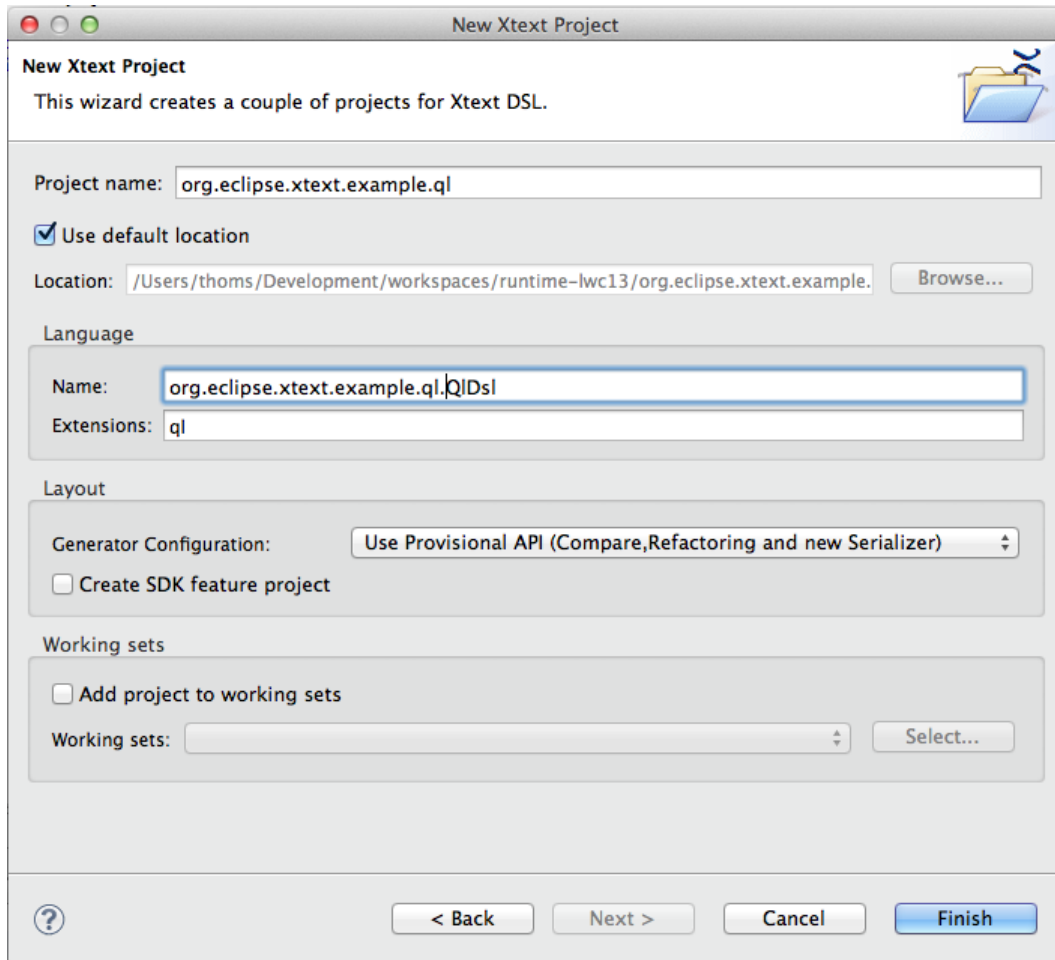
This will allow you re-running the previous launched application by just pressing the Run or Debug button in the Eclipse toolbar, or using keyboard shortcuts. The default settings does not always do what you want.

1.6 The Survey Application

2 Developing the Questionnaire Language

2.1 Create the DSL Projects

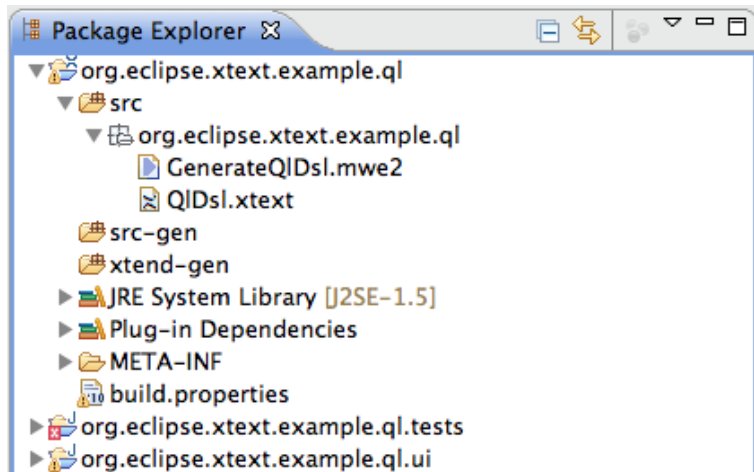
Let's start creating the projects for the Questionnaire DSL. Open the New Project Wizard with *File / New / Project*. Choose “*Xtext Project*” and press “*Next*”.



On the project wizard page enter:

1. Project name: `org.eclipse.xtext.example.q1`. 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.q1.Q1Dsl`. This is an identifier for the language, which must be unique and follows a Java full qualified identifier name pattern.
3. Language Extensions: `q1`. 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.q1`: 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.q1.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.q1.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 generation¹³. 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.

¹³More 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 file¹⁴:

```
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
23 ;
24
25 /**
26  * - Each question identified by a name that at the same time represents the result of the
27  *   question.
28  * - A question has a label that contains the actual question text presented to the user.
29  * - Every question has a type.
30  */
31 Question:
32     name=ID ":" label=STRING type=JvmTypeReference
33 ;
```

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>

```
1 import types.Money
2
3 form Box1HouseOwning {
4     hasSoldHouse: "Did you sell a house in 2010?" boolean
5     hasBoughtHouse: "Did you buy 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 }
```

Now let us explain the grammar in more detail:

```
1 grammar org.eclipse.xtext.example.q1.Q1Dsl with org.eclipse.xtext.xbase.Xbase
```

The grammar has a unique identifier named `org.eclipse.xtext.example.q1.Q1Dsl`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.

```
1 generate q1Dsl "http://www.eclipse.org/xtext/example/q1/Q1Dsl"
```

This is an instruction for the metamodel used for the language. The **generate** statement means that Xtext generates an Ecore metamodel for this grammar¹⁶. 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 (`q1Dsl`), the package's `nsURI` is the second argument ("`http://www.eclipse.org/xtext/example/q1/Q1Dsl`").
- 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](#)

¹⁵That'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>

¹⁸To 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 Pattern¹⁹: 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. Guice²⁰ is a famous Dependency Injection²¹ 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 `QlDslJvmModelInferer.xtext` is a class implemented with the Xtend language. The JVM Model Inferer will play an important role later when we introduce expressions and code generation.
- Class `QlDslJavaValidation` allows the implementation of validation rules for the DSL.

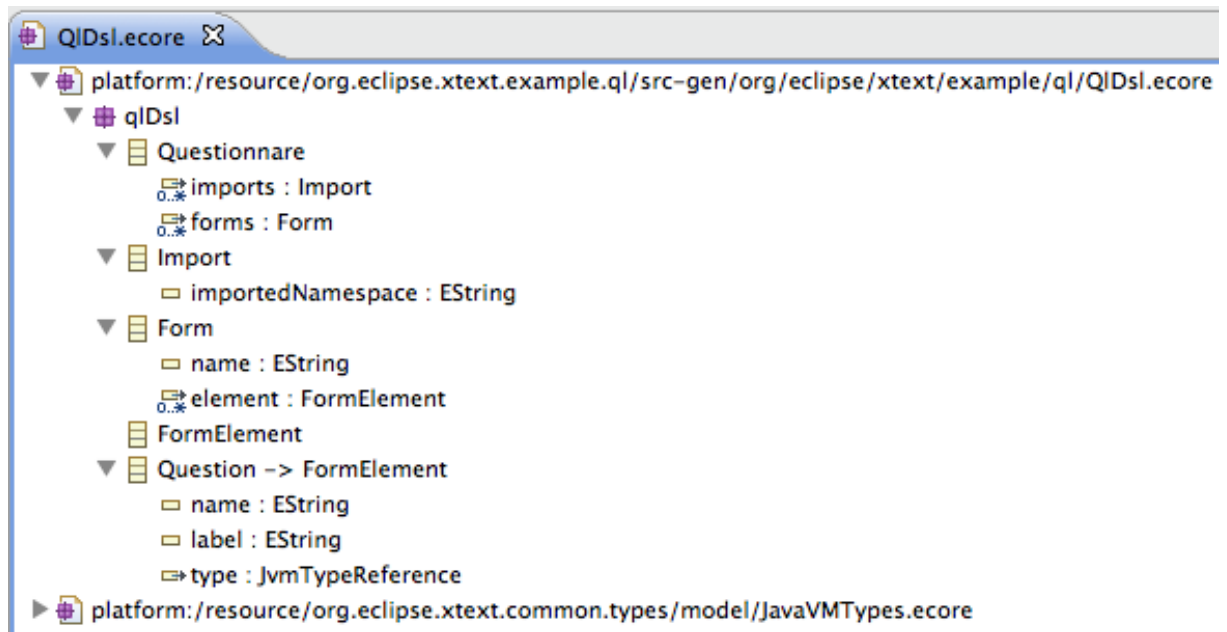
Runtime Project - folder `src-gen`

¹⁹<http://heikobehrens.net/2009/04/23/generation-gap-pattern/>

²⁰<http://code.google.com/p/google-guice/>

²¹http://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.parserantlr.internal` contains an ANTLR3²² grammar and the Lexer and Parser classes generated from it.

