

Xtext Tutorium

2017, Pierre Bayerl und Tim Schneider

Inhaltsverzeichnis

1 Zusammenfassung.....	2
2 Quellen, weiterführende Literatur.....	3
3 Arbeitsumfeld Eclipse.....	3
4 Eine erste Grammatik.....	4
4.1 Neues Xtext Projekt anlegen.....	5
4.2 Xtext Projekt übersetzen und Sprache ausprobieren.....	6
4.3 Eigene Grammatik eingeben (Teil 1).....	7
4.4 Eigene Grammatik eingeben (Teil 2).....	9
5 Validierung.....	15
5.1 Validierung: Beispiel.....	15
5.2 Option: Quickfixes.....	16
6 Code Generierung.....	17
6.1 Code Generierung: Beispiel.....	17
7 Xtend: Mehr Details.....	19
7.1 Extension Mechanismus.....	19
7.2 Verschiedenes.....	19
8 Modularisierung und Scoping.....	20
8.1 Referenzen in verschachtelten Strukturen.....	20
8.2 Scoping (Sichtbarkeit/Gültigkeit).....	21
8.3 Zusammenfassende erweiterte Aufgabe.....	23
8.4 Modularisierung.....	25
9 Verschiedenes.....	25
9.1 Standalone Compiler für die eigene DSL.....	25
9.2 Templates.....	26
9.3 QualifiedNameProvider.....	28
10 Fazit.....	30

1 Zusammenfassung

Dieses Dokument spiegelt die ersten Erfahrungen und Schritte wieder, die uns den Einstieg in Xtext erleichtert haben. Es gibt viele Dokumente, Einführungen und Kurse zu dem Thema (die meisten sind frei verfügbar). Ohne sich jedoch die Zeit zu nehmen, die allerersten Schritte zu verinnerlichen wird man viele Fehlschläge erleiden (insbesondere wo man was genau eingeben oder einstellen kann oder muss).

Abgrenzung: Dieses Dokument ist keine allumfassende Dokumentation des Themas. Dazu sind diverse Quellen angegeben (und entsprechend kommentiert). Darunter ein exzellenter Kurs /1/, ein aktuelles Buch /2/ und frei verfügbare Dokumente aus dem Internet (/3/, /4/, /5/ und /9/). Auch werden keine alternativen Werkzeuge zur Erstellung von DSLs betrachtet (siehe /7/ und /8/).

Ziel: Es soll demonstriert werden, wie man – ausgehend von einem vorhanden Modellierungsgedanken – die **Xtext Toolkette anwenden** kann, um domänenspezifische Modellierungswerkzeuge zu schaffen. Auch wenn man nicht „programmiert“, soll man eine Idee bekommen, welche **Werkzeuganforderungen** durch das Xtext Framework leicht zu realisieren sind.

2 Quellen, weiterführende Literatur

- /1/ H. Rentz-Reichert (Protos Software): "Entwicklung domänenspezifischer Sprachen und Code-Generatoren mit Xtext und Xtend" (Kurs, Regensburg, 2017/05).
 - *Empfehlenswerte Einführung*
 - *Architekt von eTrice (ROOM basiertes Modellierungswerkzeug):*
<http://www.eclipse.org/etrice/> (2017/06)
- /2/ L. Bettini: "Implementing Domain-Specific Languages with Xtext and Xtend - Second Edition" (Packt Publishing; 2nd Revised edition, 31. August 2016).
 - *Basis für den oben genannten Kurs*
- /3/ A. Mooij, J. Hooman: "Creating a Domain Specific Language (DSL) with Xtext",
http://www.cs.kun.nl/J.Hooman/DSL/Xtext_DSL_GettingStarted_Neon.pdf (2017/06)
 - *Tolle Einführung in das Thema (kompakt, mit vielen nützlichen Details)*
- /4/ A. Mooij, K. Triantafyllidis, J. Hooman: "Advanced Xtext Manual on Modularity",
<http://www.cs.kun.nl/J.Hooman/DSL/AdvancedXtextManual.pdf> (2017/06)
 - *Weiterführende Themen*
- /5/ Xtext Webseite: <https://eclipse.org/Xtext/> (2017/06)
- /6/ Hadi Hariri: „Kotlin for C++ Developers“, Overload Journal #139, June 2017;
<https://accu.org/var/uploads/journals/Overload139.pdf>, 2017/06)
- /7/ <https://www.heise.de/developer/artikel/Werkzeuge-fuer-domaenenspezifische-Sprachen-227190.html> (2017/06)
- /8/ <https://www.heise.de/developer/artikel/Episode-10-Modellierung-im-Softwarearchitekturumfeld-Teil-1-353335.html> und
<https://www.heise.de/developer/artikel/Episode-11-Modellierung-im-Softwarearchitekturumfeld-Teil-2-353355.html> (2017/06)
- /9/ <https://www.heise.de/developer/artikel/Eine-eigene-Programmiersprache-mit-Xtext-modellieren-3180566.html> (2017/06)
- /10/ Xtend Sprach-Referenz: <https://eclipse.org/xtend> (2017/07)
- /11/ http://www.eclipse.org/Xtext/documentation/310_eclipse_support.html#templates (2017/07)

3 Arbeitsumfeld Eclipse

Um die Übungen in diesem Dokument durchzuführen, muss man Eclipse mit den notwendigen Xtext Plugins installiert haben (Aktuelle Version: Neon).

Hinweise: **(1)** Der notwendige Antlr Parser darf nicht mit Eclipse zusammen verteilt werden (Open Source Lizenz passen nicht zusammen). Er muss daher ggf. separat installiert werden (<http://download.itemis.de/updates>), sonst will Eclipse ihn bei jedem neuen Xtext Projekt neu herunterladen. **(2)** Bug im Neon: Wenn CTRL-Space und andere CTRL-Tastenkombinationen nicht mehr funktionieren, muss man den „Welcome Screen“ deaktivieren (Eclipse neu starten, „Always show Welcome on start up“ deaktivieren und erneut neu starten). **(3)** In den folgenden Abschnitten sind Übungs-Meilensteine mit folgendem Symbol dargestellt:



