Parsen und Verarbeiten Textueller Spezifikationen

Markus Völter, voelter@acm.org, www.voelter.de

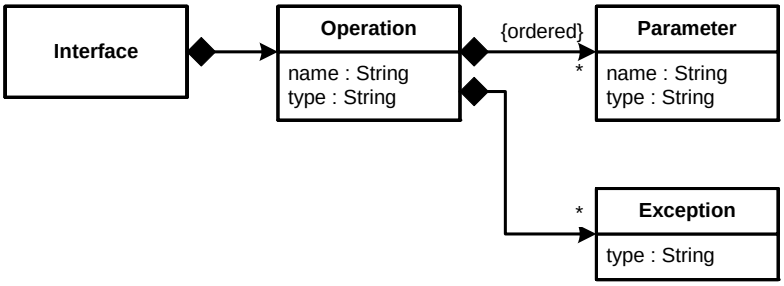
Im Zusammenhang mit Modellgetriebener Softwareentwicklung ist das Parsen von Modellen und deren Weiterverarbeitung (insbesondere zwecks Codegenerierung) sehr wichtig. Oft verwendet man grafische, UML basierte Modelle als Spezifikation. In vielen Fällen sind textuelle Spezifikationen aber mindestens genauso nützlich. Auch wenn man nicht komplett modellgetrieben arbeitet, macht es oft Sinn, sich für bestimmte Aspekte kleine, textuelle Sprachen zu bauen; man denke nur an CORBA IDL. Ich möchte in diesem Artikel zeigen, wie man mit Parsergeneratoren – hier am Beispiel von JavaCC – arbeitet. Aus eigener Erfahrung weiß ich, dass diese Art von Tools für viele Leute schwer zugänglich ist, für allem für nicht-Informatiker (bin selbst Ingenieur). Zum Zwecke der Illustration werden wir dann aus dem Modell Code generieren. Dazu verwenden wir den openArchitectureWare [oAW] Codegenerator, der bereits im Rahmen eines früheren Artikels in dieser Kolumne eingeführt wurde [oAWIntro].

Die Spezifikation

Als Beispiel einer textuellen Spezifikation betrachten wir Interfacedefinitionen, wie sie z.B. von CORBA IDL verwendet werden.

```
interface Sensor {
  operation start():void;
  operation stop():void;
  operation measure():float;
}
```

Neben dieser konkreten, textuellen Syntax besitzt jede Sprache auch eine abstrakte Syntax (auch als Metamodell bezeichnet). Hier also das Metamodell für Interfaces. Interfaces sind definiert (wie üblich) als eine Menge von Operationen, die Parameter und Exceptions haben (können):



Parsen der textuellen Syntax

Zum Parsen der textuellen Syntax wird ein Parsergenerator verwendet. Parsergeneratoren arbeiten prinzipiell folgendermaßen:

- Der Entwickler spezifiziert die Grammatik der zu parsenden Sprache in einer für den Parsergeneratoren proprietären Syntax.
- Ein Tool (der Parser*generator*) liest diese Datei und erstellt ein Programm, welches Sätze in den im Grammatikfile beschriebenen Sprache parsen kann.
- Die Ausführung des generierten Parsers mit einem Satz der Grammatik als Eingabedaten kann nun beispielsweise einen Abstrakten Syntaxbaum erstellen. Dieser repräsentiert die abstrakte Syntax des Satzes als Objektgraph in der Programmiersprache des generierten Parsers.

Wir verwenden hier den Parsergenerator JavaCC [JavaCC], wobei wir nicht auf alle Details eingehen – ggfs. sollte die Dokumentation von JavaCC konsultiert werden. Statt JavaCC könnte auch ein anderer Parsergenerator verwendet werden. Das generelle Vorgehen ist mit unter Verwendung verschiedener Parsergeneratoren identisch – die Details, vor allem die der Benutzung des Tools, variieren selbstverständlich. JavaCC generiert den Parser in Java. Dies hat zur Folge, dass der AST als Java-Objekte repräsentiert wird. Genau dies erlaubt auch die Integration mit dem openArchitectureWare-Generator: auch dieser repräsentiert Modelle als Java-Objektgraphen. Dazu später mehr.

Wir werden zunächst auf das generelle Vorgehen eingehen, bevor wir dann noch das eine oder andere Detail zu JavaCC erläutern.

Generelles Vorgehen beim Parsen und Integration in den Codegenerator

Um die verschiedenen Elemente der abstrakten Syntax im AST zu repräsentieren, werden verschiedene AST-Klassen verwendet. Diese entsprechen den Metaklassen im Kontext des openArchitectureWare Generators. Da dieser erwartet, dass alle Metaklassen von der Klasse *ModelElement* erben, müssen wir dafür sorgen, dass die von JavaCC verwendete

Basisklasse der AST-Knoten ihrerseits von *ModelElement* erbt. Damit sind dann alle AST-Klassen Modellelemenente im Sinne des Generators.