2.4 Testing the Questionnaire Language

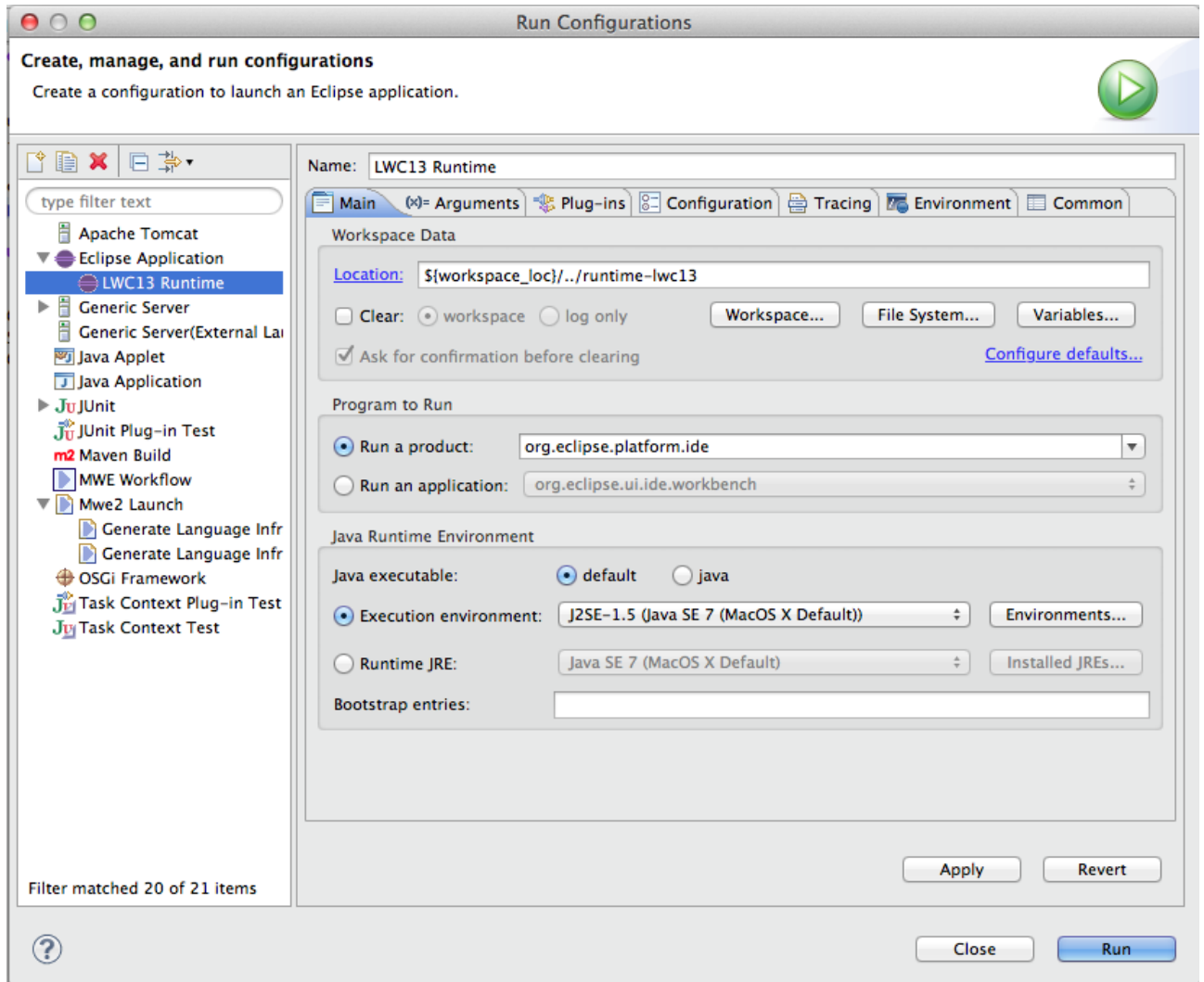
2.4.1 Creating a Launch Configuration

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.

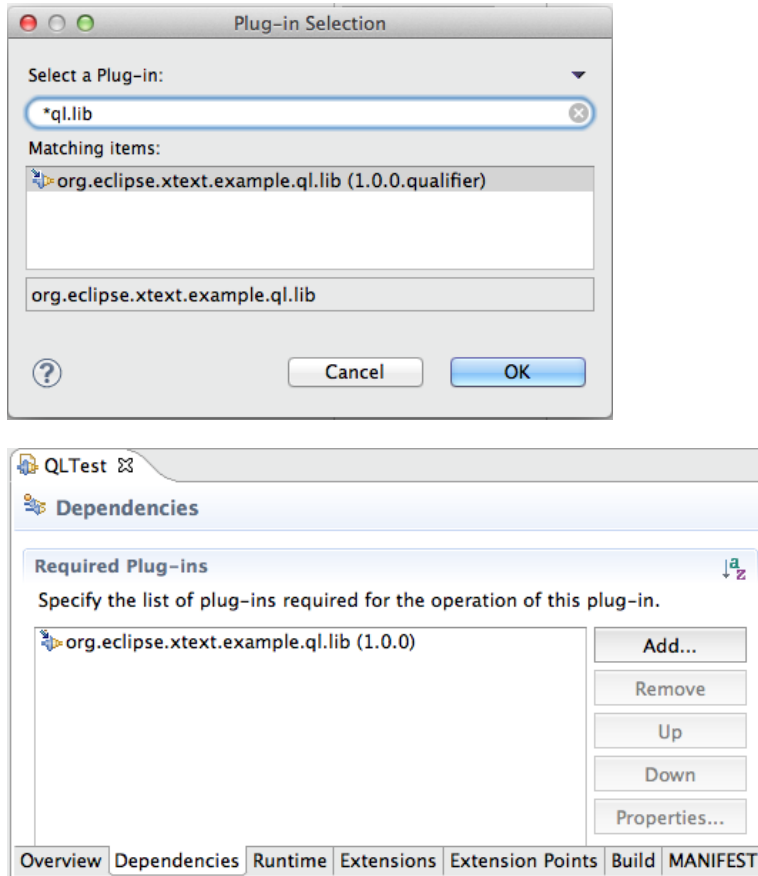
Now press the “Run” button. Another Eclipse instance will start with an empty workspace. Close the Welcome window.

2.4.2 Create Test Project

In the Runtime Workspace create a new Plug-in Project with name “QLTest”.²³

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

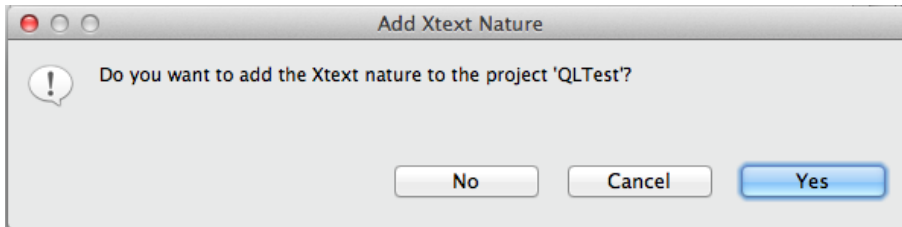
First, we will create a dependency to the library plugin. Open the META-INF/MANIFEST.MF file and switch to page Dependencies. In the *Required Plug-ins* section, add a dependency to `org.eclipse.xtext.example.q1.lib`.



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.q1"`. 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 (`.q1` 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.q1"`²⁴:

```
1 import types.Money
2
3 form Box1HouseOwning {
4     hasSoldHouse: "Did you sell a house in 2010?" boolean
5     hasBoughtHouse: "Did you buy 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>

²⁵if 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:

```
1 "a day has ".toFirstUpper() + 86_400_000 + " milliseconds."
2 // 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:

```
1 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 documentation²⁶ 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:

```
1 stressLevelResult: "Your Stress-Level: " String (
2 {
3     var Integer stressPoints = if (hasTimePressureAtWork) 30 else 0
4     stressPoints = stressPoints + daysSleepingBadPerWeek * 3
5     stressPoints = stressPoints + glassesOfAlcoholPerDay * 12
6     stressPoints = stressPoints - daysWithSportPerWeek * 2
7     if (stressPoints>80) "High" else if (stressPoints>40) "Medium" else "Low"
8 }
9 )
```

2.6 Including Expressions into the QL Language

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

```
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
```

²⁶http://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
27     question.
28  * - A question has a label that contains the actual question text presented to the user.
29  * - 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 following 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 necessity for a condition. The inner elements 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 `GenerateQ1Ds1.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 }
```

²⁸Select 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 proceed 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 inferer class³⁰ :

```

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.AbstractModelInferer
13 import org.eclipse.xtext.xbase.jvmmodel.IJvmDeclaredTypeAcceptor
14 import org.eclipse.xtext.xbase.jvmmodel.JvmTypesBuilder
15
16 class QlDslJvmModelInferer extends AbstractModelInferer {
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)
28                     ,[final = true ^static = true
29                     setInitializer([it.append("1L")])
30                 ])
31
32                 val allQuestions = form.eAllContents.filter(typeof(Question)).toList
33
34                 for (question: allQuestions) {
35                     members += question.toField(question.name, question.type)
36                 }
37             }
38         }
39     }