4 Eine erste Grammatik

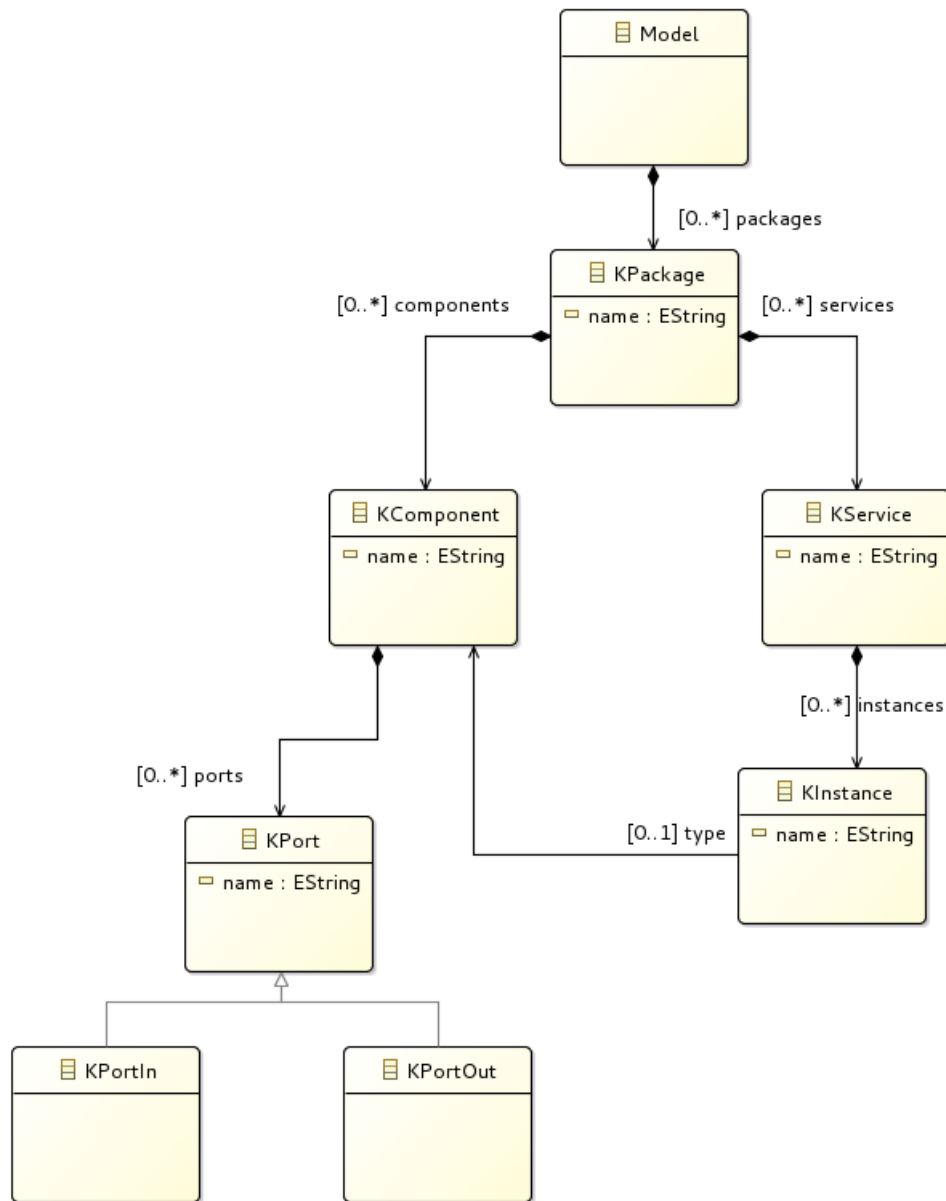
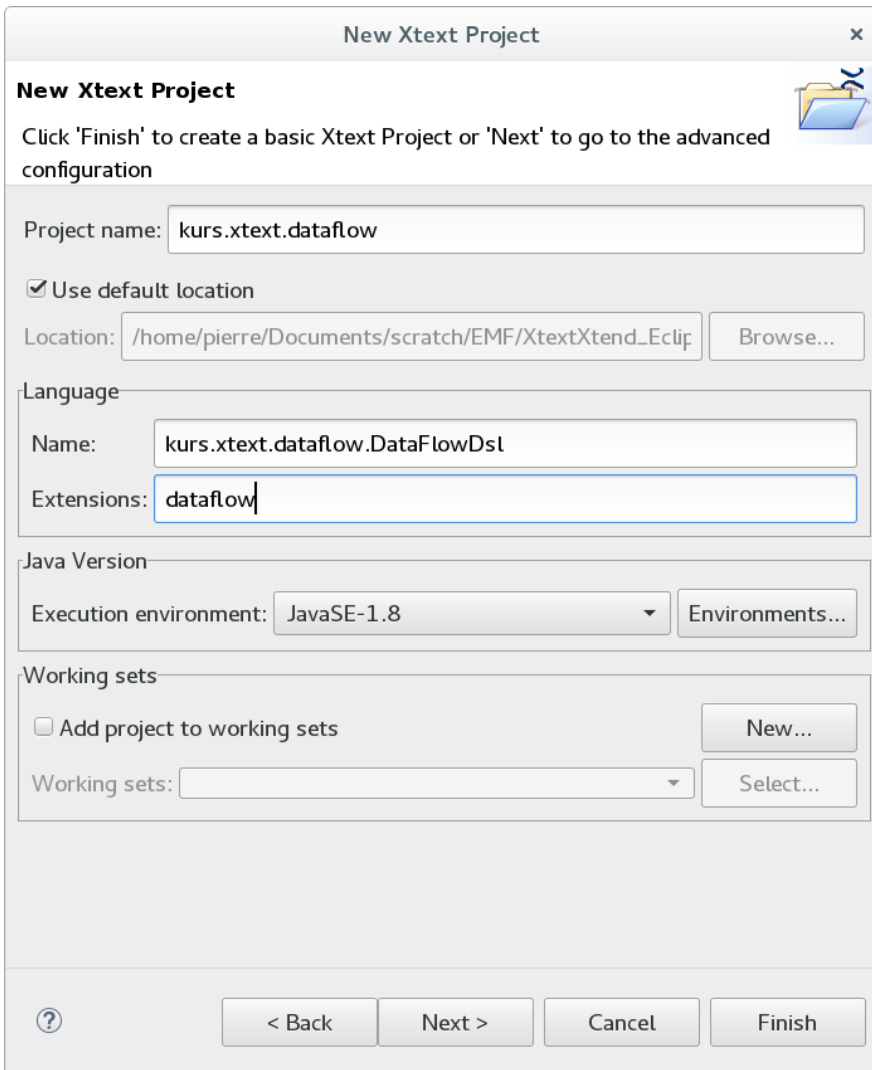


Illustration 1: Ziel-Metamodell

Ziel dieser Übung ist es, eine Grammatik für obigen Sachverhalt zu definieren: Ein **Modell**, dass **Packages** beinhaltet. Jedes Package wiederum kann **Komponenten definieren und instanziiieren**. Weiter kann eine Komponentendefinition **Eingabe und Ausgabe Ports** beinhalten. Man soll sich dabei an die Syntax der Grammatik gewöhnen. Diese ist das Mittel, um das Metamodell für die Struktur des zu definierenden Modells zu spezifizieren.

4.1 Neues Xtext Projekt anlegen



New Xtext Project

Click 'Finish' to create a basic Xtext Project or 'Next' to go to the advanced configuration

Project name:

☒ Use default location

Location:

Language

Name:

Extensions:

Java Version

Execution environment:

Working sets

☐ Add project to working sets


Working sets:



1. Im „Package Explorer“: New / Other / Xtext Project
2. Nun „**Project name**“=“kurs.xtext.dataflow“, „**Name**“=“ kurs.xtext.dataflow.DataFlowDsl“ und „**Extensions**“=“dataflow“ eingeben und „**Finish**“ klicken.

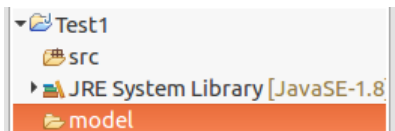
Es entstehen mehrerer Projekte (Details: siehe /3/), von denen für uns zunächst nur das erste von Interesse ist. Es wird eine Beispiel-Grammatikdefinition erstellt (Hello World) und im Editor geöffnet.

4.2 Xtext Projekt übersetzen und Sprache ausprobieren

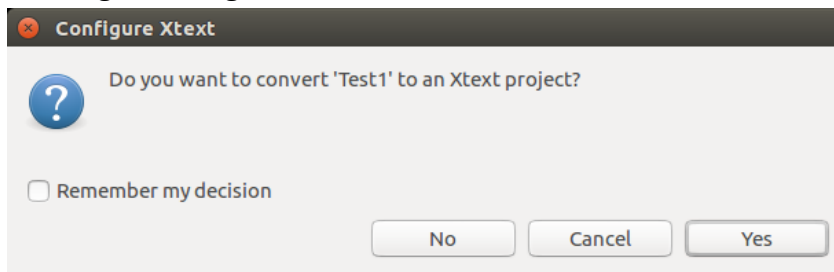
1. In die Grammatik mittels „Rechts-Klick“: **Run As / Generate Xtext Artifacts**
2. Im Menu: **Run / Debug Configurations ...**
 - „Eclipse Application“ anwählen
 - Icon „New Launch Configuration“ klicken 
 - Optional:
 - „Name“ anpassen und
 - die „Location“ anpassen (Arbeits Workspace zum Probieren der neuen Plugins)
 - „Debug“ klicken

Nun geht eine neue Eclipse-Instanz auf. Hier kann man nun die neue Sprache ausprobieren:

1. Im „Package Explorer“: **New Project** klicken.
2. „Java Project“ wählen und entsprechend ein Projekt anlegen (z.B. „Test1“) - Es ist dabei jedoch irrelevant, ob man hier Java oder z.B. C++ wählt.
3. In diesem Projekt nun einen **Ordner „model“ anlegen**:



4. In diesem Ordner nun eine Modelldatei anlegen: **New / File** mit „File Name“=“test1.dataflow“ und „Finish“ klicken.
5. Die folgende Frage mit „Yes“ beantworten:



Nun können Sie die Hallo Welt Grammatik ausprobieren
(probieren Sie CTRL-Space aus) und geben Sie dann folgendes Beispiel ein:

```
Hello Pierre!  
Hello Tim!
```



4.3 Eigene Grammatik eingeben (Teil 1)

Eine Grammatik beschreibt die Syntax (quasi ohne Semantik) und erzeugt intern ein ecore-Modell, dass man sich anschauen kann. Der „Master“ ist jedoch die Grammatik (für die Wartung).

