Markus Völter, Lars Corneliussen

DSL Entwicklung mit Eclipse Xtext

Carpe Diem

Nutze den Tag. Domänenspezifische Sprachen sind eines der vielen Mittel zur Industrialisierung von Software. In anderen Branchen ist die Automatisierung längst Alltag. Praxisnah zeigt dieser Artikel eine DSL für den Import von Mainframe-Exporten – mit Eclipse Xtext in Szene gesetzt.

Zum heutigen Zeitpunkt ist für die Entwicklung von textuellen DSLs kein ausgereiftes integriertes Werkzeug für Microsoft Visual Studio erhältlich. Die angekündigte Modellierungsplattform Microsoft „Oslo“ wird diese Lücke mittelfristig schließen. Doch warum bis dahin warten?

In der Einführung zu domänenspezifischen Sprachen „Kurze Rede, langer Sinn“ der vergangenen Ausgabe hat dotnetpro bereits auf die Open-Source-Lösung Xtext und openArchitectureWare hingewiesen. Xtext ist ein Antlr-basierter Parser- und Editorgenerator und bietet höchsten Komfort beim Entwickeln einer Grammatik und eines benutzerfreundlichen Editors für beliebige DSLs. openArchitectureWare ist ein praxiserprobtes Framework für Codegenerierung.

Dass Xtext und oAW ihren Dienst tun, hat sich in vielen erfolgreichen Referenzprojekten hinreichend gezeigt. Die aktuelle Herausforderung besteht darin, eine gute Einbindung in die Werkzeugwelt eines .NET-Entwicklungsprojektes zu gewährleisten.

In diesem Artikel möchten wir Ihnen zuerst ein Beispiel für eine DSL mit Xtext zeigen und anschließend darüber berichten, wie die Integration mit .NET Projekten aussehen kann.

Das Beispiel

Im Beispiel geht es darum, proprietäre Datenstrukturen, die von einem Mainframe exportiert wurden, in ein klassisch strukturiertes Datenformat zu transformieren. Das Beispiel ist an einen Anwendungsfall aus der Krankenhaus-EDV angelehnt. Die Daten beschreiben den Aufenthalt eines Patienten im Krankenhaus. Hier ist eine Beispieldatei:

```
P001 Voelter,Markus,*14.02.1974
A001 Grabenstrasse 4, 73033, Goeppingen
V001 Futuristic Insurance,123456767
T001 49,07161,12334454
T001 49,171,8601869
S001 01.02.2004,IN25
S002 03.02.2004,CH52
S003 05.02.2004
S001 22.03.2005,NE02
S002 25.03.2005,NE04
S002 27.03.2005,NE05
S003 28.03.2005
P001 Somebody,Else,*23.05.1960
A001 Ziegelaecker 11, 89520, Heidenheim
V001 Some Other Insurance GmbH,124345
T001 49,07321,973344
S001 23.02.1999,IN25
S003 02.03.1999
P001 Person,Else,*23.05.1960
A001 Ziegelaecker 11, 89520, Heidenheim
V001 Some Other Insurance GmbH,124345
T001 49,07321,973344
S001 23.02.1999,IN25
S003 02.03.1999
```

Listing 1

Jede Zeile enthält einen Datensatz, beginnend mit einem Kennschlüssel und gefolgt von kommaseparierten Werten. P001-Datensätze beschreiben die Stammdaten des Patienten. A001-Datensätze enthalten die Adresse, V001-Datensätze die Versicherungsinformationen und die T001-Datensätze die Telefonkontakte. In den S-Datensätzen werden die Aufenthalte des Patienten im Krankenhaus gespeichert, wobei S001-Datensätze die Aufnahme, S002-Datensätze die Umbettung und S003-Datensätze die Entlassung des Patienten beschreiben.

Um Datensätze dieser Struktur in eine Objektstruktur einzulesen, definieren wir eine domänenspezifische Sprache zur Patientenverwaltung. Programme in dieser Sprache sind zweigeteilt. Zuerst wird die Zieldatenstruktur definiert, darauf folgt die Abbildung der Quelldaten auf das definierte Zielformat.

Die beiden Abschnitte derselben Datei sind in Abbildung 1 zu sehen.