Wir modifizieren also die Klasse *SimpleNode* (die standardmäßig bei JavaCC mitgeliefert wird und die Basisklasse für alle AST-Knoten darstellt) folgendermaßen:

```
public class SimpleNode
   extends ModelElement implements Node {

  public SimpleNode(int i) {
    id = i;
    JCCHelper.getMetaEnvironment().
              addElement( this );
    setName("");
  }

  // rest as before…

}
```

Zum einen erweitern wir die vom Generator stammende Klasse *ModelElement*. Zum anderen passiert noch eine wichtige Sache im Konstruktor: Es wird das aktuelle Element zum *MetaEnvironment* des Generators dazugefügt (woher das kommt sehen wir noch). Ohne diese Zeile wird das ganze nicht funktionieren, weil der Generator das Objekt nicht „kennt".

Nun müssen wir JavaCC die konkrete Syntax beibringen. Dazu definieren wir ein Syntaxdefinitionsfile, *InterfaceParser.jjt*, aus welchem der tatsächliche Parser generiert wird. Wir beginnen mit der Definition der Parserklasse, die JavaCC generieren soll.

```
PARSER_BEGIN(InterfaceParser)
  package util;
  public class InterfaceParser {
    public static void main(String args[]) {
      InterfaceParser t =
        new InterfaceParser (System.in);
      try {
        ASTStart n = t.Start();
        n.dump("");
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
PARSER_END(InterfaceParser)
```

Die *main*-Methode ist für das Funktionieren des Parsers nicht wirklich nötig, sie dient zu Testzwecken; sie liest den Eingabetext von stdin und erstellt den AST. Dieser wird dann ausgegeben. Der eigentliche Parser wird nun mittels von JavaCC bereit gestellter Tools

generiert. Das Syntaxdefinitionsfile benötigt noch eine ganze Reihe weiterer Angaben um die Syntax parsen zu können. Wir gehen darauf im nächsten Abschnitt ein.

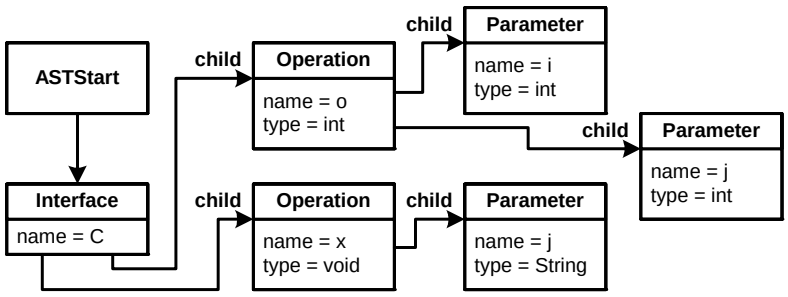
Zu Testzwecken lassen wir das folgende Interface parsen:

```
interface C {
  operation o(i:int,j:int):int;
  operation x(j:String):void;
}
```

Die Ausgabe die von der *main*-Methode erstellt wird sieht dann folgendermaßen aus:

```
this: Start
   child: Class(C)
this: Class(C)
   child: Operation(o):int
this: Class(C)
   child: Operation(x):void
this: Operation(o):int
   child: Parameter(i:int)
this: Operation(o):int
   child: Parameter(j:int)
this: Operation(x):void
   child: Parameter(j:String)
```

Zum besseren Verständnis, das ganze grafisch:



Angenommen, wir haben den Parser (*InterfaceParser.java*) generiert, so müssen wir nun als nächstes ein entsprechendes Generatorfrontend implementieren. Generatorfrontends dienen dazu, die konkrete Syntax von Modellen zu lesen und eine Menge von *ModelElements* zu liefern, die dann im Generator zu Laufzeit vorliegen und als Basis für die Generierung dienen (standardmäßig wird ein Frontend verwendet welches XMI parst um UML Modelle zu „verstehen“). Frontends müssen das *InstantiatorInterface* implementieren. Die Implementierung des Frontends delegiert das Parsen an sich an den von JavaCC generierten Parser. Das Frontend sieht damit für unseren Fall folgendermaßen aus:

```
public class JCCInstantiator
   implements InstantiatorInterface {
```

```
  public ElementSet loadDesign(
              InstantiatorEnvironment env)
              throws ConfigurationException,
              InstantiatorException {
     JCCHelper.setMetaEnvironment(
               env.getMetaEnvironment() );
      ElementSet result = new ElementSet();
       String spec = // read specification from file, etc…
      StringBufferInputStream s =
               new StringBufferInputStream( spec );
     InterfaceParser p = new InterfaceParser(s);
     ASTStart start = p.Start();
     ASTInterface cls =
                (ASTInterface)start.Child().get(0);
     result.add( cls );
      return result;
  }
}
```