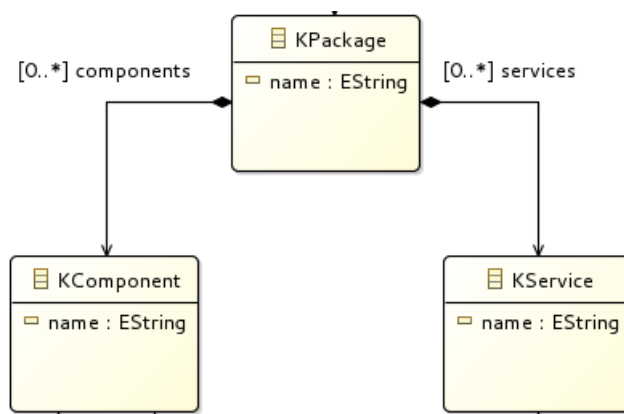
Hinweis: Es ist auch möglich, eine Grammatik aus einem ecore-Modell zu generieren (via Eclipse Wizard).

4.3.1 Basis Knoten für das Modell

- Die erste Regel in der Grammatik („Modell“ in der Hallo Welt Grammatik) beschreibt den Startpunkt des Modells (das Wurzelement).
- Statt der Modellelemente vom Typ „Greeting“ wollen wir nun „KPackage“-Elemente definieren (vgl. Illustration 1).
- Siehe auch: /3/

4.3.2 Attribute, Komposition

Ein KPackage wiederum soll eine Komposition aus KComponent und KService Elementen darstellen:



```
KPackage: 'package' name=ID '{'
    (
        components += KComponent |
        services += KService
    )*
    '}'
;
```

- 'package', '{', '}'
 - ist jeweils ein **Schlüsselwort**
- name=ID
 - „name“ ist **Attribut** im Metamodell (Das Attribut „name“ hat eine Sonderbedeutung: Es beschreibt den Namen des Elements).
 - „=“ bedeutet: es ist ein **Wert** (keine Liste von Werten)

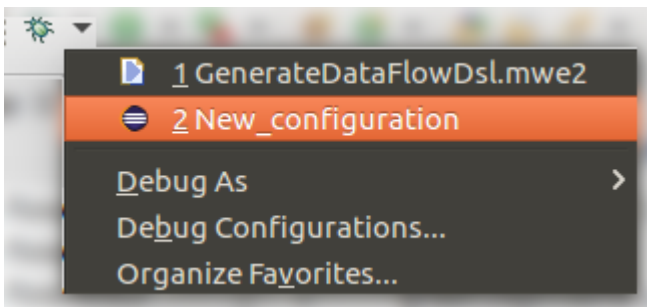
- **ID ist ein Terminal** (siehe **grammar** kurs.xtext.dataflow.DataFlowDsl **with** org.eclipse.xtext.common.Terminals - „F3“ auf Terminals klicken, um die Terminals zu sehen)
- components += KComponent
 - „components“ ist ein **Attribut** im Metamodell
 - „+=“ bedeutet es ist eine **Liste**, an die hinzugefügt wird
 - „KComponent“ und „KService“ sind Definitionen analog zu einem „KPackage“
- (... | ...)*
 - Alles innerhalb der Klammer ist 0 bis n Mal erlaubt („+“ statt „*“ bedeutet 1 bis n Mal).
 - „|“ ist hier ein „oder“

Folgende Eingaben sollen möglich werden:

```
package test1 {
    Component PC {
    }
    Component DF {
    }
    service MyService {
    }
}
```



→ Testen Sie Ihre Grammatik (wie oben beschrieben: „**Run As / Xtext Artifacts**“ und dann die angelegte Debugkonfiguration via „Icon“ in der Iconleiste starten).

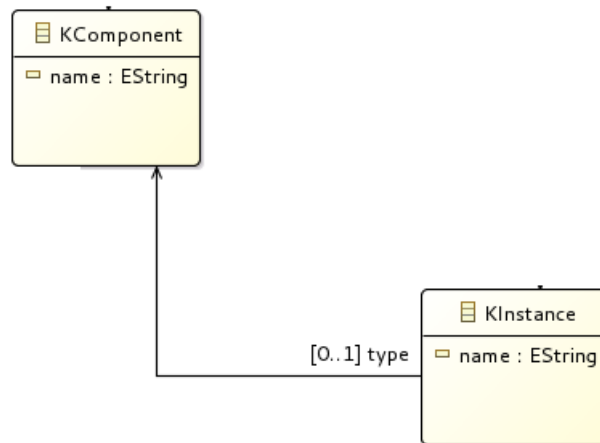


4.4 Eigene Grammatik eingeben (Teil 2)

Es folgen nun weitere Möglichkeiten, um das Metamodell zu spezifizieren.

Weitere Informationen entnehmen Sie den Referenzen. Weiter ist eine Offline Hilfe in Eclipse integriert: „Help / Help Contents“

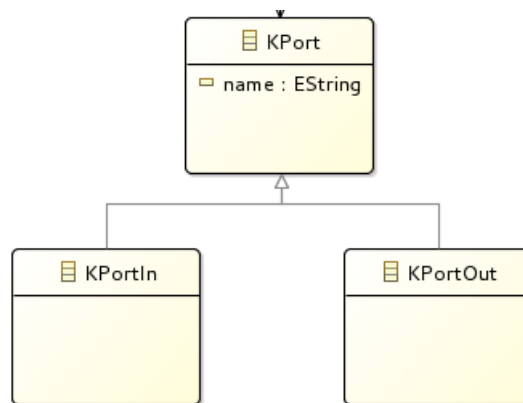
4.4.1 Referenzen



```
KInstance: 'instance' name=ID ':' type=[KComponent];
```

- `type=[KComponent]`
 - „type“ ist ein Attribut (ein Referenzattribut) im Metamodell
 - „[KComponent]“ ist eine **Referenz** auf ein „KComponent“
 - Siehe auch: /1/, /3/, /2/

4.4.2 Vererbung



```
KPort:
    KPortIn|KPortOut
;
KPortIn:
    'port_in' name=ID '{'
    '}'
;
KPortOut:
    'port_out' name=ID '{'
    '}'
;
```

- „Base: SpecialA|SpecialB;“
 - Definiert eine Basisklasse ausgehend von den Spezialisierungen.
 - Gemeinsame Attribute wandern dabei automatisch in die Basisklasse (hier: „name“)
 - Siehe auch: /1/, /3/, /2/

4.4.3 Editor

Nun müssen Sie noch den Komponenten (KComponent) Ports und den Services (KService) Instanzen hinzufügen (vgl. Illustration 1).

Probieren Sie nun Ihre Grammatik aus (Abschnitte 4.4.1, 4.4.2, 4.4.3):

```
package test1 {  
    Component PC {  
        port_in in {}  
        port_out out {}  
    }  
    Component DF {  
        port_in in {}  
        port_out out {}  
        port_out debug {}  
    }  
    service MyService {  
        instance pc : PC  
        instance df : DF  
    }  
}
```



Die Eingaben sind analog ebenso als Baum verfügbar und zeitgleich editierbar (Die Datei mit „Open With...“ „Sample Ecore Model Editor“ öffnen, verändern, speichern):