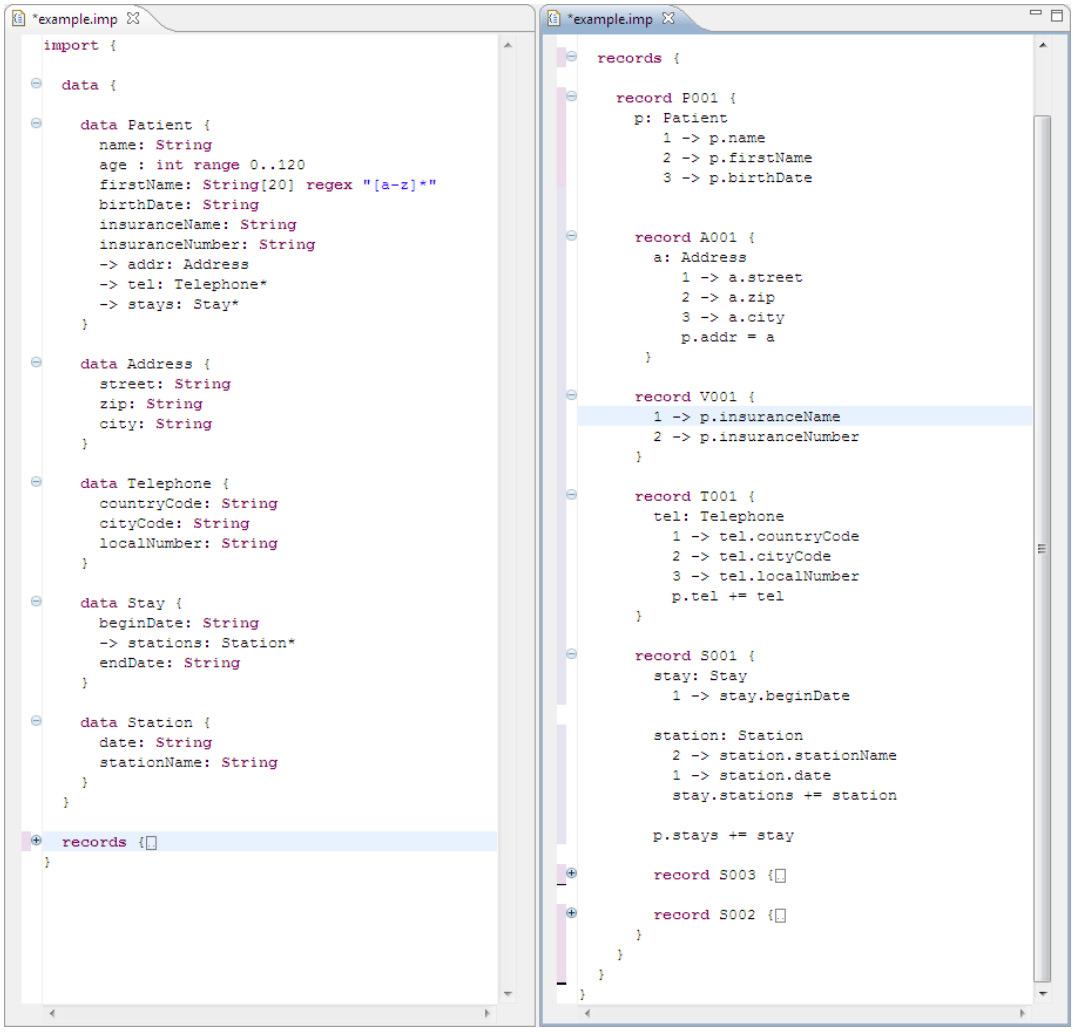

Abbildung 1

Die Zieldatenstruktur (*data*) enthält eine Liste der Entitäten. Diese haben Attribute, sowie Referenzen zu anderen Entitäten. Referenzen können mengenwertig sein, was an dem Stern hinter dem Typ erkennbar ist.

Die Importbeschreibung ist eine auf das Mainframe-Export-Format zugeschnittene Grammatik. Die Datensätze werden in ihrer gültigen Reihenfolge aufgelistet. Die Verschachtelung der Records definiert die in den Quelldaten nicht explizit erkennbare Hierarchie. Innerhalb der entsprechenden record-Abschnitte werden die extrahierten Daten auf die in der Datensatzdefinition definierten Entitäten abgebildet.