Hier sind zwei Dinge zu beachten: Zum einen wird in der ersten Zeile der Operation das aktuelle *MetaEnvironment* (welches uns hier vom Generator zur Verfügung gestellt wird) in den *JCCHelper* gespeichert – wie oben gezeigt, holen die AST-Klassen es dort in ihrem Konstruktor wieder heraus. Dann wird die *ASTInterface* Instanz herausgeholt und als einziges Element in das *result* dieser Operation geschrieben. Das Frontend liefert also als Toplevel-Element des Modells die Interface-Deklaration zurück.

Damit ist die Integration in den Generator fast fertig. Wir müssen nun lediglich noch dem Generator mitteilen, dass er dieses Frontend verwenden soll, statt des Standard-XMI-Frontends. Dazu muss im Generator folgendes System-Property gesetzt werden:

```
System.setProperty("de.bmiag.genfw.instantiator.class",
                   "util.JCCInstantiator");
```

Syntaxdefinition in JavaCC

Das oben erwähnte Syntaxdefinitionsfile enthält bis jetzt nur die Deklaration der Parserklasse; die Definition der Syntax an sich haben wir noch nicht vorgenommen. Dies holen wir jetzt nach.

Es muss zunächst definiert werden, welche Tokens vom Parser generell ignoriert werden sollen. Dies umfasst inbesondere Whitespace sowie Kommentare.

```
SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| <"//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
| <"/*" (~["*"])* "*" (~["/"] (~["*"])* "*")* "/">
```

```
}
```

Im nächsten Schritt werden nun verschiedene Literale („reservierte Wörter“) definiert; also Zeichenfolgen die der Parser als zusammenhängend erkennen soll.

```
TOKEN : /* LITERALS */ {
  <INTERFACE_LITERAL: "interface"> |
  <OPERATION_LITERAL: "operation"> |
  < INTEGER_LITERAL: <DECIMAL_LITERAL>
    (["l","L"])? > |
  < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
}
```

Dasselbe sollten wir nun noch für Identifier tun:

```
TOKEN : /* IDENTIFIERS */ {
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* > |
  < #LETTER: ["_","a"-"z","A"-"Z"] > |
  < #DIGIT: ["0"-"9"] >
}
```

Im folgenden Schritt definieren wir nun die zu parsende konkrete Syntax. Begonnen wird mit einer Toplevel-Deklaration namens *Start()*. Diese startet den eigentlichen Parsevorgang in dem die Produktion *Interface()* aufgerufen wird.

```
ASTStart Start() : {} {
  Interface()
  { return jjtThis; }
}
```

Dort, in *Interface()*, beginnt der eigentlich interessante Teil. Im Body (dem zweiten {}-Paar) versucht der Parser zunächst das <*INTERFACE_LITERAL*> zu finden, welches oben als „interface“ definiert wurde. Dann wird ein Identifier erwartet (der Name des Interfaces); diesen weißen wir dem zuvor definierten Token *t* zu. Es wird dann eine öffnende { erwartet, dann der *Body()*, sowie eine schließende }.

Wichtig ist die Deklaration #*Interface*. Dies bedeutet, dass, wenn diese Produktion erfolgreich geparst werden kann, eine Instanz der Klasse *ASTInterface* im AST instanziiert wird. Damit wird das geparste Interface mit einem Objekt im AST repräsentiert.

```
void Interface() #Interface : {
  Token t;
}
{
  <INTERFACE_LITERAL> t=<IDENTIFIER>
      "{" Body() "}" ";"
  { jjtThis.setName(t.image); }
}
```

Interessant ist noch *jjtThis.setName(t.image);* Damit wird auf der aktuellen AST-Objekt-Instanz die Methode *setName*() aufgerufen, wobei der Text des Tokens t (also der Name der Interfaces) als Parameter mitgegeben wird.

Die oben aufgerufene Produktion *Body()* muss natürlich definiert werden. Dies geschieht folgendermaßen:

```
void Body() #void : {}
{
  Operation()*
}
```

Dies sagt aus, dass kein Objekt im AST für den Body angelegt wird (#*void*); und dass der Body aus einer beliebigen Menge von *Operationen()* besteht. Konsequenterweise muss nun noch die *Operations*-Produktion definiert werden:

```
void Operation() #Operation : {
  Token name;
  Token type;
}
{
  ( <OPERATION_LITERAL> name=<IDENTIFIER>
       "(" (ParameterList())? ")" ":"
       type=<IDENTIFIER> ";" )
  {
    jjtThis.setName(name.image);
    jjtThis.setType(type.image);
  }
}

void ParameterList() #void: {}
{
  Parameter() ( "," ParameterList() )*
}

void Parameter() #Parameter : {
  Token name;
  Token type;
}
{
  name=<IDENTIFIER> ":" type=<IDENTIFIER>
  {
    jjtThis.setName(name.image);
    jjtThis.setType(type.image);
  }
}
```