▼ platform:/resource/Test1/model/test1.dataflow

▼ Model

▼ KPackage test1

▼ KComponent PC

◆ KPort In in

◆ KPort Out out

▼ KComponent DF

◆ KPort In in

◆ KPort Out out

◆ KPort Out debug

▼ KService MyService

◆ KInstance pc

◆ KInstance df

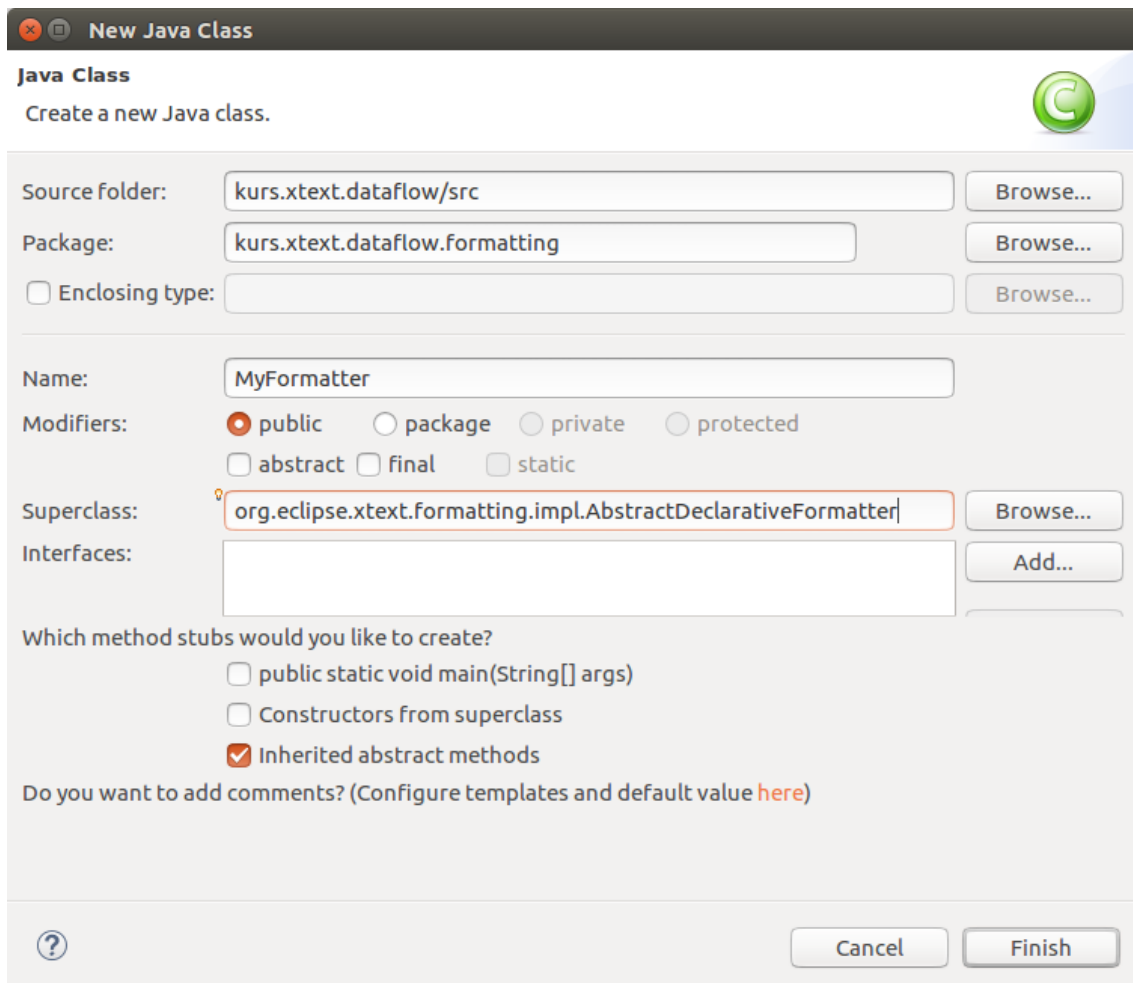
4.4.4 Mehr Grammatik...

Die Grammatik bietet noch ein paar mehr Facetten, die in diesem Rahmen nicht angesprochen werden. Siehe dazu die Xtext Hilfe in Eclipse, oder /2/, /3/, /5/.

4.4.5 Option: Auto Formatting

Hinweis: Wenn Sie in der Baumansicht neue Objekte anlegen und abspeichern, werden Sie sehen, dass der damit erzeugte Text keine Zeilenumbrüche besitzt. Auch das Auto-Formatting von Eclipse erzeugt dieses Problem (**CTRL-SHIFT-F** im Text Editor; mit CTRL-Z rückgängig machen). Dieses Verhalten kann man einfach umkonfigurieren, indem man sich um „Xtext Formatting“ kümmert:

- Legen Sie eine neue Java Klasse „**kurs.xtext.dataflow.formatting.MyFormatter**“ im Ihrem Projekt mit der Grammatik an, das von „**org.eclipse.xtext.formatting.impl.AbstractDeclarativeFormatter**“ abgeleitet ist.



New Java Class

Java Class
Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ package ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

- Geben Sie hier folgenden Code ein, um u.a. das Verhalten um „{“ und „}“ zu steuern:

```
package kurs.xtext.dataflow.formatting;

import org.eclipse.xtext.Keyword;
import org.eclipse.xtext.formatting.impl.AbstractDeclarativeFormatter;
import org.eclipse.xtext.formatting.impl.FormattingConfig;
import org.eclipse.xtext.util.Pair;

import kurs.xtext.dataflow.services.DataFlowDslGrammarAccess;

public class MyFormatter extends AbstractDeclarativeFormatter {

    @Override
    protected void configureFormatting(FormattingConfig config) {

        DataFlowDslGrammarAccess ga = (DataFlowDslGrammarAccess) getGrammarAccess();

        for (Pair<Keyword, Keyword> pair : ga.findKeywordPairs("{", "}")) {
            config.setLinewrap().after(pair.getFirst());
            config.setIndentationIncrement().after(pair.getFirst());
            config.setLinewrap().before(pair.getSecond());
            config.setIndentationDecrement().before(pair.getSecond());
            config.setLinewrap().after(pair.getSecond());
        }
        config.setLinewrap(1, 1, 1).after(ga.getKInstanceRule());
    }
}
```

- Diesen neuen Formatter müssen Sie nun noch anmelden. Geben Sie in der Datei kurs.xtext.dataflow.DataFlowDslRuntimeModule.xtend folgenden Code ein:

```
override Class<? extends org.eclipse.xtext.formatting.IFormatter> bindIFormatter() {
    MyFormatter
}
```

(Hinweis: Xtend ist eine Xtext Sprache, die direkt in Java Code umgewandelt wird. Jede Xtend Klasse kann auch in Java geschrieben werden. Xtend bietet viele Eigenschaften, die insbesondere Codegenerierung und kompakte Schreibweisen erleichtern – die man auch in anderen modernen Hype-Sprachen – wie z.B. Kotlin o.ä. wiederfindet; vgl. z.B. /6/.

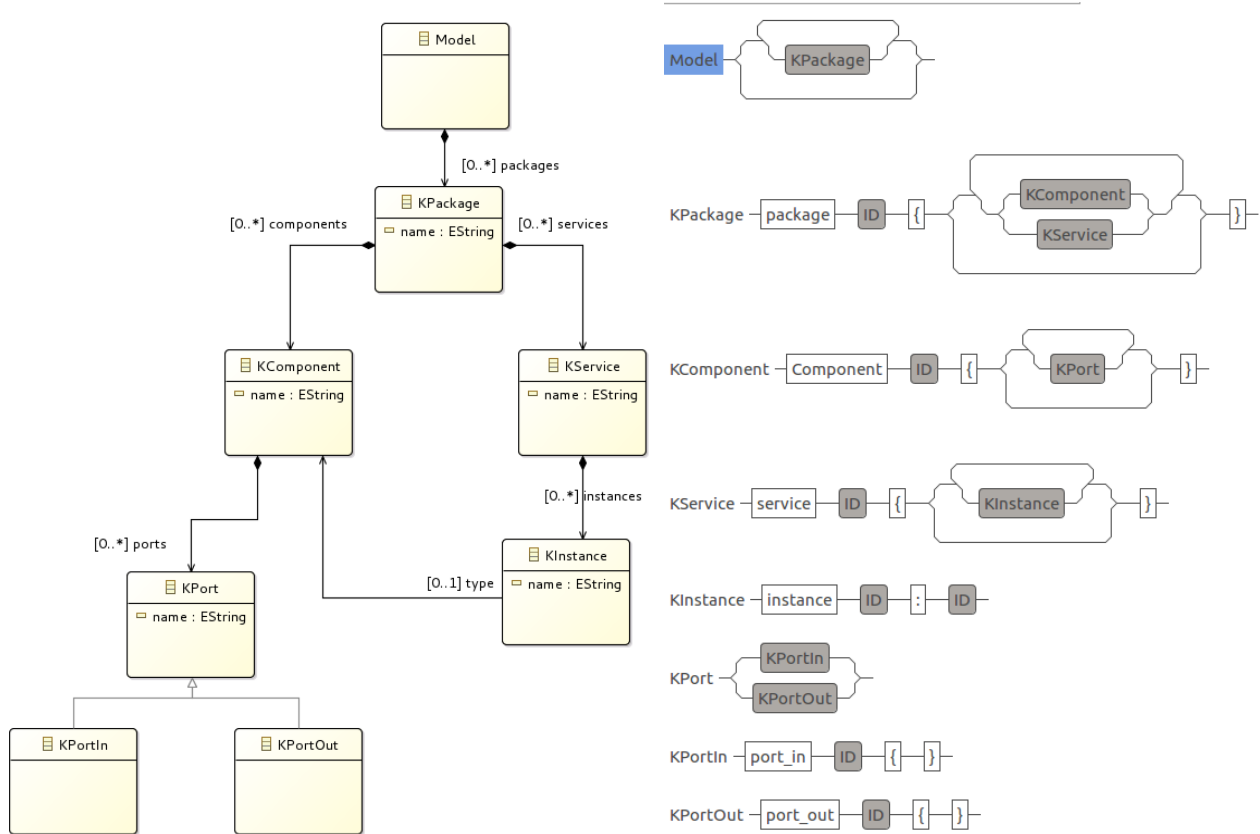
- Starten Sie Ihre Arbeits Workbench nun neu und probieren Sie den neuen Formatter aus (CTRL-SHIFT-F).