Dieses Programm wird mit Hilfe eines Codegenerators in ausführbaren C#-Code transformiert. Auch der generierte Code besteht aus zwei Teilen. Zum einen werden die Datenstrukturen als POCOs (Plain Old CLR Object) mit entsprechenden Properties abgebildet. Zum anderen wird aus der Importbeschreibung eine Klasse erzeugt, die Dateien einlesen und in POCOs umwandeln kann.

Natürlich könnten wir alternativ auch einen Interpreter bauen. Wir haben uns allerdings für einen Generator entschieden, da der generierte Parser potenziell große Datenmengen mit hoher Geschwindigkeit verarbeiten können soll. Ein weiterer Vorteil ist, dass die importierten Daten mit allem Komfort starker Typisierung in der Importumgebung verwendet werden können. Zum Beispiel ließen sie sich über einen O/R-Mapper leicht in eine Datenbank speichern.

Im Rest des Artikels werden wir uns anschauen, wie die DSL in Xtext definiert wird, und wie mit Hilfe von oAW und Xpand ein Codegenerator entsteht. Um das Beispiel überschaubar zu halten, werden wir uns auf den Aspekt der Datendefinition konzentrieren.

Definition der Sprache

Im Falle von Xtext beginnt die Definition einer textuellen DSL immer mit der Festlegung der Grammatik, deren Form mehr oder weniger der im ersten Artikel dieser Reihe erklärten EBNF entspricht. Grammatiken sind Beschreibungen, die die konkrete textuelle Syntax beschreiben und auf einen abstrakten Syntaxbaum (AST) abbilden. Der AST ist im Falle von Eclipse Xtext als EMF-Modell repräsentiert. Das dahinter stehende Metamodell wird automatisch aus der Grammatik abgeleitet. Hier ein Ausschnitt aus der Grammatikdefinition.

```
DataStructure:
  "data" name=ID "{"
    ( attributes += Attribute |
      references += Reference)*
  "}";

Reference:
  "->" name=ID ":" type=[DataStructure] (ismulti?="*")?;

Attribute:
  name=ID ":" type=AttributeType;

AttributeType:
  IntType | BooleanType | StringType;

IntType:
  "int" (range=Range)?;

Range:
  "range" min=INT ".." max=INT;

BooleanType:
  "boolean";

StringType:
  "String" ("["len=INT"]")? ("regex" regex=STRING)?;
```

Listing 2

Grammatiken bestehen aus benannten Regeln gefolgt von Produktionen. Jede Regel wird zu einer Metaklasse im Metamodell der abstrakten Syntax. Properties in der Grammatik (wie beispielsweise *name*) werden auf Attribute einer Metaklasse abgebildet. Referenzen auf weitere Klassen werden mit „->“ angeführt. Der AST enthält dann statt eines primitiven Wertes eine Referenz. Einfache Zeichenketten in der Grammatik werden mit dem Typ ID (Identifier) beschrieben.

Wenn eine Regel eine andere benutzt, wird dies ebenfalls als Zuweisung geschrieben. Beispielsweise ruft die Regel *IntType* die Regel *Range* auf. Das Ergebnis der Produktion von *Range* wird in *IntType.range* abgelegt. Wichtig zu verstehen ist, dass Grammatiken in einen Baum von Objekten resultieren. Mit anderen Worten, eine Instanz von *IntType* enthält eine Instanz von *Range* im *range*-Attribut.

Zusätzlich unterstützt Xtext Querreferenzen sowohl innerhalb eines Baumes als auch dateiübergreifend. Ein Beispiel ist das *type* Attribut der *Reference* Regel. Der in die eckigen Klammern eingefügte Regelname deutet darauf hin, dass hier die ID (das Attribut *name*) einer woanders deklarierten Instanz von *DataStructure* erwartet wird. Im AST ergibt dies eine echte Objektreferenz.

Abbildung 2 zeigt das zu dem Grammatikausschnitt passende Metamodell. Hier ist klar ersichtlich, dass die Grammatik und das Metamodell strukturell identisch sind.