```

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 defJvmOperation createIsEnabledMethod (Question question) {
61     question.toMethod("is"+question.name.toFirstUpper+"Enabled", typeReferences.
        getTypeForName("boolean", question, null)) [
62         body = [it.append(''return <question.expression == null>;'')]
63     ]
64 }
65
66 /** Create a method <code>public boolean isGroup[groupIndex]Visible ()</code> */
67 defJvmOperation 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 QlDslJvmModelInferer extends AbstractModelInferer {  
2   @Inject extension JvmTypesBuilder  
3   @Inject TypeReferences typeReferences  
4   def dispatch void infer(Questionnaire element, IJvmDeclaredTypeAcceptor acceptor, boolean  
       isPreIndexingPhase) {  
5     ...  
6   }  
7 }
```

The inferer class implements `IJvmModelInferer`, but for convenience we derive from its abstract implementation `AbstractModelInferer`. 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 defined in `IJvmModelInferer`.

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 inferer. 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 <<
10      question.expression == null>>')]
11  ]
12 }
```

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  }
```



```
6 }
7
8 def JvmOperation createIsGroupVisibleMethod (ConditionalQuestionGroup group, int groupIndex)
9 {
10 group.toMethod("isGroup"+groupIndex+"Visible", typeReferences.getTypeForName("boolean",
11 group, null)) [
12 if(group.condition != null) {
13 body = group.condition
14 } else {
15 body = [it.append(''return true;'')]
16 }
17 ]
18 }
```

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 referable 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 manual³¹, several articles³² and implementation examples³³.

The Xtext framework already provides default implementations to solve scoping and linking. In the case of Xbase based languages the `XbaseBatchScopeProvider` is configured by default as implementation of the `IScopeProvider` interface.

For our use case, the default behavior is already sufficient. Through the JVM model inference the implicit `"this"` variable is already bound to the class representing the form, and thus the fields representing the question elements are known as callable features.

³¹Xtext manual:Scoping

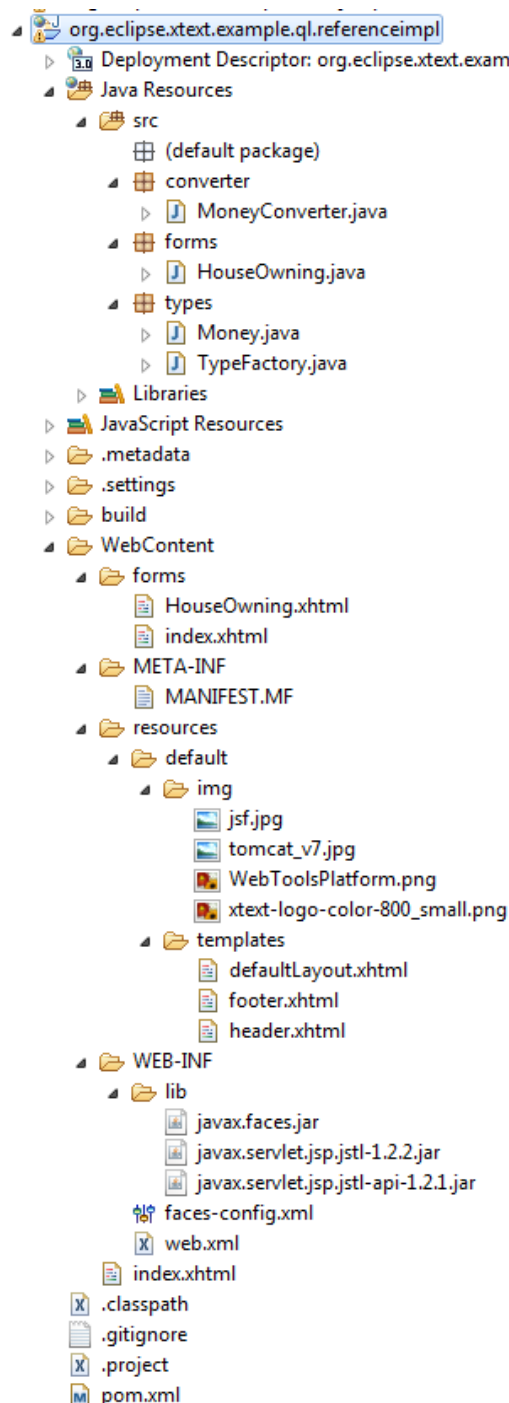
³²e.g. <http://blogs.itemis.de/stundzig/archives/776>

³³e.g. <https://github.com/LorenzoBettini/xtext-scoping>

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, see 1.2) based application, which can be deployed on any Java Web container (also known as a Servlet container) like Glassfish, JBoss and Apache Tomcat.

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

After describing some necessary configuration

of the IDE we will focus on the parts which are dependent on the QL model and thus subject of code generation in sub section 3.1.4. 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/*`)

3.1.1 IDE Configuration

Webtools install new software

<http://download.eclipse.org/releases/juno/>

Web, XML, Java EE and OSGi Enterprise Development

- Eclipse Java EE Developer Tools 3.4.0.v201107072300
- JavaServer Faces Tools (JSF) Project 3.4.1.v201208241503
- JST Server Adapters 3.2.200.v20120517_1442
- JST Server UI 3.4.0.v20120503_1042
- JST Server Adapters Extensions 3.3.101.v20120821_1416
- Eclipse Web Developer Tools 3.4.1.v201208170345
- Eclipse Java Web Developer Tools 3.4.1.v201208231800

Tomcat installation add new server: - tomcat v7

3.1.2 Import and run reference

Checkout from git OR download from project home page @ code.google.com

import project from git: checkout instructions: <https://code.google.com/a/eclipselabs.org/p/lwc13-xtext/source/checkout>

download zip from: project home: <http://code.google.com/a/eclipselabs.org/p/lwc13-xtext/>

run as MWE2 Workflow

1. `/org.eclipse.xtext.example.ql/src/org/eclipse/xtext/example/ql/GenerateQlDsl.mwe2`

2. /org.eclipse.xtext.example.qls/src/org/eclipse/xtext/example/qls/GenerateQlsDsl.mwe2

Run configuration

- LWC13 Runtime

import existing project -

lwc13-xtext

examples

QLTest

3.1.3 Main Layout

The logical entry to the web application is the welcome file `WebContent/index.xhtml` declared in `WebContent/WEB-INF/web.xml`. The `index.xhtml` composes the main layout with page contents and will later be helpful to integrate the generated artifacts with the web application's layout by using JSF's XHTML templating ³⁴.

```
1 <?xml version='1.0' encoding='UTF-8' ?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml"
5     xmlns:h="http://java.sun.com/jsf/html"
6     xmlns:ui="http://java.sun.com/jsf/facelets">
7 <body>
8
9     <ui:composition
10         template="/resources/default/templates/defaultLayout.xhtml">
11         <ui:define name="content">
12             Hello World!
13         </ui:define>
14     </ui:composition>
15
16 </body>
17 </html>
```

Subpages of the application should use the `index.xhtml` placed in `WebContent/` itself as their template and overwrite the content section with custom output via the same pattern.

```
1 <ui:composition template="/index.xhtml">
2     <ui:define name="content">
3         ...my xhtml content
```

³⁴<http://docs.oracle.com/javaee/6/javaxserverfaces/2.1/docs/vldocs/facelets/>

```
4     </ui:define>
5 </ui:composition>
```

Everything between the opening and closing `facelet:define` tag within the subpage will be passed into a corresponding `facelet:insert` section (name attribute is set to "content") defined in the template (here: `defaultLayout.xhtml`) or one of its parent templates when the HTML output is rendered by the JSF framework.

Default template

We keep the main layout definition and the page contents separated from each other. The `WebContent/index.xhtml` defines the main composition of the application's structural layout and content of pages as described in section 3.1.3. Layout template definitions should be placed in a folder `WebContent/resources/*templateName*` in our web application.

To change the main layout it is just necessary to change the `template` reference of the `facelets:composite` in `WebContent/index.xhtml`.

```
1 ...
2 <ui:composition
3     template="/resources/default/templates/defaultLayout.xhtml">
4 ...
```

The reference application is shipped with a default template placed in `WebContent/resources/default/`. It is a very simple one providing only a skeleton `/templates/defaultLayout.xhtml` with basically 3 sections (header, content, footer) where clients can add custom content.

The current web application expects a defined `facelets:insert` section with name 'content' within the template or one of its parents for proper composition. In our reference implementation it is declared in `/resources/default/templates/defaultLayout.xhtml`.

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml"
4     xmlns:h="http://java.sun.com/jsf/html"
5     xmlns:ui="http://java.sun.com/jsf/facelets">
6 <h:head>
7     <title><ui:insert name="title">LWC 2013 Xtext</ui:insert></title>
8 </h:head>
```

```

9 <body>
10   <div id="header">
11     <ui:insert name="header">
12       <ui:include src="/resources/default/templates/header.xhtml" />
13     </ui:insert>
14   </div>
15   <div id="content">
16     <ui:insert name="content">
17       Content area. Compose by use of tag facelet:define & name="content".
18     </ui:insert>
19   </div>
20   <div id="footer">
21     <ui:insert name="footer">
22       <ui:include src="/resources/default/templates/footer.xhtml" />
23     </ui:insert>
24   </div>
25 </body>
26 </html>

```

We added two `facelets:insert` sections to give the possibility to replace the header and footer. To simplify the concrete JSF compositions later we let JSF include default content for both by use of `facelets:include` for a fixed source `src`.

