Methodology

Met de release van het Java 6 platform is het onderwerp 'Domain Specific Languages' (DSL) nu officieel een hot item voor Java ontwikkelaars. De opname van JSR 223 ('Scripting for the Java platform') in de Standard Edition heeft niet eens zo heel veel mogelijk gemaakt dat we voorheen nog niet konden. Desondanks valt te verwachten dat de combinatie van Java met scripttalen en eigen gedefinieerde talen, in 2007 een grote vlucht zal gaan nemen. In dit artikel kijken we naar het gebruik en naar het zelf definiëren van domeinspecifieke talen vanuit Java.

Domeinspecifieke talen met JavaCC en JSR 223

Martin Wolf
is als consultant werkzaam bij Info Support. Hij heeft bij een grote verscheidenheid aan klanten praktijkervaring opgedaan met applicatieontwikkeling in Java en .NET, en is momenteel vooral werkzaam als software-architect op JEE-projecten.

Java is zo'n geweldige taal; waarom zou je hem niet overal voor willen gebruiken? Er zijn aardig wat redenen te bedenken. Om te beginnen is Java een general-purpose taal en zijn voor specifieke toepassingen vaak efficiëntere weergavevormen te bedenken. Een goed voorbeeld daarvan is de aloude reguliere expressie, voor het beschrijven van de regels waaraan een gegeven string moet voldoen. Om een postcode te beschrijven ben ik klaar met '\d{4}[A-Z]{2}' en een aardige benadering voor floating-point getallen is

```
[+-]?\d+(\.\d+)?([Ee][+-]?\d+])?
```

Toegegeven, met name die tweede ziet er wat intimiderend uit voor de regexp-nieuweling, maar lang niet zo intimiderend als de vele honderden regels Java code die anders nodig geweest zouden zijn om te bevestigen dat 6.022E+23 een geldig getal is.

Bovendien is Java een gecompileerde taal. Dat is fijn voor statische applicatiecode, maar minder toepasselijk voor zaken die op runtime gewijzigd moeten kunnen worden, zoals configuratiebestanden, business rules, of dynamische content op een webpagina (denk aan JSP of JSF pagina's die EL expressies bevatten).

Een ander voorbeeld van een situatie waarin hercompilatie niet praktisch zou zijn, is een grafische desktop-applicatie (zoals een tekstverwerker of een e-mail client) die door de gebruiker moet kunnen worden aangepast met behulp van macro-faciliteiten. In zo'n geval is een taal gewenst die aan expressiviteit niet onderdoet voor Java zelf en waarmee een groot deel van het interne objectmodel van de applicatie kan worden benaderd, maar die geen compilatie vereist. Het in de applicatie integreren van een scripttaal zoals Groovy of Jython zou in dat geval de logische keuze zijn.

Het is zelfs mogelijk, de macro-taal onderdeel te laten uitmaken van de applicatie-architectuur. De in de Unix-wereld tot op de dag van vandaag populaire editor Emacs is daar het ultieme voorbeeld van. Deze editor is opgebouwd in de vorm van een library met hulpcomponenten, geschreven in C, met daar bovenop een interpreter voor de klassieke programmeertaal Lisp. De feitelijke teksteditor is vervolgens geschreven in Lisp en kan worden aangepast door de gevorderde eindgebruiker. Het resultaat van deze architectuurkeuze is een vrijwel onbegrensde uitbreidbaarheid; vrijwel iedere gewenste feature kan worden bijgebouwd zonder de basiscomponenten te hercompileren.

Tenslotte moet een deel van de configuratie van een applicatie vaak onderhouden kunnen worden door niet-programmeurs, die gespecialiseerd zijn in een bepaald businessdomein maar met weinig of geen programmeerkennis. Een voorbeeld hiervan is een applicatie waarin de kredietwaardigheid van een potentiële klant moet worden beoordeeld. Zo'n kredietscore bestaat typisch uit een enorme boom van conditionele tests, die ieder een kleine bijdrage doen aan de eindscore. Die bomen worden normaal gesproken onderhouden door kredietspecialisten en niet door softwareontwikkelaars, maar de specialist wil wel zelf-