Weitere Informationen: /1/ und JavaDoc von FormattingConfig

(<http://download.eclipse.org/modeling/tmf/xtext/javadoc/2.3/org/eclipse/xtext/formatting/impl/FormattingConfig.html>, 2017/06).

4.4.6 Option: Meta Modell visualisieren

Man kann das Ecore Metamodell, das durch die Grammatik definiert wird, visualisieren: siehe /3/, Kapitel „3.1.3 Optional: Ecore diagram“. Weiter kann man die Grammatik selbst graphisch darstellen: siehe /3/, Kapitel „3.1.2 Construct grammar of DSL“, Abschnitt „Optional: View Diagram of Xtext Grammar“.



Links: Erzeugtes Ecore Diagramm („Initialize Ecore Diagram ...“)

Rechts: Syntax Graph der Grammatik (Window/Show View/Other.../Xtext/Xtext Syntax Graph)

5 Validierung

Sie können nun Ihre Grammatik validieren. Hier kann durch explizite programmatische Regeln das eingegeben Modell geprüft werden. Fehler oder Warnungen werden an Modellelementen angehängt und werden im Eclipse Editor angezeigt. Weitere Informationen (u.a.): siehe /3/.

In /1/ wurde suggeriert, in die Grammatik möglichst wenig Semantik einzubauen (um diese einfach zu halten). Also z.B. lieber „*“ statt „+“ und entsprechende Semantik in die Validierung zu stecken.

5.1 Validierung: Beispiel

Öffnen Sie die Datei `kurs.xtext.dataflow.validation.DataFlowDslValidator.xtend`, kommentieren Sie den auskommentierten Teil ein und modifizieren Sie ihn:


```
package kurs.xtext.dataflow.validation

import org.eclipse.xtext.validation.Check
import kurs.xtext.dataflow.dataFlowDsl.KComponent
import kurs.xtext.dataflow.dataFlowDsl.DataFlowDslPackage

class DataFlowDslValidator extends AbstractDataFlowDslValidator {

    public static val INVALID_NAME = 'invalidName'

    @Check
    def checkComponentStartsWithCapital(KComponent c) {
        if (!Character.isUpperCase(c.name.charAt(0))) {
            warning('Name should start with a capital',
                DataFlowDslPackage.Literals.KCOMPONENT__NAME,
                INVALID_NAME)
        }
    }
}
```



Probieren Sie die Regel im Editor aus.

Hinweis:

- Eine Regel wird hier durch die Annotation „@Check“ als Regel markiert und bekommt ein Modellelement als Eingabe.
- Die Regel wird bei der Eingabe geprüft (@Check bietet hier noch weitere Optionen für langsame Checks: während der Eingabe, beim Speichern und auf Anfrage – siehe `org.eclipse.xtext.validation.CheckType`; „F3“ auf @Check anwenden)
- Die Regel betrifft Modellelemente vom Typ „KComponent“ (Parameterübergabe). An diesen Elementen wird der Fehler auch aufgehängt.
- Die hier verwendete Sprache ist Xtend. Alternativ kann die Regel auch in Java programmiert werden.
- Eine Regel kann „errors“, „warnings“ und „infos“ erzeugen, welche über IDs identifizierbar werden (bei uns „INVALID_NAME“) - damit kann man später Hotfixes mit den Meldungen verknüpfen. Weiter können Stringinformationen an die Hotfixes mit übergeben werden.

5.2 Option: Quickfixes

Öffnen Sie `kurs.xtext.dataflow.ui.quickfix.DataFlowDslQuickfixProvider.xtend` im **ui-Projekt**. Hier können Sie Quickfixes implementieren.

Beispiel (der in der Datei schon vorhandene Beispiel-Code kann in unserem Beispiel quasi ohne Anpassung aktiviert werden):

```
@Fix(DataFlowDslValidator.INVALID_NAME)
def capitalizeName(Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, 'Capitalize name', 'Capitalize the name.', 'upcase.png') [
        context |
        val xtextDocument = context.xtextDocument
        val firstLetter = xtextDocument.get(issue.offset, 1)
        xtextDocument.replace(issue.offset, 1, firstLetter.toUpperCase)
    ]
}
```

Im vorgegebenen Beispiel bekommt der Quickfix direkten Zugriff auf den Text, auf den er angewendet werden soll. Über die ID „INVALID_NAME“ ist der Quickfix mit einem Validierungsproblem verknüpft.

Alternativ kann man auch den Zugriff auf das Modellelement bekommen, das das Validierungsproblem verursacht hat:

```
@Fix(DataFlowDslValidator.INVALID_NAME)
def capitalizeName2(Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, 'Capitalize name 2', 'Capitalize the name.', 'upcase.png') [
        element, context |
        val c=element as KComponent
        c.name = c.name.toFirstUpper
    ]
}
```

Hinweis:

- Die Annotation „Fix“ markiert die Methode als Quickfix und verknüpft diesen Quickfix mit einem Validierungsproblem über eine entsprechende eindeutige ID.
- Weitere Anmerkungen zu Xtend (z.B. Bedeutung von „val“, „as“, ...) siehe Kapitel 6.

6 Code Generierung

Die Codegenerierung wird bei Xtext typischerweise in Xtend programmiert.

Arbeitet man in der Arbeitsworkbench im Debug Modus, kann man die Code Generierung normalerweise ohne Neustart dieser Workbench anpassen.

Die Code Generierung in der Arbeitsworkbench für eine Modelldatei entspricht dem Compilieren einer Java Datei in einem Java Projekt. Normalerweise passiert das Code Generieren beim Abspeichern der Modelldatei (auto-compile).

6.1 Code Generierung: Beispiel

Öffnen Sie `kurs.xtext.dataflow.generator.DataFlowDslGenerator.xtend`. Hier kann ein erster Code Generator implementiert werden:

Hier bekommt man insbesondere die Modellinformation über eine sogenannte Ressource (Parameter **resource**) und eine Möglichkeit mit dem Dateisystem (Parameter **fsa**) zu interagieren.

```
override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
    IGeneratorContext context) {
    // -----
    // Option 1:
    var txt = ""
    for (obj: resource.allContents.toIterable) {
        if (obj instanceof KComponent) { // inside this "if" obj is a KComponent
            if (txt.length>0) txt = txt + ", "
            txt = txt + obj.name;
        }
    }
    fsa.generateFile('Components1.txt', 'Component declarations: ' + txt);

    // -----
    // Option 2:
    fsa.generateFile('Components2.txt', 'Component declarations: ' +
        resource.allContents
            .filter[ obj | obj instanceof KComponent ]
            .map[ obj | return (obj as KComponent).name ]
            .join(', '))

    // -----
    // Option 3:
    fsa.generateFile('Components3.txt', 'Component declarations: ' +
        resource.allContents.filter(KComponent).map[name].join(', '))
}
```

Es wird auf drei verschiedene Weisen eine Liste der Komponenten in einem Modell erzeugt:

1. Eine traditionelle for-each Schleife
 - Besonderheit „auto-casting“ (wie z.B. bei Kotlin: /6/): wenn man in einer Bedingung sicherstellt, dass ein Objekt einen bestimmten Typ hat (if „instanceof“), dann wird das Objekt in diesen Typ automatisch gecastet.
 - Strichpunkte sind optional.
 - „var“ und „val“ dienen dazu, Variablen zu deklarieren (val = finale Variablen).
2. Eine Filter/Map/Join Kombination mit expliziten Lambda („[param | code]“)
 - Erwartet eine Methode eine Funktion als Parameter, kann eine Lambdafunktion (in eckigen Klammern) direkt nach dem Funktionsaufruf angegeben werden (runde Klammern kann man dabei weglassen). Die Parameter der Lambdafunktion werden mit einem „|“ vom Rumpf getrennt.
3. Eine Filter/Map/Join Kombination mit kompakten Lambdas und Klassenfilter:
 - `filter(Klasse)` filtert direkt nach einer Klasse

- map: hier ist der Parameter der Lambdafunktion implizit geben (mit „it“ ansprechbar: z.B. it.name). Weiter wird dieser implizite Parameter automatisch verwendet („name“ entspricht also „it.name“):
„[a | a.name]“ → „[name]“ oder „[it.name]“
- „return“ kann man weglassen: Der letzte Befehl bildet den Rückgabewert.