Default CSS

... TODO nach Erhalt der Quellen

3.1.4 Reference Forms

general (eg dependency `baseForm`, `form`)

Bean `yxcvyxcvyxcvyxcvyxcv`

Form `yxcvyxcvyxcvyxcvxcv`

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 documentation³⁵.

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 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. Consequently, 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 them (maybe you already guess where Xtend got its name from). You can easily write your own extensions, e.g. for the Integer type:

```
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

³⁵<http://www.eclipse.org/xtend/documentation.html>

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(SubTypeA input) {
2      // do something SubTypeA specific
3  }
4
5  def dispatch doSomething(SubTypeB input) {
6      // do something SubTypeB specific
7  }
8
9  def useDispatching() {
10     val SuperType a = new SubTypeA()
11     val SuperType b = new SubTypeB()
12     a.doSomething + b.doSomething
13 }
```

In this example the types `SubTypeA` and `SubTypeA` are sub types of `SuperType`. The method `doSomething` is declared for each of the two sub types as input parameter with accordingly different body (here only indicated by comments). In line 12 these methods are called on the objects `a` and `b` which have `SuperType` as static type and one of the sub types as compile time type. Xtend invokes the correct method here, i.e. for `a` the method in lines 1-3 and for `b` the one in lines 5-7 is called.

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 themselves return a String can be inserted (surrounded by guillemots `«»`). There are also logical structures like `for` loops or `if-else` statements supported:

```
1  def htmlContent(List<String> contents) '''
2      <html>
3      <body>
4          «FOR content: contents BEFORE '<p>' SEPARATOR '<br/>' AFTER '</p>'»
5          «IF content.length > 10»
6              Large Content: «content.toFirstUpper»
7          «ELSE»
8              Small Content: «content.toFirstUpper»
9          «ENDIF»
10         «ENDFOR»
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 iterations can be specified. Calling the example method with `['hello', 'Some more text']` results in:

```
1 <html>
2 <body>
3 <p>
4     Small Content: Hello<br/>
5     Large Content: Some more text
6 </p>
7 </body>
8 </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 based on the type of the switch argument is possible, like in the following example:

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

```
6     }  
7 }
```

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.q1` 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:

```
1 package org.eclipse.xtext.generator;  
2  
3 public interface IGenerator {  
4     /**  
5      * @param input - the input for which to generate resources  
6      * @param fsa - file system access to be used to generate files  
7      */  
8     public void doGenerate(Resource input, IFileSystemAccess fsa);  
9 }
```

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.q1.generator`:

```
1 package org.eclipse.xtext.example.q1.generator;  
2  
3 import javax.inject.Inject;  
4  
5 import org.eclipse.emf.ecore.resource.Resource;
```

```
6 import org.eclipse.xtext.generator.IFileSystemAccess;
7 import org.eclipse.xtext.generator.IGenerator;
8 import org.eclipse.xtext.xbase.compiler.JvmModelGenerator;
9
10 @SuppressWarnings("restriction")
11 public class Root implements IGenerator {
12     @Inject
13     JvmModelGenerator jvmModelGenerator;
14
15     public void doGenerate(Resource input, IFileSystemAccess fsa) {
16         // dispatch to other generators
17         jvmModelGenerator.doGenerate(input, fsa);
18     }
19 }
```

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

Next, Xtext has to know that `Root` is the template that has to be invoked as generator implementation. 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 custom `IGenerator` implementation, this has to be added to the `QlDslRuntimeModule` class. Open this class and add a configuration that binds the `IGenerator` interface to the `Root` class.

```
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 Output Configuration Provider

In JSF applications all the web related content is normally placed under `./WebContent` instead of `./src-gen`, which is mostly used as output for generated java artifacts. We want to adapt to the web applications structure and separate generated java classes and JSF artifacts from each other. For that purposes add a class called `JsfOutputConfigurationProvider.java` derived from `org.eclipse.xtext.generator.OutputConfigurationProvider`³⁶ within package

³⁶<http://xtextcasts.org/episodes/15-output-configurations>

org.eclipse.xtext.example.q1.generator. The new provider adds an additional `OutputConfiguration` for the output directory `./WebContent` as shown in the listing below.

```
1 package org.eclipse.xtext.example.q1.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
21             .setDescription("Read-only Output Folder for web generated application artifacts");
22         webContent.setOutputDirectory("./WebContent");
23         webContent.setOverrideExistingResources(true);
24         webContent.setCreateOutputDirectory(true);
25         webContent.setCleanUpDerivedResources(true);
26         webContent.setSetDerivedProperty(true);
27         outputConfigurations.add(webContent);
28
29         return outputConfigurations;
30     }
31 }
```

The interface `OutputConfiguration` provides several options to configure the behavior of the so called `Outlet`. We use the defaults as in `OutputConfigurationProvider` except its name, description and `outputDirectory`.

Our `JSFOutputConfigurationProvider` can be bound in the `Q1DslRuntimeModule` by overriding/extending the method `configure`.

```
1 @Override public void configure(Binder binder) {
2     super.configure(binder);
3     binder.bind(IOutputConfigurationProvider.class)
```

```

4      .to(JsfOutputConfigurationProvider.class).in(Singleton.class);
5  }

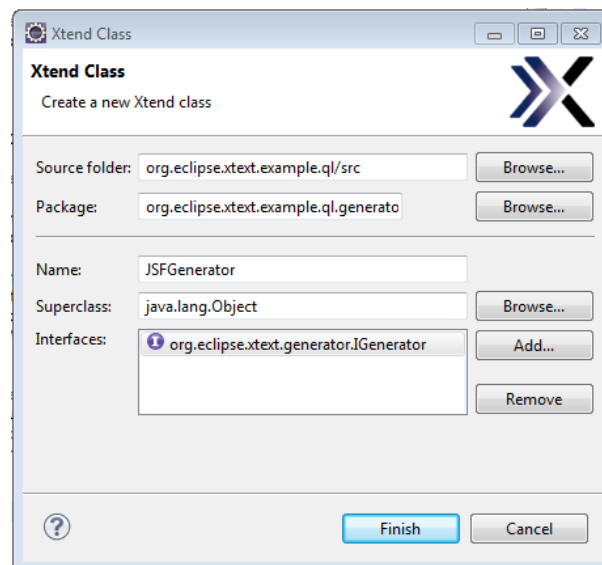
```

After this step we can refer to the additional `OutputConfiguration` in generators by use of the constant `WEB_CONTENT` defined in class `JsfOutputConfigurationProvider`.

3.3.3 JSF Generator

After creation of the class `Root` in section 3.3.1 where we easily can add new Generators and the definition of the `JsfOutputConfigurationProvider` in section 3.3.2 which provides an output folder for JSF artifacts, 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 the figure above, we can finish the wizard so that the class shown in the following listing 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 }
```

To get the created JSFGenerator executed we have to inject and dispatch to it in our dispatcher template Root.java which was created in section 3.3.1 earlier.

```
1 package org.eclipse.xtext.example.ql.generator;
2
3 import javax.inject.Inject;
4
5 import org.eclipse.emf.ecore.resource.Resource;
6 import org.eclipse.xtext.generator.IFileSystemAccess;
7 import org.eclipse.xtext.generator.IGenerator;
8 import org.eclipse.xtext.xbase.compiler.JvmModelGenerator;
9
10 public class Root implements IGenerator {
11     @Inject
12     JvmModelGenerator jvmModelGenerator;
13     @Inject
14     JSFGenerator jsfGenerator;
15
16     public void doGenerate(Resource input, IFileSystemAccess fsa) {
17         // dispatch to other generators
18         jvmModelGenerator.doGenerate(input, fsa);
19         jsfGenerator.doGenerate(input, fsa);
20     }
21 }
```

Now it is time to add some functionality to the JSFGenerator. Open the file JSFGenerator.xtend and go to the doGenerate extension which is responsible to generate artifacts. Delete the auto-generated body of the extension - initially it just throws an UnsupportedOperationException - and add the following lines as first statements to prevent execution of the generator if the file extension does not fit.

```
1 if (input.URI.fileExtension!="ql")
2     return
```

After this pre condition is passed we want to execute the generator logic for our model so it is

a good idea to save the models root node in a variable.

```
1 val questionnaire = input.contents.head as Questionnaire
```

Because we want to generate JSF artifacts into the `WebContent` folder in the following steps we let Guice add a `JSFOutputConfigurationProvider` extension to our `JSFGenerator`.

```
1 class JSFGenerator implements IGenerator{
2     @Inject extension JSFOutputConfigurationProvider
3     ...
4 }
```

After this we have the possibility to use the `WEB_CONTENT` outlet constant as described in section 3.3.2.