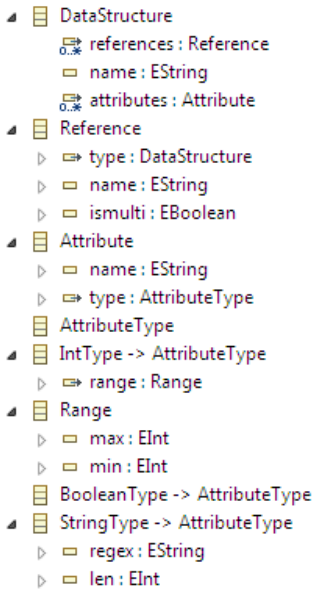

Abbildung 2

Eine einfache, dem Grammatikausschnitt genügende, Datenstruktur könnte wie folgt aussehen:

```
data Person {
  firstName: String
  lastName: String[30]
}
```

Der dazu passende AST (Abstract Syntax Tree), zur vereinfachten Darstellung in XML serialisiert:

```
<DataStructure name=“Person“>
  <attributes>
    <Attribute name=“firstName“/>
    <Attribute name=“lastName“ len=“30“/>
  </attributes>
</DataStructure>
```

Wenn die Grammatik definiert ist, kann durch Ausführung eines Generierungsschrittes der Editor erstellt werden. In diesem Schritt entsteht auch das oben gezeigte Metamodell, und die zum Parsen verwendete antlr-Grammatik (Antlr ist ein beliebter Parsergenerator für Java). Der Editor unterstützt automatisch Syntax-Highlighting, Code Completion, Code Folding sowie Goto Definition und Find References. So bekommt man ohne weiteres Zutun ein Metamodell und einen benutzerfreundlichen Editor einzig aus der Definition einer Grammatik.

Definition von Constraints

Bei Xtext wird das Metamodell aus der Grammatikdefinition generiert und enthält deshalb ausschließlich syntaktische Regeln. Trotzdem sind zusätzliche Constraints zur Überprüfung der Semantik notwendig. Diese lassen sich bei Xtext mittels der Sprache Check beschreiben. Check ist eine pragmatische Umsetzung der OCL (Object Constraint Language, OMG).

Eine Constraint beginnt mit dem Schlüsselwort *context* gefolgt vom Namen der Metaklasse (Regel in der Grammatik) , für deren Instanzen die Constraint gilt. Das Schlüsselwort *ERROR* oder WARNING klassifiziert die folgende Textmeldung. Diese Textmeldung wird ausgegeben, wenn die auf den Doppelpunkt folgende boolesche Constraint-Expression auf *false* evaluiert.

Die folgende Constraint überprüft für alle im DSL-Programm definierten Datenstrukturen, ob der Name mit einem Großbuchstaben beginnt.

```
context DataStructure ERROR "data structure names must begin with an uppercase letter":
          name.toFirstUpper() == name;
```

Eine weitere Constraint ist etwas komplizierter. Sie überprüft, ob in einer Datenstruktur Attribute mit gleichen Namen mehrfach vorkommen. Ausgehend von dem Attribut wird zur besitzenden Datenstruktur navigiert. Von dieser holen wir uns alle Attribute, und filtern uns die heraus, deren Name dem Namen des gerade überprüften Attributes entspricht. Wir erwarten, dass wir genau ein solches Attribut finden, nämlich das, für dass wir die Constraint gerade überprüfen. Finden wir mehr als ein Vorkommen, haben wir einen doppelten Namen erkannt.

```
context Attribute ERROR "duplicate attribute name":
          ((DataStructure)eContainer).attributes.select(a|a.name == this.name).size == 1;
```

Mit der Sprache Check lassen sich sehr prägnant auch nicht triviale Constraints beschreiben. Nach einer kurzen Eingewöhnungszeit will man die Sprachmächtigkeit nicht mehr missen.

Die Constraints werden im Editor überprüft, sobald man das gerade in Bearbeitung befindliche Modell speichert. Fehler werden in der Eclipse Fehlerliste angezeigt und auch mit einem kleinen roten Kreuz links am Editor markiert (s. Abbildung 3). Im Rahmen der Verarbeitung durch einen Codegenerator wird diese Überprüfung ebenfalls durchgeführt. Dazu später mehr.