Die erzeugten Dateien beinhalten alle folgenden Text:

Component declarations: PC, DF

Weiter bietet die Sprache Xtend eine spezielle Template-Engine an, mit der man einfach Text- und Generator-Code mischen kann:

```
// -----
// Option 4:
val model = resource.contents.get(0) as Model
fsa.generateFile('Components4.txt', '''
The Components of the model are:
«FOR p: model.packages»
    Package «p.name»
    «FOR c: p.components»
        KComponent «c.name»
        «FOR port: c.ports SEPARATOR ", "»
            - with port «port.name»
        «ENDFOR»
    «ENDFOR»
«ENDFOR»
''');
```

Drei einfache Anführungszeichen werden für solche Texte verwendet. Im Text können innerhalb französischer Anführungszeichen Befehle gesetzt werden (z.B. «p.name», CTRL-< und CTRL->). Weiter werden Einrückungen intelligent behandelt: Für den Text relevante Teile sind grau hinterlegt. Hinweise:

- Dieses Beispiel zeigt auch, wie man das Wurzelement des Modells bekommt (val model=...).
- Weiter zeigt es, wie man bei Xtend castet: „obj as Type“.

Ausgabe:

```
The Components of the model are:
Package test1
KComponent PC
- with port in,
- with port out
KComponent DF
- with port in,
- with port out,
- with port debug
```



7 Xtend: Mehr Details

Neben den in Abschnitt 6 skizzierten Eigenschaften von Xtend werden ein paar weitere Details der Sprache vorgestellt.

7.1 Extension Mechanismus

Xtend hat seinen Namen daher, dass man in dieser Sprache Klassen erweitern kann, ohne den Vererbungsmechanismus zu verwenden. Hierzu existieren verschiedene Möglichkeiten. Ein Beispiel einer erweiterten Klasse ist `java.lang.String`: Diese Klasse wurde beispielsweise um die Methode „`toFirstUpper`“ ergänzt.

Eine einfache Möglichkeit, dies zu bewerkstelligen ist es, eine lokal sichtbare Methode `f(TypX x, ...)` zu definieren („**local extension methods**“, /10/). Damit ist `TypX` um die Methode `f(...)` erweitert:

```
def printNTimes(String s, int n) {  
    for(var i=0;i<n;i++) println(s)  
}  
  
...  
    "Hello".printNTimes(3)  
...
```

Weitere Optionen (vgl. /10/) bieten u.a. die Möglichkeit, analog zum obigen Beispiel statische Methoden zu verwenden („**extension imports**“), die über einen speziellen import-Befehl angezogen werden: „`import static extension package.Type.printNTimes`“.

7.2 Verschiedenes

- „`==`“ vs. „`===`“ (analog „`!=`“ vs. „`!==`“):
 - „`==`“ ruft die Methode „`equals`“ auf, um die betroffenen Objekte zu vergleichen.
 - „`===`“ prüft, ob tatsächlich dasselbe Objekt referenziert ist (quasi ein Zeiger-Vergleich).
- Man kann prüfen, ob ein (optionales) Modell Element vorhanden ist, indem man das Element mit null vergleicht.
- Man kann prüfen, ob ein Modell Element noch nicht „geladen“ ist (`eIsProxy==true`).

8 Modularisierung und Scoping

Im Folgenden geht es um Techniken dateiübergreifende Modelle zu gestalten. Dazu gehört auch die Sichtbarkeit und **Gültigkeit** bestimmter Elemente (Scoping) in verschachtelten Datenmengen bzw. verschiedenen Kontexten.

8.1 Referenzen in verschachtelten Strukturen

Bisher hat man Referenzen nur innerhalb einer Ebene (innerhalb eines KPackages) verwendet. Möchte man Referenzen in andere KPackages machen, schlägt dies fehl:

```
package components {
    Component PC {
        port_in in {}
        port_out out {}
    }
    Component DF {
        port_in in {}
        port_out out {}
        port_out debug {}
    }
}
package services {
    service MyService {
        instance pc : PC
        instance df : DF
    }
}
```

❌ Errors (2 items)

- ❌ Couldn't resolve reference to KComponent 'DF'.
- ❌ Couldn't resolve reference to KComponent 'PC'.


Xtext ermöglicht es, in seinem Standardverhalten jedes Element über seinen vollen Namen analog zur Namensgebung von Java-Klassen anzusprechen (alle Namen der Elemente im Baum mit „“ voneinander getrennt: z.B. „components.PC“ im Beispiel oben).

Eine **Referenz** in der **Grammatik** (z.B. „[KComponent]“) wird jedoch standardmäßig über eine ID angesprochen (ohne „“ im Namen). Daher kann der volle Name nicht formuliert werden. Möchte man dies ermöglichen, kann man wie folgt vorgehen:

```
...
KInstance: 'instance' name=ID ':' type=[KComponent|FQN];
FQN hidden(WS): ID('.'ID)*; // allows "packagename,componame"
// (without spaces inbetween)
...
```

Damit kann man nun das obige Beispiel erfolgreich korrigieren:

```
...
package services {
    service MyService {
        instance pc : components.PC
        instance df : components.DF
    }
}
...
```



8.2 Scoping (Sichtbarkeit/Gültigkeit)

Es gibt Fälle, bei denen es nicht erwünscht ist, eine beliebige Referenz auf ein Objekt zu ermöglichen. Beispiele sind Methoden von Objektinstanzen (z.B. bei Java) oder Ports von Instanzen (in unserem Beispiel). Im ersten Beispiel möchte man nur Methoden einblenden, die der Typ der Objektinstanz beinhaltet. Im zweiten Beispiel möchte man nur Ports einblenden, die zur selektieren Instanz passen:

```
...
KService: 'service' name=ID '{'
  (
    instances += KInstance|
    connections += KConnection
  )*
'}';
KConnection:
  'connect'
  srcInstance=[KInstance] '.' srcPort=[KPortOut]
  'to'
  dstInstance=[KInstance] '.' dstPort=[KPortIn]
;
...
```

Ohne weitere Anpassung wird folgender Code nicht akzeptiert (da die Ports nicht sichtbar sind). Man könnte nun die Referenz auf die Ports wie in Kapitel 8.1 realisieren, dies würde jedoch keinen Zusammenhang zwischen den Ports und dem Typ der Instanz berücksichtigen.

```
...
    service MyService {
        instance pc : PC
        instance df : DF
        connect pc.out to df.in // Wunsch!
    }
...
```

Man kann nun diesen semantischen Zusammenhang „nur Ports zulassen, die zum Typ der Instanz gehören“ wie folgt in das Modell einbringen: Man definiert sich einen eigenen Scope-Provider (Geltungsbereich = Scope): Öffnen Sie dazu die schon vorhandene Datei `kurs.xtext.dataflow.scoping.DataFlowDslScopeProvider` und überladen Sie die Methode „`getScope`“ (vgl. /1/, /4/):

- Hier kann man über eine „referenz“ das Attribut aus dem Modell in der Grammatik selektieren, für das ein neuer Gültigkeitsbereich definiert werden soll: In unserem Beispiel ist das die Referenz auf den Quell-Port („srcPort“) und Ziel-Port („dstPort“) der `KConnection`.
- Weiter hat man den Zugriff über einen „context“ auf das betroffene Modellelement (hier die `KConnection`).
- Hat man die Liste der gültigen Einträge erstellt, gibt man diese zurück (mithilfe der statischen Methode „`Scopes.scopeFor`“; siehe /4/).

- Für alle Attribute, die man nicht gesondert behandeln möchte, gibt man das Ergebnis der Basisimplementierung zurück (vgl. Beispiel).

```
class DataFlowDslScopeProvider extends AbstractDataFlowDslScopeProvider {  
  override getScope(EObject context, EReference reference) {  
    if (reference == DataFlowDslPackage.Literals.KCONNECTION__SRC_PORT) {  
      val kconn = context as KConnection;  
      val ports = kconn.srcInstance.type.ports.filter(KPortOut)  
      return Scopes.scopeFor(ports);  
    }  
    else if (reference == DataFlowDslPackage.Literals.KCONNECTION__DST_PORT) {  
      val kconn = context as KConnection;  
      val ports = kconn.dstInstance.type.ports.filter(KPortIn)  
      return Scopes.scopeFor(ports);  
    }  
    super.getScope(context, reference);  
  }  
}
```



Aufgabe: Probieren Sie obiges Beispiel aus.