The following paragraphs will describe the different extensions of the `JSFGenerator` which are responsible to generate the JSF artifacts described in 3.1. In a real world project it can be a good decision to separate different artifacts in different `Xtend` files. Our sample is a very simple one, so we will add a new extension definition derived from the sample below to the `JSFGenerator.xtend` class which encapsulates the logic to generate a single artifact.

```
1 def generate_Artifact (EObject modelInfo)
2     '''<?xml version='1.0' encoding='UTF-8' ?>
3         <!-- @generated -->
4         <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
5             xhtml1/DTD/xhtml1-transitional.dtd">
6         <html xmlns="http://www.w3.org/1999/xhtml"
7             xmlns:h="http://java.sun.com/jsf/html"
8             xmlns:ui="http://java.sun.com/jsf/facelets">
9             ...
10            </html>
11        '''
```

To get a valid `xhtml` page we have to generate the `DOCTYPE` and `HTML` tag into each `xhtml` file what was replaced in the following listings to focus on what matters.

JSF Form index

To get simple access to all generated forms in the application we want to generate an index page where a link is included for each form which is defined in our model. The generated index page should be saved in a file called `index.xhtml` within a subfolder `'generated/forms/'` of the `WEB_CONTENT` outlet created in section 3.3.2.

```
1 class JSFGenerator implements IGenerator{
```

```

2  @Inject extension JSFOutputConfigurationProvider
3  @Inject extension QlDslExtensions
4
5  override doGenerate(Resource input, IFileSystemAccess fsa) {
6      if (input.URI.fileExtension!="ql")
7          return
8      // model root
9      val questionnaire = input.contents.head as Questionnaire
10     // generate index page with links to generated forms
11     val contentIndex = generate_FormIndex(questionnaire.forms)
12     val fileNameIndex = "generated/forms/index.xhtml"
13     fsa.generateFile(fileNameIndex,WEB_CONTENT, contentIndex)
14     ...
15 }
16 ...
17 }

```

For the new artifact add a new extension called `def generate_FormIndex`. It receives a list of `Form` elements. In a short loop it generates a `html:outputlink` node for each element in the given list of forms. Because `doGenerate` is called for each QL resource, the generator currently has the limitation that all QL model elements have to be defined within the same QL resource to ensure generation of a correct form index page.

```

1  def generate_FormIndex (List<Form> forms)
2  '''...
3      <ui:composition template="/index.xhtml">
4          <ui:define name="content">
5
6              <<FOR elem: forms SEPARATOR "<br/>">
7                  <h:outputLink value="«elem.name».jsf">«elem.name»</h:outputLink>
8              <<ENDFOR>
9
10             </ui:define>
11         </ui:composition>
12         ...
13     '''

```

By using the attribute `template` of a `composition` tag as already described in 3.1.3, the structure and styles of `index.xhtml` will be derived in our form index page. Our generated form index needs a `index.xhtml` in the applications root folder which itself or one of its parent templates defines a `facelet:insert` section with name 'content' like described in section 3.1.3.

To get the possibility to change the template for all generated files in a single file easily later we will use the generated form index as template for generated form pages in a later step. The generated form index looks similar to the one described in section 3.1.4.

JSF Form

TODO

JSF Form base

JSF Form composite

QIDslExtensions

TODO

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

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. How this can be done is covered in section 5.1. In a section (or page) the styling information for the questions of the included form can be defined as in lines 3-9 and 11. 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 occurrences 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:

```
1 grammar org.eclipse.xtext.example.qls.QlsDsl with org.eclipse.xtext.common.Terminals
2
3 generate qlsDsl "http://www.eclipse.org/xtext/example/qls/QlsDsl"
4 import "http://www.eclipse.org/xtext/example/ql/QlDsl" as ql
5
6 QuestionnaireStyleModel:
```

```

7   pages+=Page*;
8
9   Page:
10  "page" name=ID ("uses" form=[ql::Form|ID])? "{"
11      element+=PageElement*
12      navigation=Navigation?
13  "}"
14  ;
15
16  PageElement:
17      QuestionStyling | Section
18  ;
19
20  QuestionStyling:
21      "question" question=[ql::Question] styling+=StyleInformation?
22  ;
23
24  StyleInformation: {StyleInformation}
25  "["(
26      ("font-style:" fontStyle=STRING)? &
27      ("font-weight:" fontWeight=STRING)? &
28      ("font-color:" fontColor=STRING)? &
29      ("font-family:" fontFamily=STRING)? &
30      ("widget:" widget=Widget)?
31  "]"
32  ;
33
34  Widget: {Widget}
35  widgetType=("Radio"|"DropDown"|"CheckBox"|"Text"|"Slider") ("[" labels+=STRING ("," labels
36      +=STRING)* "]" )?
37  ;
38
39  Section:
40      "section" name=ID ("uses" form=[ql::Form|ID])? "{"
41      element+=PageElement*
42      "}"
43  ;
44
45  Navigation: {Navigation}
46  "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:

```
1 StyleInformation: {StyleInformation}
2   "["(
3     ("font-style:" fontStyle=STRING)? &
4     ("font-weight:" fontWeight=STRING)? &
5     ("font-color:" fontColor=STRING)? &
6     ("font-family:" fontFamily=STRING)? &
7     ("widget:" widget=Widget)?
8   )" "
9 ;
```

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:

```
1 Navigation: {Navigation}
2   "navigation" "{" (nextPage+=[Page|ID])+ "}"
3 ;
```

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 languages 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:

```
1 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 (`::`):

```
1 QuestionStyling:
2   "question" question=[ql::Question] styling+=StyleInformation?
3 ;
```

This grammar rule allows for referring 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:

```
1 class QlsDslScopeProvider extends AbstractDeclarativeScopeProvider {
2
3   @Inject extension QlsDslExtensions
4
5   def IScope scope_QuestionStyling_question(EObject context, EReference ref) {
6     Scopes::scopeFor(
7       EcoreUtil2::getAllContentsOfType(context.form, typeof(Question))
8     );
9   }
10 }
```

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. This is for example the case when the content assist needs to compute the elements to be proposed. The method's implementation is quite simple. The static method `scopeFor` expects a list of elements which are in scope (=visible) in the current context. For computing the visible elements, we use a recursive algorithm to get the form declaration of the parent section or page which is mandatory. This algorithm is defined in the Xtend class `QlsDslExtensions` which is imported in line 3. Note, that the expression `context.form` in line 7 will call the corresponding extension method in `QlsDslExtensions`. There, we use Xtend's polymorphic dispatching feature (see also section 3.2):

```
1 class QlsDslExtensions {
2
3   def dispatch Form getForm(Section section) {
4     if (section.form != null) {
5       section.form
6     }
7     else {
8       section.eContainer.form
9     }
10  }
11
12   def dispatch getForm(Page page) {
13     page.form
14   }
15 }
```

Note, that for the first method the return type `Form` needs to be declared explicitly since it is recursively calling itself in line 8 and thus its return type cannot be derived automatically. The calculated form is used in the scope provider as input for the helper method `getAllContentsOfType` which is used to collect all questions defined in the given form.

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

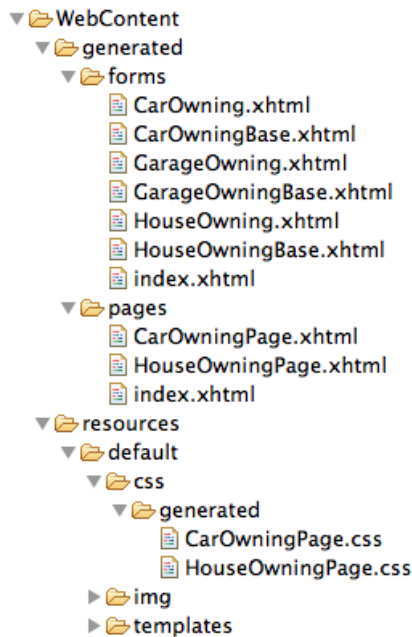
Let's take the following QLS model as a reference example:

```
1 page HouseOwningPage {
2   section house uses HouseOwning {
3     question hasBoughtHouse [font-style:"italic"]
4     question valueResidue [
5       font-weight: "bold"
6       font-color: "#2233FF"
7       font-family: "Verdana"
8     ]
9   }
10  section garage uses GarageOwning {
11    question hasBoughtGarage [widget: Radio["Yepp", "Nope"]]
12  }
13  navigation {
14    CarOwningPage
15  }
```

```

16 }
17
18 page CarOwningPage uses CarOwning {
19     question hasSoldCar [font-color: "green"]
20     question hasBoughtCar [font-color: "red"]
21 }

```



The intended generated file structure is visualized in the screenshot on the left. The files in the folder `forms` are generated by the QL model to code generator as described in chapter 3. The code generator for the QLS model needs to generate the XHTML files under the folder `pages` and the CSS files in `resources/default/css/generated`. The file `pages/index.xhtml` serves as root page containing the link to the first actual page to be presented.