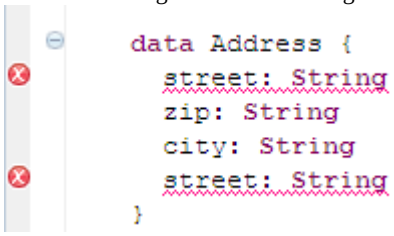

Abbildung 3

Extensions

Die Modellverarbeitung enthält in den meisten Fällen einen großen Anteil an Navigation über das Modell beziehungsweise Filterung von Elementen. Um eine gute Wartbarkeit sicherzustellen, gibt es die Möglichkeit, Funktionen zu extrahieren, die komplexere Ausdrücke kapseln. Diese Funktionen, auch „Extensions“, werden mit der Sprache Xtend implementiert. Diese können beispielsweise im Rahmen eines Constraintchecks aufgerufen werden, oder auch im Codegenerator Verwendung finden. An dieser Stelle möchte ich zwei Beispiele geben.

Die folgende, sehr einfache, Extension liefert für eine Datenstruktur alle von ihr referenzierten Datenstrukturen zurück. Dabei wird über alle Referenzen der aktuellen Datenstruktur iteriert und von jeder einzelnen der Typ zurückgegeben. Eine Extension besteht aus dem Namen, einer Reihe von Parametern, und rechts vom Doppelpunkt der Expression, die die Extension kapseln soll.

Üblicherweise benennt man das erste Argument this, denn Extensions werden üblicherweise in Punktnotation aufgerufen (also *someDataStructure.allReferencedDataStructures()*).

```
allReferencedDataStructures( DataStructure this ): references.type;
```

Man beachte den Ausdruck rechts des Doppelpunktes: man muss keine expliziten Schleifen programmieren, sondern kann auf der Collection direkt das gewünschte Attribut aufrufen. Das Äquivalent mit C# 3 sieht wie folgt aus:

```
reference.Select(r=>r.type)
```

Der Rückgabetyp einer Extension muss nicht explizit angegeben werden. Er wird automatisch mittels Type Inference berechnet. Die folgende Extension mit expliziter typisierung ist äquivalent.

```
Collection[DataStructure] allReferencedDataStructures(
DataStructure this ): references.type;
```

Hier ein weiteres Beispiel für Extensions. Die folgende liefert den *RecordHandler* zurück, indem sich der aktuelle *RecordHandler* befindet. Die Expression besteht aus zwei geschachtelten bedingten Zuweisungen (?:-Operator). Die erste überprüft, ob das aktuelle Element überhaupt einen Besitzer hat. Wenn ja, dann wird mit der zweiten überprüft, ob es sich dabei um eine Instanz von *RecordHandler* handelt. Falls dies der Fall ist, wird dieser gecastet und zurückgeliefert. In allen anderen Fällen wird *null* zurückgegeben.

```
parentHandler( RecordHandler this ):
        eContainer != null
                ? ( RecordHandler.isInstance(eContainer) ? ((RecordHandler)eContainer) : null )
                : null;
```

Anpassungen des Editors

Der aus der Grammatik abgeleitete Editor ist prinzipiell voll funktionsfähig. Trotzdem will man in vielen Fällen Anpassungen vornehmen, um die Benutzerfreundlichkeit des Editors zu steigern. Alle Editoranpassungen geschehen durch Implementierung beziehungsweise Überschreibung bestimmter, vordefinierter Extensions. Hier einige Beispiele.

Um im Outline-View des Editors das Icon und das Textlabel anzupassen, müssen die Extensions *label* und *image* für die betreffenden Typen überschrieben werden. Im Falle von Label kann mittels der Expression Sprache ein beliebiger String zusammengebaut werden. Im Falle von image wird einfach der Name einer Bilddatei zurückgegeben (die betreffenden Bilder müssen im *icons* Verzeichnis des Editorprojekts zu finden sein).

```
label( Instance this ): name+": "+type.name;
label( Reference this ): "-> "+name+": "+type.name;

image( Instance this ): "instance.gif";
image( RecordSection this ): "region.gif";
image( RecordHandler this ): "recordHandler.gif";
```

Abbildung 4 zeigt das Ergebnis der Anpassungen.