8.3 Zusammenfassende erweiterte Aufgabe

Testen Sie auch eine der folgenden (oder alle) Variationen des Beispiels aus Abschnitt 8.2:

- **Variation 1 „Ports mit Messages“:** Erlauben Sie „KMessages“ in einem „KPackage“ zu definieren (Neues Modellelement mit Syntax „**Message** <NAME>“). Assoziieren Sie nun jeden Port mit einem KMessage und erlauben Sie nur Ports zu verknüpfen, die mit derselben Message assoziiert sind (siehe ein Beispiel eines solchen Modells unten).
 - **Lösung 1:** Modifizieren Sie den Scope Provider für den Ziel Port, um dort nur zum Quell Port passende Ports anzubieten (der Quell Port ist dann schon im Modell vorhanden – andersherum gibt es Probleme)
 - **Lösung 2:** Verwenden Sie die Modell Validierung (vgl. Abschnitt 5) um lesbare Fehlermeldungen zu erzeugen.
- **Variation 2 „Vererbte Komponenten“:** Erlauben Sie KComponenten **optional** von einer anderen KComponent „**abzuleiten**“ (Syntax: „Component A **extends** B {...}“). Nun sollen die „geerbten Ports“ ebenfalls angeboten werden.
 - Hinweis: Man kann Listen mit „+“ vereinen.
 - Hinweis: Optionale Attribute in der Grammatik können mit „?“ spezifiziert werden (vgl. /3/: such nach *,as indicated by the “?” notation‘*)

Werden beide Variationen realisiert, ist folgende Eingabe möglich:

```
package test1 {  
  
    // Variation 1:  
    Message IQ_Data  
    Message PC_Data  
    Message PC_DEBUG_Data  
    Message DF_Data  
    Message DF_DEBUG_Data  
  
    Component PC {  
        port_in pcin { IQ_Data }           // Variation 1  
        port_out pcout { PC_Data }         // Variation 1  
        port_out pcdbg { PC_DEBUG_Data }   // Variation 1  
    }  
    Component ExPC extends PC {             // Variation 2  
        port_out pcout_ex { PC_Data }      // Variation 1  
    }  
    Component DF {  
        port_in dfin { PC_Data }           // Variation 1  
        port_in dfdbg { PC_DEBUG_Data }    // ...  
        port_out dfout { IQ_Data }  
        port_out dfdbg { DF_DEBUG_Data }  
    }  
    service MyService {  
        instance pc : ExPC  
        instance df : DF  
        connect pc.pcout_ex to df.dfin      // Variation 1  
        connect pc.pcdbg to df.dfdbg       // Variation 1+2 (pcdbg geerbt)  
    }  
}
```



Code Generierung:

Versuchen Sie nun aus einem Modell wie dem Folgenden eine entsprechende Ausgabe zu erzeugen (Das Modell kann mit oder ohne Messages vorliegen). Kopieren Sie dazu die Ausgabe in Ihr Code Generator Skript (zwischen die dreifachen Hochkommas) und ergänzen Sie sukzessive den Text mit Modell Informationen, FOR-Schleifen, etc. (vgl. Abschnitt 6).

Modell:

```
package test1 {  
  
    Message IQ_Data  
    Message PC_Data  
    Message DF_Data  
    Message THRESH_Data  
    Message DET_Data  
  
    Component PC {  
        port_in in { IQ_Data }  
        port_out out { PC_Data }  
    }  
    Component DF {  
        port_in in { PC_Data }  
        port_out out { DF_Data }  
    }  
    Component THRESH {  
        port_in in { DF_Data }  
        port_out out { THRESH_Data }  
    }  
    Component DET {  
        port_in in1 { THRESH_Data }  
        port_in in2 { DF_Data }  
        port_out out { DET_Data }  
    }  
  
    service MyRadarProc {  
        instance pc : PC  
        instance df : DF  
        instance thresh : THRESH  
        instance det : DET  
        connect pc.out to df.in  
        connect df.out to thresh.in  
        connect thresh.out to det.in1  
        connect df.out to det.in2  
    }  
}
```

Gewünschte Ausgabe:

```
@startuml  
package "MyRadarProc" {  
    object pc #Wheat|CornflowerBlue  
    object df #Wheat|CornflowerBlue  
    object thresh #Wheat|CornflowerBlue  
    object det #Wheat|CornflowerBlue  
    pc -> df : out > in  
    df -> thresh : out > in  
    thresh -> det : out > in1  
    df -> det : out > in2  
}  
@enduml
```

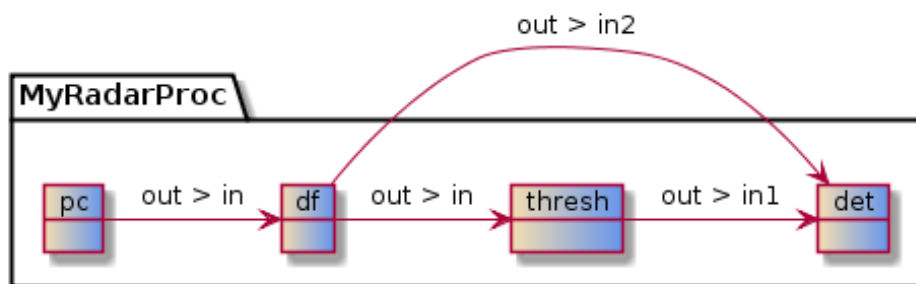
Hinweis: Die Teile der Ausgabe, die aus dem Modell stammen, sind farbig hinterlegt.



Optionale Analyse und Dokumentation:

- Erzeugen Sie das **Ecore Diagramm** für Ihre Grammatik.
- **Kommentieren** Sie Ihre Grammatik („// Kommentar“), um Verweise auf wichtige **semantische Zusätze** zu dokumentieren.

Diese Ausgabe kann man dann z.B. in <http://plantuml.com/> online eingeben um sie in ein Bild umzuwandeln (oder man installiert das PlantUML-Werkzeug):



8.4 Modularisierung

In der Standardkonfiguration kann ein Modell jederzeit Modell Elemente aus anderen Dateien anziehen, die im selben Projekt vorliegen. Alternativ kann man auch fordern, dass ein speziell zu definierender import Befehl verwendet wird, um andere Modell Dateien anzuziehen (dies sind dann Modellelemente, die ein Attribut „**importURI**“ besitzen; dieses Attribut ist ein STRING, der eine andere Datei lokalisiert). Das Vorgehen dafür ist in /4/ erklärt.

Das Einbinden anderer Modelle basierend auf anderen Metamodellen / Grammatiken ist ebenfalls möglich. Dies erfordert jedoch weitere Einstellungen. Für andere Grammatiken ist das Vorgehen in /4/ beschrieben.

Optionale Aufgabe: Teilen Sie Ihre Modelldatei auf. Probieren Sie auch die Variante mit „importURI“ aus: Lesen Sie hier in /4/ nach, was zu tun ist (Stichwort: „ImportUriGlobalScopeProvider“).

9 Verschiedenes

9.1 Standalone Compiler für die eigene DSL

Um aus dem Java Projekt einen eigenständigen Compiler zu erzeugen, ist wie folgt vorzugehen (Quelle: /2/ und persönliche Kommunikation mit /1/):

- In der mwe2-Datei (in unserem Beispiel src/kurs/xttext/dataflow/GenerateDataFlowDsl.mwe2) ist im Bereich „language“ folgender Code einzufügen: „fragment = generator.GeneratorFragment2 {generateJavaMain = true}“.

```
...
        language = StandardLanguage {
            name = "org.example.xttext.fsm.FiniteStateMachine"
            fileExtensions = "fsm"

            serializer = {
                generateStub = false
            }
            validator = {
                // composedCheck = "org.eclipse.xttext.validation.NamesAreUniqueValidator"
            }

            fragment = generator.GeneratorFragment2 {
                generateJavaMain = true
            }
        }
    }
    ...
```

- Dann die Grammatik neu erzeugen.
- Die neu erzeugte Klasse kurs.xttext.dataflow.generator.Main ausführen (Run As / Java Application). → Bricht mit einem Fehler (keine Eingabedateien) ab.
Optional: Man kann die Main Datei modifizieren, so dass sie nicht nur eine übergebene Datei „compiliert“, sondern auch viele Dateien (z.B. mit „*“ an der Konsole).
- Abschließend mit „Rechts-Klick im Projekt Browser / Export“ das Projekt exportieren:
 - Selektion „**Runnable JAR File**“, „Next“

- **Launch Configuration:** „Main“ (das ist der Grund, dass man die Main 1x ausführen musste),
- **Export destination:** „mycompiler.jar“,
- Selektion „**Package required libraries into generated JAR**“, „Finish“

Mit dem so erzeugten Compiler („mycompiler.jar“) und den Modelldateien kann man Modelle validieren und Code generieren:

```
pierre@solo2:~/demo$ ls
mycompiler.jar  test1.dataflow

pierre@solo2:~/demo$ java -jar mycompiler.jar test1.dataflow
Code generation finished.

pierre@solo2:~/demo$ tree
.
├── mycompiler.jar
├── src-gen
│   ├── Components1.txt
│   ├── Components2.txt
│   ├── Components3.txt
│   └── Components4.txt
└── test1.dataflow

1 directory, 6 files
```

Ist das Modell fehlerhaft oder erzeugt die Validierung eine Warnung (vgl. Abschnitt 5.1), wird kein Code generiert (siehe Main.java):

```
pierre@solo2:~/demo$ java -jar mycompiler.jar test1.dataflow
WARNING:Name should start with a capital (test1.dataflow line : 10 column : 12)

pierre@solo2:~/demo$
```

9.2 Templates

Templates sind Textschnipsel, die man über Abkürzungen in ein Modell einfügen kann.