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` together:

```

1 <html xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:ui="http://java.sun.com/jsf/facelets"
3     xmlns:h="http://java.sun.com/jsf/html"
4     xmlns:f="http://java.sun.com/jsf/core">
5 <h:head></h:head>
6 <ui:composition template="/index.xhtml">
7     <ui:define name="content">
8         <h:outputStylesheet library="default/css/generated" name="HouseOwningPage.css" />
9         <div><ui:include src="/generated/forms/HouseOwningBase.xhtml" /></div><p/>
10        <div><ui:include src="/generated/forms/GarageOwningBase.xhtml" /></div>
11        <div>
12            <h:outputLink value="CarOwningPage.jsf">CarOwningPage</h:outputLink>
13        </div>
14    </ui:define>
15 </ui:composition>
16 </html>

```

The two pages are referenced in lines 9 and 10. Line 12 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 our scenario this is always the id of the parent form concatenated with the id of the element 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 clarified, the code generator itself can be written. Here again we use Xtend (see section 3.2):

```
1 class QlsDslGenerator implements IGenerator {
2     @Inject extension JsفOutputConfigurationProvider
3     @Inject extension QlDslExtensions
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     «FOR styleInfo: page.eAllContents.filter(typeof(StyleInformation)).toList»
26     «styleInfo.id» {
27         «IF styleInfo.fontColor != null»color: «styleInfo.fontColor»;«ENDIF»
28         «IF styleInfo.fontFamily != null»font-family: «styleInfo.fontFamily»;«ENDIF»
29         «IF styleInfo.fontStyle != null»font-style: «styleInfo.fontStyle»;«ENDIF»
30         «IF styleInfo.fontWeight != null»font-weight: «styleInfo.fontWeight»;«ENDIF»
31     }
32     «ENDFOR»
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="«page.name».css" />
45             «IF page.form != null»
46             <div><ui:include src="/generated/forms/«page.form.name»Base.xhtml" /></div>
47             «ELSE»
48                 «FOR section: page.eAllContents.toList.filter(typeof(Section)).toList SEPARATOR '<p'
49                     />'»
50                 <div><ui:include src="/generated/forms/«section.form.name»Base.xhtml" /></div>
51                 «ENDFOR»
52             «ENDIF»
53         </div>
54         «IF page.navigation != null»
55             «FOR nextPage: page.navigation.nextPage»
56             <h:outputLink value="«nextPage.name».jsf">«nextPage.name»</h:outputLink>
57             «ENDFOR»
58         «ENDIF»
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="«page.name».jsf">«page.name»</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   "#" + question.form.id + "\\:lbl" + question.id.toFirstUpper
83 }
84 }
```

The `doGenerate` method is similar to the one of the `JSFGenerator` from section 3.3.3. It basically defines file names and delegates to other methods for defining the generated files' contents (`generateCssFile`, `generateXhtmlFile`, `generateIndexPage`). We won't go into details here. Instead let's test the generated application.

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 framework³⁷. The Xtext User Manual contains a Validation chapter that is worth reading additionally³⁸.

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

³⁷<http://www.eclipse.org/modeling/emf/?project=validation>

³⁸see <http://www.eclipse.org/Xtext/documentation.html#validation>

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.example` 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 }
```

Derive `QlDslJavaValidator` from the the Xtend validator:

```
1 public class QlDslJavaValidator extends QlDslXtendValidator {
2     // do nothing here, rules are implemented in Xtend
3 }
```

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:

```
1 if (x) { y: "Y?" boolean }
2 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`.

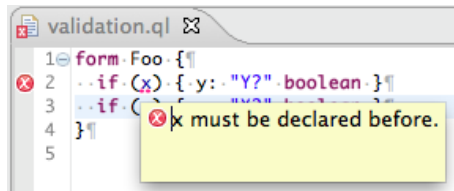
Now open the `QlDslXtendValidator` class again and add this method³⁹:

```
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,
```

³⁹<https://gist.github.com/kthoms/5240455>

```
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:

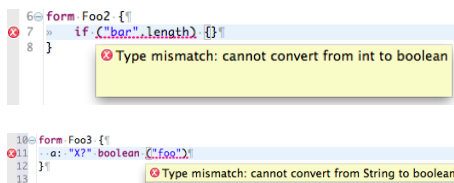


5.1.3 Constraint: Type conformance check

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 unnecessary to declare types anywhere. For dynamic languages 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 Xtend framework already contains the necessary infrastructure. Remember that Xtend 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, "
34             ERR_FEATURE_CALL_BEFORE_DECLARATION", "must be declared before")
35     }
36
37     @Test
38     def void testValidation_CallBeforeDeclaration_expectSuccess () {
39         '''
40         form Foo {
41             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
48 // correct and should ultimately be booleans.
49 // The assigned variables should be assigned consistently: each assignment should use the
50 // same type.
51
52 @Test
53 def void testValidation_ConditionTypeCheck_expectError () {
54   '''
55   form Foo {
56     if ("foo".length) { a: "X?" boolean }
57   }
58   ''' .parse.assertError(XbasePackage::eINSTANCE.XMemberFeatureCall,
59     "org.eclipse.xtext.xbase.validation.IssueCodes.incompatible_types", "Type mismatch")
60 }
61
62 @Test
63 def void testValidation_ConditionTypeCheck_expectSuccess () {
64   '''
65   form Foo {
66     if ("foo".length>1) { a: "X?" boolean }
67   }
68   ''' .parse.assertNoErrors
69 }
70
71 @Test
72 def void testValidation_AssignmentTypeCheck_expectFailure () {
73   '''
74   form Foo {
75     a: "X?" boolean ("foo".length)
76   }
77   ''' .parse.assertError(XbasePackage::eINSTANCE.XMemberFeatureCall,
78     "org.eclipse.xtext.xbase.validation.IssueCodes.incompatible_types", "Type mismatch")
79 }
80
81 @Test
82 def void testValidation_AssignmentTypeCheck_expectSuccess () {
83   '''
84   form Foo {
85     a: "X?" boolean ("foo".length>1)
86   }
87 }
```

```
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`.

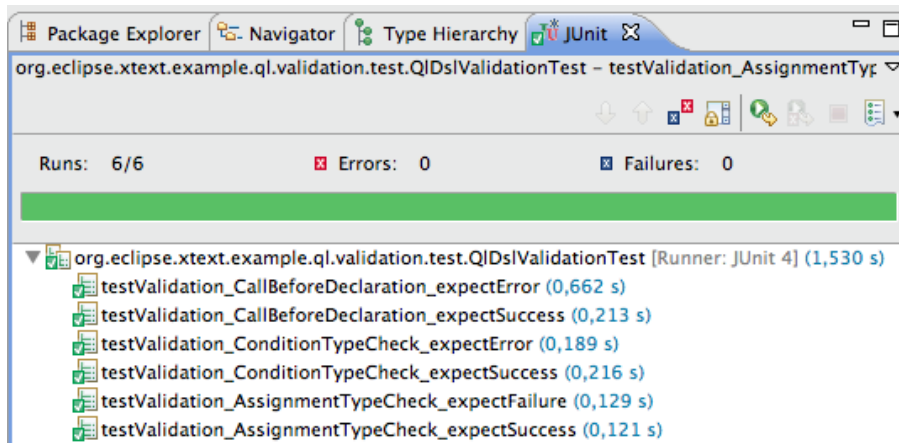
Next, the class adds two extension classes. `ParseHelper` provides a `parse()` method for char sequences. This will parse and validate the model. Afterwards the observed issues can be asserted with the methods from the

```
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*).



5.2 Build

The build of software products is usually done on build servers without manual interaction. In the Eclipse ecosystem several build systems are available. The one gaining most attention nowadays is Maven Tycho, which is a set of plugins for the well-known Maven build framework.

In Maven, the build descriptors are so-called POM files (usually `pom.xml`). For the LWC example we have added such POM files to enable a headless build.

5.2.1 settings.xml

Maven needs to know about repositories from where it can download artifacts. By default, Maven only knows about Maven Central, which is the repository hosted by Apache. For our build we need to consume some artifacts that are not available at Maven Central:

- The Fornax Workflow plugin is used to execute MWE workflows to generate code from the Xtext grammar.
- The Xtend plugin compiles Xtend classes to Java code.

All plugin dependencies must be resolvable through Eclipse p2 repositories. The layout p2 is a special layout contributed by Tycho. The plugins consume dependencies from

- Eclipse Juno: Composite repository of the Eclipse Juno simultaneous release. Most plugins are resolved through this repository.
- Eclipse Orbit: Orbit contains OSGi bundles of 3rd party components (like logj, javax.faces, javax.inject, Google Guava etc.).

- Eclipse Xtext: The release repository for Xtext, Xtend, MWE2.

In `settings.xml`, repository configurations must be contained in a profile. We define a profile `external-repositories`, which is activated by default in the `activeProfiles` section.

```
1 <settings>
2   <activeProfiles>
3     <activeProfile>external-repositories</activeProfile>
4   </activeProfiles>
5   <profiles>
6     <profile>
7       <id>external-repositories</id>
8       <repositories>
9         <repository>
10           <id>Eclipse Juno</id>
11           <layout>p2</layout>
12           <url>http://download.eclipse.org/releases/juno</url>
13         </repository>
14         <repository>
15           <id>Eclipse Orbit</id>
16           <layout>p2</layout>
17           <url>http://download.eclipse.org/tools/orbit/downloads/drops/R20120526062928/
18             repository/
19           </url>
20         </repository>
21         <repository>
22           <id>Eclipse Xtext</id>
23           <layout>p2</layout>
24           <url>http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/
25           </url>
26         </repository>
27       </repositories>
28       <pluginRepositories>
29         <pluginRepository>
30           <id>fornax.releases</id>
31           <name>Fornax Release Repository</name>
32           <url>http://www.fornax-platform.org/nexus/content/repositories/releases/
33           </url>
34         </pluginRepository>
35         <pluginRepository>
36           <id>xtend</id>
37           <url>http://build.eclipse.org/common/xtend/maven/</url>
38         </pluginRepository>
39       </pluginRepositories>
40     </profile>
```

```
40 </profiles>
41 </settings>
```

The settings file is placed in the repository under `/devenv/lwc13.devenv/settings.xml`.