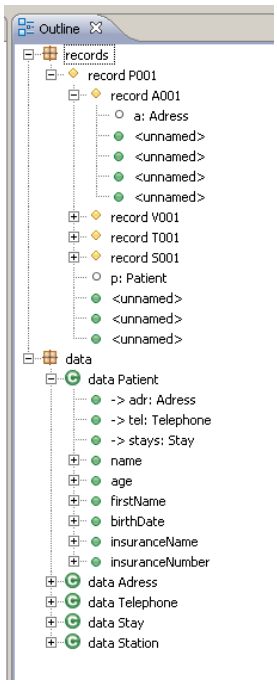

Abbildung 4

Codegenerierung

Codegenerierung geschieht in unserem Beispiel mittels openArchitectureWare Xpand. Xpand ist eine Templatesprache, eine DSL zur Generierung von Textdateien aus Modellen. Mittels der Sprache kann also sehr effizient über das Modell navigiert, und dann als Seiteneffekt Textdateien ausgegeben werden. Abbildung 5 zeigt ein einfaches Beispiel. Dort wird ein Template namens *properties* definiert. Dieses Template ist der Metaklasse *DataStructure* zugeordnet. Man kann sich das analog zu Methoden in der objektorientierten Programmierung vorstellen. Eine Methode ist einer Klasse zugeordnet. Das bedeutet, dass es innerhalb der Methode eine Variable namens *this* gibt, die auf die aktuelle Instanz der Klasse zeigt, in der die Methode definiert ist. Gleiches gilt für Templates. Innerhalb eines Templates existiert eine Variable *this*, die auf die aktuelle Instanz der Metaklasse zeigt, für die das Template definiert ist. Mittels des FOREACH-Konstruktes kann dann beispielsweise über eine Collection iteriert werden. In vorliegenden Fall iterieren wir über die Referenzen der Datenstruktur. Für jede der Referenzen generieren wir Code. Dieser enthält eine private generische Liste deren Itemtyp über die Extension *fqClassname()* aufgelöst wird. Der Name der Referenz geht ebenfalls in den Feldnamen ein Die französischen Anführungszeichen werden verwendet, um zwischen dem zu generierenden Text und dem Templatecode hin und her zu schalten. Innerhalb französischer Anführungszeichen können beliebige Expressions stehen oder auf die oben bereits erwähnten Extensions zugegriffen werden.

```
«DEFINE properties FOR DataStructure-»
    «FOREACH references AS r-»
        «IF r.ismulti-»
        private readonly List<«r.type.fqClassname()»>
          «r.name»List = new List<«r.type.fqClassname()»>();
        «ENDIF-»
    «ENDFOREACH-»

    «FOREACH attributes AS a-»
        public «a.type.clrType()» «a.name.toFirstUpper()» { get; set; }
    «ENDFOREACH-»

    «FOREACH references AS r-»
        «IF r.ismulti»
        public IList<«r.type.fqClassname()»> «r.name.toFirstUpper()»
        {
            get
            {
                return «r.name»List;
            }
        }
        «ELSE-»
        public «r.type.fqClassname()» «r.name.toFirstUpper()» { get; set; }
        «ENDIF-»
    «ENDFOREACH-»
«ENDDEFINE»
```

Abbildung 5

Für die in *Patient* referenzierte Klasse Telephone wird zum Beispiel folgende Zeile generiert:

```
private readonly List<Telephone> telList = new List<Telephone>();
```

Der öffentliche Zugriff erfolgt über ein entprechendes Property:

```
public IList<Telephone> Tel
{
  get
  {
    return telList;
  }
}
```

Abbildung 5 zeigt auch weitere Features der Templatesprache. Wie zu erwarten war, lässt sich mittels IF eine boolesche Bedingung abfragen. Das Minus vor den schließenden französischen Anführungszeichen unterdrückt die folgenden Leerzeichen und Umbrüche. Für die Generierung von Java-Quellcode werden sogenannte „beautifier“ eingesetzt, um den generierten Quelltext „hübsch“ zu machen. Für C# ist noch kein „beautifier“ verfügbar. Deshalb muss das Template die Ausgabe jedes Zeichens exakt kontrollieren.