Der Parser generiert alle Klassen für den AST selbst. *ASTInterface, ASTOperation,* sowie *ASTParameter* werden also automatisch generiert. Allerdings funktioniert dies nur korrekt, wenn keine spezifischen Operationen aufgerufen werden. Dies ist hier aber nicht der Fall. Im Falle von *ASTInterface* wird *setName()* aufgerufen, im Rahmen von *ASTParameter setName()* und *setType()*. Diese Klassen müssen also manuell implementiert werden; *ASTOperation* könnte ungefähr folgendermaßen aussehen. Wir erben von *SimpleNode*, die

Klasse die ihrerseits (siehe oben) von *ModelElement* erbt. Dies sorgt dafür, dass wir den geparsten AST im Generator verwenden können.

```
public class ASTOperation extends SimpleNode {
    private String type;
    private String name;

    public ASTOperation(int id) {
        super(id);
    }

    public void setType(String type) {
        this.type = type;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

Damit wäre die Definition der Syntax beendet. Wie bereits erwähnt, muss noch ein von JavaCC zur Verfügung gestellter Generatore angeworfen werden um den eigentlichen Parser zu generieren. Letztendlich wird dann eine Klasse *InterfaceParser* generiert, die im oben gezeigten Frontend Verwendung findet.

Codegenerierung

Beim openArchitectureWare-Generator ist die Codegenerierung– wie bereits mehrmals erwähnt – unabhängig von der konkreten Syntax des Eingangsmodells. Außerdem kann man bei der Codegenerierung ausnutzen, dass der Generator auf Template-Ebene Polymorphisms unterstützt. Konkret bedeutet dies folgendes: Wenn in einer Template-Datei mehrere Templatedefinitionen gleichen Namens vorhanden sind die aber für verschiedene Metaklassen gelten, so expandiert der Generator für ein Modellelement immer die Template, deren Typdeklaration dem dynamischen Typ des Modellelements am genauesten entspricht.

Was bedeutet dies nun für die Generierung der Interfaces aus der textuellen Syntax? Jeder Baumknoten hat eine Operation *Child*() die die Menge seiner Kinder zurückliefert. Unter Ausnutzung der polymorphen Templates lässt sich die Generierung der Java-Interfaces folgendermaßen realisieren:

```
«DEFINE JavaImpl FOR ASTInterface»
  interface «Name» {
    «EXPAND JavaImpl FOREACH Child»
  }
«ENDDEFINE»
```

```
«DEFINE JavaImpl FOR ASTOperation»
  «Type» «Name»(«EXPAND JavaImpl FOREACH Child
                       USING SEPARATOR ","»);
«ENDDEFINE»

«DEFINE JavaImpl FOR ASTParameter»
  «Type» «Name»
«ENDDEFINE»
```

Dem aufmerksamen Leser wird nicht entgangen sein, dass wir noch die eine oder andere Operation in den AST-Klassen implementieren müssen; zum Beispiel *Type()* in *ASTOperation* und *ASTParameter* sowie *Child()* auf Ebene des *SimpleNodes* (weil dies für alle AST-Knoten relevant ist). Die ersten beiden sind trivial, sie geben nur die vorher jeweils gesetzten Werte für den Typ zurück, *Child()* ist auch nicht weiter dramatisch:

```
public ElementSet Child() {
  ElementSet s = new ElementSet();
  if ( children != null ) {
  for (int i = 0; i < children.length; i++) {
    SimpleNode n = (SimpleNode)children[i];
      s.add( n );
    }
  }
  return s;
}
```

Fazit

Durch die Verwendung von Parsergeneratoren in Verbindung mit (geeigneten) Codegeneratoren lassen sich textuelle Spezifikationen im Rahmen modellgetriebener Entwicklung einfach nutzen und mit anderen, z.B. grafischen Modellen integrieren. Schlüssel dazu ist, dass der Codegenerator intern Modelle als (Java-)Objektgraph repräsentiert. Da Parsergeneratoren dies oft genauso machen, liegt eine Integration nahe und ist in der Praxis mit wenig Aufwand zu bewerkstelligen.

Referenzen

[oAW] openArchitectureWare Generator, http://www.sourceforge.net/projects/architecturware

[oAWIntro] Markus Völter, Metamodellbasierte Codegenerierung in Java

[JavaCC] JavaCC Project Homepage, http://javacc.dev.java.net