In order to use this settings file, it has to be put to `<HOME>/.m2` or passed with the “-s” option to the `mvn` command.

5.2.2 Parent POM

Since Maven supports a simple POM inheritance, it is common to extract common settings for a set of project modules into a so-called Parent POM. This `pom.xml` file is placed in the root of the repository⁴⁰.

The parent POM is responsible for different things:

- Definition of the common `groupId` and version

```
1 <groupId>org.eclipse.xtext.example.q1</groupId>
2 <version>1.0.0-SNAPSHOT</version>
```

- Tycho Plugin configuration. Note the `<extensions>true</extensions>` entry. To make the Tycho version that is used easier configurable, a property `tycho-version` is defined, which is used for all Tycho plugin configurations.

```
1 <properties>
2   <tycho-version>0.17.0</tycho-version>
3 </properties>
4 <build>
5   <plugins>
6     <plugin>
7       <groupId>org.eclipse.tycho</groupId>
8       <artifactId>tycho-maven-plugin</artifactId>
9       <version>${tycho-version}</version>
10      <extensions>true</extensions>
11    </plugin>
12    ...
```

- Supported Target Environments. Eclipse is partially OS dependent. Which platforms should be considered in the target platform configuration is configured with the `target-platform-configuration` plugin.

⁴⁰<http://code.google.com/a/eclipseorg/p/lwc13-xtext/source/browse/pom.xml>

```
1 <plugin>
2   <groupId>org.eclipse.tycho</groupId>
3   <artifactId>target-platform-configuration</artifactId>
4   <version>${tycho-version}</version>
5   <configuration>
6     <resolver>p2</resolver>
7     <pomDependencies>consider</pomDependencies>
8     <environments>
9       <environment>
10        <os>win32</os>
11        <ws>win32</ws>
12        <arch>x86</arch>
13      </environment>
14      <environment>
15        <os>win32</os>
16        <ws>win32</ws>
17        <arch>x86_64</arch>
18      </environment>
19      <environment>
20        <os>macosx</os>
21        <ws>cocoa</ws>
22        <arch>x86_64</arch>
23      </environment>
24    </environments>
25  </configuration>
26 </plugin>
```

- Java Source Code version. The produced code requires Java 1.6 for compilation. This needs to be configured at the `tycho-compiler-plugin`.

```
1 <plugin>
2   <groupId>org.eclipse.tycho</groupId>
3   <artifactId>tycho-compiler-plugin</artifactId>
4   <version>${tycho-version}</version>
5   <configuration>
6     <encoding>UTF-8</encoding>
7     <meminitial>128m</meminitial>
8     <maxmem>1024m</maxmem>
9     <source>6.0</source>
10    <target>6.0</target>
11    <verbose>true</verbose>
12  </configuration>
13 </plugin>
```

- Maven Plugin Management. In the `pluginManagement` section the plugins that potentially participate in the build are configured with their versions. This is because Maven would select the latest available version of a plugin instead and prints warnings during the build. It is better to fix the versions of plugins that are used to those with which the build has been tested. Also basic plugin configurations can be added here.

```
1 <plugin>
2   <groupId>org.eclipse.tycho</groupId>
3   <artifactId>tycho-p2-repository-plugin</artifactId>
4   <version>${tycho-version}</version>
5 </plugin>
6 <plugin>
7   <groupId>org.apache.maven.plugins</groupId>
8   <artifactId>maven-clean-plugin</artifactId>
9   <version>2.4.1</version>
10 </plugin>
11 ...
```

5.2.3 Reactor POM

The project consists of several modules, which is called a [Multi Module Project](#) in Maven terms. The modules of a project are listed in a `<modules>` section. Often the modules are enlisted in the Parent POM, but we splitted this. There is a POM that only contains module entries in the `projects` folder⁴¹:

```
1 <modules>
2   <module>org.eclipse.xtext.example.q1</module>
3   <module>org.eclipse.xtext.example.q1.ui</module>
4   <module>org.eclipse.xtext.example.q1s</module>
5   <module>org.eclipse.xtext.example.q1s.ui</module>
6   <module>org.eclipse.xtext.example.q1.sdk</module>
7   <module>org.eclipse.xtext.example.q1.repository</module>
8 </modules>
```

When building the DSL projects, this POM has to be executed. Assuming the build is executed from the repository root, the typical build command would be:

```
1 mvn -s devenv/lwc13.devenv/settings.xml -f projects/pom.xml clean install
```

⁴¹<http://code.google.com/a/eclipseorg/p/lwc13-xtext/source/browse/projects/pom.xml>

5.2.4 QL Runtime Project POM

Let's look a bit at the POM of the QL Runtime Project `org.eclipse.xtext.example.ql`.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
   maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.eclipse.xtext.example.ql</groupId>
7     <artifactId>parent</artifactId>
8     <version>1.0.0-SNAPSHOT</version>
9     <relativePath>../../pom.xml</relativePath>
10  </parent>
11  <artifactId>org.eclipse.xtext.example.ql</artifactId>
12  ...
```

At the beginning we have the typical Maven coordinates and parent dependency. The Parent POM is at the root of the repository, which is 2 directories up. The POM one directory up is the Reactor POM. Note that it is important to mention here the exact same version as declared in the Parent POM. This module inherits the parent's version and groupId coordinates, only the artifactId changes.

Maven Tycho enforces that the POM version matches **Bundle-Version**, and the **artifactId** must be equal to the **Bundle-SymbolicName** of the manifest. A bit special are SNAPSHOT versions, which is a special Maven concept for intermediate development versions of artifacts. When the project version is a snapshot version, the bundle version must have an additional qualifier. In our case here, the project version is `1.0.0-SNAPSHOT`, which means the bundle version must be `1.0.0.qualifier`. The qualifier gets replaced at build time by a timestamp.

```
1 Bundle-Name: org.eclipse.xtext.example.ql
2 Bundle-Version: 1.0.0.qualifier
```

The next entry, packaging, is especially important.

```
1 <packaging>eclipse-plugin</packaging>
```

Tycho introduces some Eclipse specific packaging types, which trigger the module to be built by Tycho plugins.

```
1 <build>
2   <resources>
```

```
3      <resource>
4          <directory>${project.build.directory}/xtext</directory>
5      </resource>
6  </resources>
7  <plugins>
8      <!-- Copy all Xtext related sources to separate folder that is registered as resource
9           folder -->
10     <plugin>
11         <artifactId>maven-resources-plugin</artifactId>
12         <executions>
13             <execution>
14                 <id>copy-resources</id>
15                 <phase>initialize</phase>
16                 <goals>
17                     <goal>copy-resources</goal>
18                 </goals>
19                 <configuration>
20                     <outputDirectory>${project.build.directory}/xtext</outputDirectory>
21                     <resources>
22                         <resource>
23                             <directory>src</directory>
24                             <includes>
25                                 <include>**/*.xtext</include>
26                                 <include>**/*.mwe2</include>
27                             </includes>
28                         </resource>
29                     </resources>
30                 </configuration>
31             </execution>
32         </executions>
33     </plugin>
```

Next, we do something special for MWE2. In order to execute an MWE2 workflow this workflow file must be loadable from the project's classpath. But the MWE2 file is contained in the source path of a project. Therefore an additional directory **target/xtext** is added as resource folder to the project (resource folders are added to the classpath), and copy Xtext related resources from the source directory to this directory by use of the **maven-resource-plugin**.

```
1  <plugin>
2      <artifactId>maven-clean-plugin</artifactId>
3      <configuration>
4          <filesets>
5              <fileset>
6                  <directory>src-gen</directory>
```

```

7      <excludes>
8          <exclude>.gitignore</exclude>
9      </excludes>
10 </fileset>
11 <fileset>
12     <directory>xtend-gen</directory>
13     <excludes>
14         <exclude>.gitignore</exclude>
15     </excludes>
16 </fileset>
17 <fileset>
18     <directory>../${project.artifactId}.ui/src-gen</directory>
19     <excludes>
20         <exclude>.gitignore</exclude>
21     </excludes>
22 </fileset>
23 <fileset>
24     <directory>../../tests/${project.artifactId}.tests/src-gen</directory>
25     <excludes>
26         <exclude>.gitignore</exclude>
27     </excludes>
28 </fileset>
29 </filesets>
30 </configuration>
31 </plugin>

```

When the workflow for an Xtext project is executed, Xtext generates code not only into the runtime project, but also into the UI and Tests project. When these projects are build in a multi-module build, each project is built one after the other, and all required lifecycle phases are executed per project. This has a special consequence for the clean phase: When Xtext is generating code into the UI module, and the UI module is cleaned after the runtime module was build, the code is removed again and compilation would fail. Further, if we clean the project, we also want that the generated code in the UI project is removed. To solve this build issue, the runtime project has to configure the **maven-clean-plugin** to clean up sources also from the dependend projects. Normally we would need to configure the UI and Tests project to skip cleaning, but the code is generated to the **src-gen** folder, which is not recognized as an output folder. By default, the clean plugin just removes the target folder. Often, project structures are “mavenized” to follow a standard layout. Then the sources would be in **src/main/java**, and generated sources below **target** (e.g. **target/generated/java**). And then we would the mentioned issue that cleaning must be skipped for the dependend projects.