Diese kann man am einfachsten in der Runtime-Workbench editieren (vgl. /11/):

- Menu: "Window/Preferences" → „DataFlowDsl“ (Unsere Grammatik) → „Templates“
- Hier kann man neue Templates anlegen: „New...“
 - Name: „NewComponent“ (Das ist das Kürzel, mit dem man den Code Schnipsel einfügen kann)
 - Context: „KComponent“
 - Description: „Add a new Component“
 - Pattern (Hier kann man das meiste mit CTRL-Space vervollständigen):

```
Component MyComponent {  
    port_in myinput {${message:CrossReference(KPort.message)}}  
    port_out myoutput { ${message:CrossReference(KPort.message)}}  
}
```

- „OK“

Probieren Sie das neue Template aus: Geben Sie in einem KPackage nun „NewC“+CTRL-Space ein (damit selektieren Sie das neue Pattern „NewComponent“).

Um die **Templates permanent zur Verfügung zu stellen**, muss man wie in /11/ beschrieben vorgehen:

- Im obigen Dialog die Templates exportieren (templates.xml).
- Einen Ordner „templates“ im „UI“-Projekt anlegen.
- Dort diesen neuen Ordner zu „bin.includes“ in den „build.properties“ hinzufügen: **(1)** Doppel-Klick auf „build.properties“ Datei. **(2)** Haken beim Ordner „templates“ im Bereich „Binary Build“ setzen.
- Die exportierte Datei „templates.xml“ in diesen Ordner „templates“ einfügen.
- In der Datei „templates.xml“ zu den einzelnen Templates ein Attribut „id="ID<NUM>“ hinzufügen.

Beispiel:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><templates><template id="ID1"  
autoinsert="true" context="kurs.xtext.dataflow.DataFlowDsl.KComponent" deleted="false"  
description="Add a new Component" enabled="true" name="NewComponent">Component MyComponent {  
    port_in myinput {${message:CrossReference(KPort.message)}}  
    port_out myoutput { ${message:CrossReference(KPort.message)}}  
}</template></templates>
```

9.3 QualifiedNameProvider

Wir haben bereits gesehen, wie mittels Referenzen auf andere Modellelemente anhand deren Namen (oder qualifizierenden Namen) verwiesen wird. Existiert kein Name oder anderes Attribut, welches als Referenz dient (oder soll der von Xtext erzeugt qualifizierende Name angepasst werden), dann kann man mit minimalem Aufwand definieren, wie dieser Name ermittelt wird.

Als Beispiel verwenden wir diesmal eine Sprache zur Definition von Variablen und deren Initialisierung, deren Grammatik wie folgt definiert ist:

grammar kurs.xtext.VarLang **with** org.eclipse.xtext.common.Terminals

generate varLang "http://www.xtext.kurs/VarLang"

Model:

```
/* Ein Modell voller Variablen */
variable+=VariableInitialization (';' variable+=VariableInitialization)*;
```

VariableInitialization:

```
/* Eine Variable besteht wird deklariert und danach einem Wert zugewiesen, der in einem Ausdruck definiert wird */
declaration = VariableDeclaration '=' value = Expression
;
```

VariableDeclaration:

```
/*Eine VariablenDeklaration besteht aus einem Typ und einem Namen*/
visibility=Type name = ID
;
```

enum Type:

```
/* Es sind nur UINT32,UINT16 oder UINT 8 als Typen erlaubt */
UINT32='UINT32' | UINT16='UINT16' | UINT8='UINT8'
;
```

Expression:

```
/* Ein Ausdruck kann aus einem einzelnen IntValue oder einer Refrenz auf eine initialisierte Variable bestehen.
* Kann ein Ausdruck die Summe aus solchen Elemente beschreiben. */
(value += IntValue | value += VarReference) ( '+' (value += IntValue | value += VarReference))*
;
```

IntValue:

```
/* IntValue enthält einen Ganzzahligen Wert */
value = INT
;
```

VarReference:

```
/* Eine VariablenReferenz enthält eine Referenz auf eine initialisierte Variable und übernimmt ihren
* qualifizierenden Namen als ihren eigenen namen.*/
name = [VariableInitialization]
;
```



Nach dem Erzeugen des Editors werden Eingaben wie

```
UINT16 v1 = 1 + 1 ;
UINT16 v2 = 1 + 2
```

bereits erfolgreich vom Parser erkannt. Versuchen wir jedoch stattdessen

```
UINT16 v1 = 1 + 1 ;
UINT16 v2 = 1 + v1
```

erhalten wir einen Fehler bei dem Verweis auf v1 in Zeile 2.

Die **Ursache für diesen Fehler** ist, dass die Referenz auf eine *VariableInitialization* in der Grammatik-Regel *VarReference* von Xtext nicht aufgelöst kann, weil *VariableInitialization* weder einen Namen, noch ein anderes Attribut hat, das als ein qualifizierender Name dienen könnte.

Lösung: Wir können jedoch durch Überschreiben des von Xtext verwendeten *QualifiedNameProvider* selbst bestimmen, wie der qualifizierende Name eines Modellelements aussehen soll.

Dazu legen wir eine neue Xtend-Klasse *MyCustomQualifiedNameProvider.xtend* an (Strg + N und dann nach „Xtend Class“ filtern), diese kann z.B. im selben Package wie die *.xtend Grammatik abgelegt werden (kurs.xtext).

Darin definieren wir, dass der qualifizierende Name einer *VariableInitialization* der Name der in der enthaltenen *VariableDeclaration* sein soll:

```
import org.eclipse.xtext.naming.DefaultDeclarativeQualifiedNameProvider
import kurs.xtext.varLang.VariableInitialization;
import org.eclipse.xtext.naming.QualifiedName

class MyCustomQualifiedNameProvider extends DefaultDeclarativeQualifiedNameProvider {
    def qualifiedName(VariableInitialization initialization) {
        return QualifiedName.create(initialization.declaration.name);
    }
}
```

anschließend müssen wir Xtext noch mitteilen, dass wir unseren eigenen *QualifiedNameProvider* verwenden wollen, indem wir folgendes in der *VarLangRuntimeModule.xtend* ergänzen:

```
...
override bindIQualifiedNameProvider() {
    /* weise Xtext an unseren angepassten QualifiedNameSpaceProvider zu nutzen */
    return MyCustomQualifiedNameProvider
}
...
```

Nachdem wir den Editor neu generiert haben, wird nun das vorherige Beispiel vom Parser erfolgreich erkannt:

```
UINT16 v1 = 1 + 1 ;
UINT16 v2 = 1 + v1
```

Wir können bei Bedarf auch das Standard Verhalten für die Erzeugung für qualifizierende Namen von Xtext überschreiben (override statt def):

```
override qualifiedName(VariableInitialization initialization) {
    return QualifiedName.create(initialization.declaration.name);
}
```

10 Fazit

Dieses Dokument demonstriert, wie man mit relativ wenig Aufwand ein Metamodell mittels Xtext spezifizieren kann, wie man Modelle validiert und aus den Modellen Code generiert.

Die so erzeugten Modelle und Metamodelle gliedern sich nahtlos in die EMF Welt ein. Dies erlaubt auf natürliche Weise eine Integration mit anderen Werkzeugen und Formaten (z.B. ReqIF Format aus DOORS (<https://eclipse.org/rmf/>, 2017/07), XMI UML Format aus Papyrus (<https://eclipse.org/papyrus/>, 2017/07), ... und andere hauseigene Modelle).

Besonders hervorzuheben ist die hohe Flexibilität textueller Modellrepräsentationen:

- Einfach zu editieren (notfalls mit einem Texteditor).
- Eingabe unvollständiger Modelle ist möglich.
- Diff / Suchen-Ersetzen ist mit normalen Werkzeugen möglich.
- Merge Aktivitäten sind einfach zu handeln (da die Modellrepräsentation auf Platte der gewohnten Eingabesyntax entspricht)

Wir hoffen es hat Spaß gemacht!