XPand erinnert sehr an XSL. Es lässt sich auch ähnlich einsetzen. Im Gegensatz zu XSLT (in erster Version) bietet es jedoch einen typsicheren Durchgriff auf den abstrakten Baum (AST) der DSL (inkl Intellisense, etc.) sowie eine einfachere Einbindung von dynamischen Extensions. Außerdem unterstützt XPand weitere Konstrukte, die auf das Generieren von Textdateien ausgelegt sind. So können mittels FILE mehrere Dateien in einem Template ausgegeben werden, oder mit AROUND Aspekte wie zum Beispiel Caching oder Logging nachträglich in separaten Templates ergänzt werden.

Workflows

Wir haben nun eine Sprache definiert, einen Editor gebaut, sowie Code Generierungstemplates definiert. Wir wollen nun den Code Generator ausführen. Dazu fehlt allerdings noch ein Baustein, der Workflow. Der Workflow dient dazu, Parser,

Constraintchecks, und Codegeneratoren auszuführen. Workflows können entweder innerhalb Eclipse ausgeführt werden oder auch mit einer Batch-Datei aufgerufen werden. Damit lassen sie sich in beliebige Entwicklungsumgebungen einbetten.

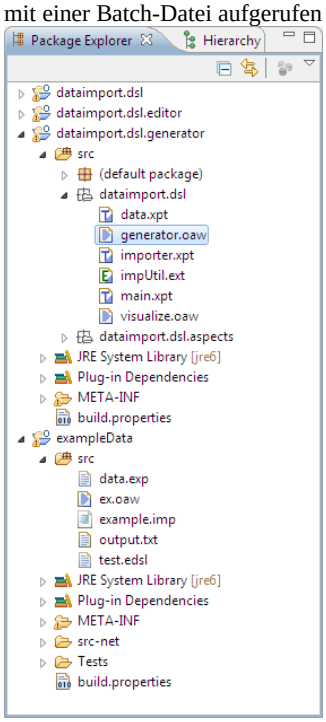

Abbildung 3

Um die Struktur der Workflows zu verstehen, sollten wir zunächst einen Blick auf die generelle Projektstruktur werfen. Abbildung sechs zeigt diese. Im *dataimport.dsl*-Projekt findet sich die Sprachdefinition, also die Grammatik, die Constraints, das abgeleitete Metamodell und der Parser. Im Projekt *dataimport.dsl.editor* liegen alle die generierten Editorartefakte, sowie die von uns implementierten Editoranpassungen. Das Editorprojekt hat eine Abhängigkeit zum Sprachprojekt. Im *dataimport.dsl.generator* Projekt finden sich die Codegenerierungstemplates sowie ein Workflow, der die eigentliche Generierung durchführt (*generator.oaw*). Der Editor und der Generator gelten für alle Programme, die mit der Import-DSL geschrieben werden.

Das Beispielprogramm selbst und die Importbeschreibung für die Krankenhaus-Daten liegen im Projekt *exampeProject*. Der Workflow *ex.oaw* ruft den allgemeinen Codegenerator auf und übergibt das DSL-Programm *example.imp*.

Werfen wir nun einen Blick auf die Workflows. Ein Workflow ist ein XML File mit Properties und Komponenten. Eine Workflow Komponente repräsentiert eine Java Klasse, die eine gewisse Funktionalität ausführt, oder eine weitere Workflowdatei, die aufgerufen wird. Beide sind parametrisierbar.