```

1 <plugin>

```



```

2   <groupId>org.fornax.toolsupport</groupId>
3   <artifactId>fornax-oaw-m2-plugin</artifactId>
4   <executions>
5     <execution>
6       <id>xtext</id>
7       <phase>generate-sources</phase>
8       <goals>
9         <goal>run-workflow</goal>
10      </goals>
11     <configuration>
12       <workflowEngine>mwe2</workflowEngine>
13       <workflowDescriptor>org.eclipse.xtext.example.ql.GenerateQlDsl</workflowDescriptor>
14       <timestampFileName>xtext-generator.timestamp</timestampFileName>
15       <jvmSettings>
16         <fork>true</fork>
17         <jvmArgs>
18           <jvmArg>-Xms100m</jvmArg>
19           <jvmArg>-Xmx700m</jvmArg>
20           <jvmArg>-XX:MaxPermSize=128m</jvmArg>
21           <jvmArg>-Dlog4j.configuration=file:${basedir}/META-INF/log4j.properties</jvmArg>
22         </jvmArgs>
23       </jvmSettings>
24     </configuration>
25   </execution>
26 </executions>
27 </plugin>

```

The `fornax-oaw-m2-plugin` plugin is responsible to invoke the MWE2 workflow `GenerateQlDsl.mwe2`. The `module` value in this workflow is configured as parameter `workflowDescriptor` to this plugin.

The plugin is executed in the `generate-sources` lifecycle phase, which is processed before compilation. You will see this output when it is executed:

```

1 [INFO] --- fornax-oaw-m2-plugin:3.4.0:run-workflow (xtext) @ org.eclipse.xtext.example.ql
   ---
2 [INFO] Fornax Model Workflow Maven2 Plugin V3.4.0
3 [INFO] Executing workflow in forked mode.
4 [INFO] Workflow 'org.eclipse.xtext.example.qls.GenerateQlsDsl' finished.

```

Last but not least the `xtend-maven-plugin` is configured:

```

1 <plugin>
2   <groupId>org.eclipse.xtend</groupId>

```

```
3 <artifactId>xtend-maven-plugin</artifactId>
4 </plugin>
```

This plugin compiles the Xtend files to Java files. It does some fancy stuff, since Xtend files might have references to Java classes which are not compiled or might even not be compilable before Xtend classes are compiled. The plugin therefore scans Xtend files for Java type references and precompiles stub classes. These stubs are not complete, but enough for Xtend to enable cross referencing these classes. Later, when Xtend has created the Java sources, they are compiled together with the other Java sources by the compiler plugin.

5.2.5 QL UI Project POM

The POM for the UI project is surprisingly simple. Nothing special has to be configured.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
   maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.eclipse.xtext.example.ql</groupId>
7     <artifactId>parent</artifactId>
8     <version>1.0.0-SNAPSHOT</version>
9     <relativePath>../../pom.xml</relativePath>
10  </parent>
11
12  <artifactId>org.eclipse.xtext.example.ql.ui</artifactId>
13  <packaging>eclipse-plugin</packaging>
14 </project>
```

5.2.6 SDK Feature POM

Not mentioned yet, but we have prepared also a feature project that bundles the QL and QLS plugins into one feature. The feature project is `org.eclipse.xtext.example.ql.sdk`, which can be found in the projects folder of the Git repository.

For feature projects, the packaging type is `eclipse-feature`. Nothing special has to be configured further.

```
1 <?xml version="1.0" encoding="UTF-8"?>
```

```

2 <project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
  maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.eclipse.xtext.example.ql</groupId>
7     <artifactId>parent</artifactId>
8     <version>1.0.0-SNAPSHOT</version><!--X-->
9     <relativePath>../..pom.xml</relativePath>
10  </parent>
11
12  <artifactId>org.eclipse.xtext.example.ql.sdk</artifactId>
13  <packaging>eclipse-feature</packaging>
14 </project>

```

5.2.7 p2 Repository

The result of the build should be a p2 repository, which the user can configure as update site. The repository project `org.eclipse.xtext.example.ql.repository`⁴² contains a `category.xml` file, which references the feature that should be available on this repository:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <site>
3   <feature url="features/org.eclipse.xtext.example.ql.sdk_1.0.0.qualifier.jar" id="org.
4     eclipse.xtext.example.ql.sdk" version="1.0.0.qualifier">
5     <category name="DSL"/>
6   </feature>
7   <category-def name="DSL" label="DSL"/>
8 </site>

```

The POM for this project has the packaging type `eclipse-repository`, besides that it is as simple as the previous POM.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
3   maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5   <modelVersion>4.0.0</modelVersion>
6   <parent>
7     <groupId>org.eclipse.xtext.example.ql</groupId>
8     <artifactId>parent</artifactId>

```

⁴²<http://tinyurl.com/cho28ox>

```

8   <version>1.0.0-SNAPSHOT</version><!--X-->
9   <relativePath>../../pom.xml</relativePath>
10  </parent>
11
12  <artifactId>org.eclipse.xtext.example.q1.repository</artifactId>
13  <packaging>eclipse-repository</packaging>
14 </project>

```

The resulting repository is then available in the `target/repository` folder.

5.2.8 Continuous Integration

For continuous integration we use the build infrastructure from Cloudbees, which is free for open-source software. Cloudbees offers Jenkins as CI server, and we have set up a build job for the DSL projects: <https://kthoms.ci.cloudbees.com/job/lwc13-dsl/>

The screenshot shows the Jenkins web interface for the project 'lwc13-dsl'. The browser address bar shows the URL <https://kthoms.ci.cloudbees.com/job/lwc13-dsl/>. The page has a navigation bar with links like 'ClickStart', 'Apps', 'DBs', 'Builds', 'Repositories', 'Services', and 'Support'. The 'Builds' tab is active.

On the left sidebar, there are links for 'Zurück zur Übersicht', 'Status', 'Änderungen', 'Arbeitsbereich', 'Jetzt bauen', 'Projekt Löschen', 'Konfigurieren', 'Module', 'Groups', 'Roles', 'Move', 'Forge Hook Log', and 'Git Abfrage-Protokoll'.

The main content area is titled 'Projekt lwc13-dsl'. It includes a section for 'Arbeitsbereich' (Workspace) with a list of 'Letzte erfolgreiche Artefakte' (Latest successful artifacts) showing files like 'artifacts.jar', 'content.jar', and various SDK and UI JARs with their sizes. There is also a 'Letzte Änderungen' (Latest changes) section.

Below the workspace, there is a 'Build-Verlauf' (Build history) table with columns for build number, status, and timestamp. The table shows builds #1 through #8, with build #8 being the most recent and successful. The table also includes links for 'RSS aller Builds' and 'RSS der Fehlschläge'.

At the bottom, there is a 'Permalinks' section with links to specific builds, such as 'Letzter Build (#8), vor 3 Stunden 40 Minuten' and 'Letzter stabiler Build (#8), vor 3 Stunden 40 Minuten'.

This build could be reproduced on any Jenkins server with the following configuration settings:

- Create a job of style “Maven2/3”
- Git repository URL:
`https://code.google.com/a/eclipselabs.org/p/lwc13-xtext/`
- Build-trigger: poll repository every 30 minutes

```
1 */30 * * * *
```

- In the Maven build section, choose a Maven 3 installation
- Configure POM: `projects/pom.xml`
- Goals and options: `clean install`
- Alternative settings file (Advanced options): `devenv/lwc13.devenv/settings.xml`
- Add post-build action - archive artifacts: `projects/*/target/repository/**`

6 Closing Words

Thank you for reading this document. We have been writing it with the intention that it should provide easy access for first-time users of Xtext to this powerful Language Workbench. This explains also the size of the document. We could have written it shorter, if we would assume more background knowledge of the potential readers, or if we would leave many things with just short comments.

Xtext has grown over years, gaining more and more experience from real-life projects that want to leverage the power of DSLs in integrated environments. We could only touch the surface of Xtext and Xtend here, and tried to choose the most simple solution. The QL assignment did fit quite well into what Xtext can provide mostly out-of-the-box, but its real power is unveiled when DSLs have non-standard requirements. Almost every piece in Xtext is highly customizable and it is one of the most flexible frameworks we know of. This flexibility comes to the price of complexity. Xtext is well documented by the reference manual, several blogs show advanced concepts in detail, and hundreds of projects solve different real-world requirements. Many open projects exist which are worth to study. Learning Xtext does not mean following a single tutorial, it needs training. The sources and the debugger are valuable friends when solving issues. The community around Xtext is very helpful and likely the largest of all language workbenches.

We hope that this document was helpful for you to understand some concepts of Xtext and gave you the right level of abstraction to learn how to use this tool. If we helped you to decide to use Xtext or Xtend in your project, then it was worth the effort and we would be pleased to hear from you!

Karsten Thoms, Johannes Dicks, Thomas Kutz

April 2013