Der Workflow ist eine Kette von Ausführschritten. Die Datei *generator.oaw ist* für die generelle Codegenerierung zuständig und erstellt zuerst den Parser für die DSL und generiert daraufhin den Importer für die im Parameter *modelFile* mitgegebene Importbeschreibung.

```
<workflow abstract="true">

  <property name='modelFile' />
  <property name='targetDir' value='src-gen/'/>

  <component file='dataimport/dsl/parser/Parser.oaw'>
     <modelFile value='${modelFile}'/>
     <outputSlot value='theModel'/>
  </component>

  <component class='oaw.workflow.common.DirectoryCleaner' directories='${targetDir}'/>

  <component class='oaw.xpand2.Generator'>
     <metaModel id='mm' class='org.eclipse.m2t.type.emf.EmfRegistryMetaModel'/>
     <expand value='dataimport::dsl::Main::main FOR theModel'/>
     <genPath value='${targetDir}'/>
  </component>

</workflow>
```

Um das Bild zu vervollständigen, sollten wir uns noch kurz den Workflow des *exampleData* Projektes anschauen.

```
<workflow>

    <bean class="org.eclipse.mwe.emf.StandaloneSetup">
         <platformUri value=".."/>
    </bean>
```

```
    <cartridge file="dataimport::dsl::generator.oaw"
         modelFile="example.imp"/>

</workflow>
```

Dieser ist extrem einfach. Er initialisiert das Generatorframework, und ruft den generellen Workflow auf. Als Parameter gibt er dabei den Namen des zu verarbeitenden Models an. openArchitectureWare verwendete den Java Classpath um die Ressource zu lokalisieren.

Integration in die .NET-Welt

Eine zusätzliche Herausforderung besteht darin die generierten Artefakte in den Entwicklungsprozess einer .NET-Lösung einzubinden. Dazu gibt es verschiedene Ansätze. Auch wenn Eclipse mit Hilfe des Plugins emonic (Eclipse Mono Integration) einen passablen Editor für C# bietet, ist es nicht geeignet Visual Studio vollständig zu ersetzen. Trotzdem hilft das Plugin, wenn man nicht laufend beide Umgebungen geöffnet halten möchte.

Da der benutzerfreundliche Editor für die eigene DSL nur unter Eclipse läuft, bietet es sich an, Eclipse zur Bearbeitung der Codebasis, die diese DSL benutzt, zu verwenden. Der Generator kann den Quellcode entweder direkt in ein Verzeichnis mit einem vorbereiteten .NET-Projekt ausgeben oder im gleichen Schritt direkt eine Assembly (DLL) erzeugen. Bei einer Mischform von generiertem und nicht generiertem Code innerhalb derselben Assembly muss die *.csproj*-Datei für wegfallende und hinzugenommene Dateien entweder manuell oder automatisiert angepasst werden.

Bei einem isolierten Build direkt in eine Assembly empfiehlt sich ein Buildskript, zum Beispiel mit MSBuild oder NAnt, das ohne eine Projektdatei auskommt.

Im Beipielprojekt auf der Heft-CD sind sowohl das Eclipse-Projekt, als auch eine entsprechende Visual-Studio-Solution (Abbildung 4 und 5) zu finden.

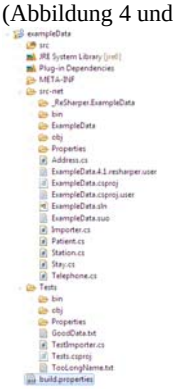

Abbildung 4

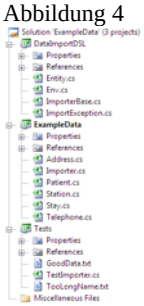

Abbildung 5

Tipp: Bei Verwendung von Codegeneratoren fragt Visual Studio häufig, ob die geänderten Dateien neu geladen werden sollen. Mit der Option [Extras > Options > Environment > Documents > Detect when file is changed outside the environment > Auto-load changes, if saved] werden die Dateien ohne lästige Abfragen aktualisiert.

Fazit

Die Rechnung ist einfach. Wenn eine Datei im Format des Beispiels in die Datenbank importiert werden soll, wird es manuell wohl am schnellsten gehen. Sind es 500 Dateien in diesem Format, ist ein manuell entwickelter Parser sicherlich der bessere Weg. Werden es aber je ein paar tausend Dateien in 20 Formaten, geht die Rechnung, eine DSL dafür zu erstellen, auf.

Die Technologiewahl ist zweitrangig. Und doch muss eine Wahl getroffen werden. Eclipse Xtext ist ein praxiserprobtes Werkzeug, das sowohl bei der Umsetzung einer DSL als auch bei der Codegenerierung sehr viel Komfort bietet.