standig aanpassingen kunnen doen zonder daarvoor iedere keer een change request te moeten indienen bij het ontwikkelteam. Om dit mogelijk te maken zijn rule engines ontwikkeld, een goed voorbeeld van een domeinspecifieke taal.
De de facto taal om dergelijke complexe configuraties in te beschrijven, is natuurlijk XML. Het wordt ondersteund door ieder zichzelf respecterend ontwikkelplatform van de laatste zes jaar; in Java bijvoorbeeld door object-XML mapping frameworks zoals JAXB. Dankzij grafische tools zoals Altova XMLSpy is het ook relatief eenvoudig te gebruiken voor de meer technisch georiënteerde eindgebruiker. *Relatief* eenvoudig. Want XML is prima bruikbaar voor computer-naar-computer communicatie (wanneer performance geen halszaak is), maar heeft serieuze beperkingen als mens-naar-computer interface. Die stelling zal worden bevestigd door iedereen die wel eens handmatig een complexe configuratiefile heeft moeten bewerken voor bijvoorbeeld Hibernate, of J2EE 1.4 entity bean mappings. Goede grafische tooling kan helpen, maar de ontwikkeling daarvan is voor een custom-applicatie meestal niet kosten-efficiënt. Producten als XMLSpy proberen het probleem generiek op te lossen, maar slagen daar niet altijd even goed in.

Kortom: Domain Specific Languages (DSL) behoren tot het gereedschap dat een applicatie-architect in zijn gereedschapskist dient te hebben, zelfs al zal het niet altijd bovenop liggen. Laten we dus eens gaan kijken hoe we dat kunnen aanpakken.

JSR 223

We onderkennen twee mogelijke situaties: het integreren van een bestaande taal (anders dan Java) in onze Java-applicatie, en het ontwikkelen van een volledig eigen taal. Voor de eerste optie bestonden tot voor kort verschillende de facto standaard API's, met als populairste kandidaat het Apache Bean Scripting Framework. Maar sinds Java 6 is de winnaar bekend: JSR 223, 'Scripting for the Java platform'. Deze API wordt standaard bijgeleverd bij Java 6 Standard Edition en beschrijft een generieke interface voor het beschrijven van Scripting Engines voor specifieke talen. Standaard wordt er een scripting engine bijgeleverd voor Mozilla Rhino, een open-source implementatie van de taal JavaScript.
JSR 223 beschrijft hoe een applicatie stukken scriptcode aan de scripting engine kan doorgeven en hoe data ermee kan worden uitgewisseld. In het volgende eenvoudige voorbeeld wordt een JavaScript-engine geïnstantieerd, twee variabelen krijgen een waarde, een regel JavaScript-code wordt gebruikt om de waarden bij elkaar op te tellen, en het resultaat wordt weer uitgelezen door de aanroepende Java-code.

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

class JSR223Demo {

  public static void main(String[] args)
                                throws Exception {
    ScriptEngineManager sem
        = new ScriptEngineManager();
    ScriptEngine engine
        = sem.getEngineByName("JavaScript");

    engine.put("a", 3);
    engine.put("b", 3);
    engine.eval("c = a + b;"); // JavaScript code!
    Object c = engine.get("c");
    System.out.println(c);
  }
}
```

In dit voorbeeld wordt de scriptcode aangeleverd middels een hardcoded string in de aanroepende Java-code. De eval method biedt echter ook de mogelijkheid, scriptcode te lezen uit een InputStream en zodoende bijvoorbeeld macro's in te lezen uit een tekstbestand. Dit alles is mogelijk in Java 6, met faciliteiten die allemaal standaard in de run-time library aanwezig zijn. Daarnaast is een groot aantal andere talen beschikbaar, zoals Groovy en BeanShell (twee populaire general-purpose scripttalen die sterk op Java lijken), JEP (een uitgebreide expressietaal voor wiskundige formules), Python en Ruby. Zie https://scripting.dev.java.net/ voor links naar de op dit moment via JSR223 beschikbare talen.

Zwarte magie

Maar wat nu als we echt een taal willen bouwen die helemaal op ons probleemgebied is toegespitst (soms een "micro language" genoemd)? Dan zullen we zelf een parser moeten bouwen. Nu wordt het schrijven van parsers door veel ontwikkelaars als zwarte magie beschouwd, en inderdaad is voor het implementeren van een taal als JavaScript een stevige theoretische ondergrond vereist. Maar general-purpose talen zijn er al genoeg, en het bouwen van een eenvoudig taaltje in een beperkt probleemdomein is een stuk eenvoudiger dan de meeste mensen denken. In dit artikel zullen termen als LALR parsers, Backus-Naur form en look-ahead expressies dan ook zorgvuldig worden vermeden.
Wat we daarvoor nodig hebben is een parsergenerator, van oudsher een compiler-compiler genoemd. Mensen die het C-tijdperk nog hebben mee gemaakt zullen zich wellicht de tools lex en yacc herinneren; yacc staat voor Yet Another Compiler-Compiler, hetgeen aangeeft dat de term alweer even mee gaat.
De de-facto parsergenerator voor Java is JavaCC, als open-source product te vinden op https://javacc.dev.java.net/. Deze tool interpreteert grammaticafiles, bestaande uit een beschrijving van de taal plus bijbehorende Java-code.

Java is een geweldige taal, waarom zou je hem niet overal voor willen gebruiken?

De uitvoer van de tool is een aantal Java-files, die zonder verdere aanpassing kunnen worden gecompileerd en in een grotere applicatie worden opgenomen. Het zal niemand verbazen dat JavaCC zelf gebouwd is met JavaCC, maar de tool is zeker niet beperkt tot het genereren van compilers die Java code produceren.

Voorbeeld: kredietscore

Als voorbeeld gaan we kijken naar een zeer eenvoudig taaltje voor het uitvoeren van kredietscores, zoals eerder in dit artikel behandeld. Wat we gaan bouwen is een parser voor het verwerken van scriptbestanden in het volgende formaat:

```
inkomen < 100000:      basisscore
schuld < inkomen:      30
inkomen < schuld:     -10
hypotheek < inkomen:   50
```

Iedere regel van dit script bestaat uit een vergelijking van twee waarden, gevolgd door een score die bij het totaal moet worden opgeteld of ervan afgetrokken. Elk van de twee waarden, alsmede de score, kan een getal zijn of een extern aangeleverde waarde. De extern aangeleverde waarden zullen worden gelezen uit een tekstbestand in het standaard Java property's formaat:

```
basisscore = 50
inkomen    = 60000
schuld     = 12000
hypotheek  = 300000
```

Om hiervoor in JavaCC een parser te genereren, schrijven we een file CreditScore.jj. Allereerst moet daarin de Java class worden gedefinieerd waarbinnen JavaCC de code van onze parser zal plaatsen. Dat ziet er als volgt uit:

```
PARSER_BEGIN(CreditScore)
  import java.util.Properties;
  import java.io.FileInputStream;

  public class CreditScore {
    private static Properties props
        = new Properties();

    public static void main(String args[])
                        throws Exception {
      props.load(new
          FileInputStream("test.props"));
      CreditScore parser = new CreditScore(
          new FileInputStream("test.script"));
      int totalScore = parser.Expressions();
      System.out.println("Total score: "
          + totalScore);
    }
  }
PARSER_END(CreditScore)
```

Tussen PARSER_BEGIN en PARSER_END kan willekeurige Java-code worden geplaatst. In de code hierboven wordt de scriptcode gelezen uit de file test.script, en de waarden die door het script moeten worden verwerkt worden gelezen uit de file test.props. Het feitelijk inlezen en uitvoeren van het script wordt gedaan door de method Expressions, die we zo meteen gaan definiëren. Terug in CreditScore.jj definiëren we vervolgens de tokens die in onze scripttaal kunnen voorkomen. Dat zijn: property's, getallen, het kleiner-dan teken en de dubbele punt. Alle witruimte tussen de tokens moet worden genegeerd.

```
SKIP: { " " | "\t" | "\n" | "\r" }
TOKEN: {
    <LESSTHAN:    "<"                      >
  | <COLON:       ":"                      >
  | <PROPERTY:    (["a"-"z","A"-"Z"])+     >
  | <NUMBER:      ("-")?(["0"-"9"])+       >
}
```

De tokens worden gedefinieerd in een variant op reguliere expressies, maar iets minder compact (en daardoor mogelijk iets leesbaarder) dan de voorbeelden die we in het begin van dit artikel zagen. De rest van CreditScore.jj bestaat uit een definitie van de grammatica van onze taal. De onderdelen daarvan heten productions en lijken enigszins op Java-methods, en kunnen ook waarden opleveren en binnen krijgen zoals in Java. Het zal dan ook geen verwondering wekken dat ze, tijdens de compileerslag door JavaCC, vertaald worden naar methods. Eén ervan is de method Expressions, die we in onze main method al tegenkwamen en die het entry-point vormt van onze parser.

```
int Expression() :
{ int score1 = 0, score2 = 0; }
{
  ( score1 = Expression()
    score2 = Expressions() | <EOF> )
  { return score1 + score2; }
}
```

In tegenstelling tot een normale Java-method, heeft een JavaCC 'production' twee body's. De eerste kan gebruikt worden voor Java-code voor het initialiseren van variabelen. De tweede bevat de feitelijke parser, en beschrijft een deel van de input, in dit geval "een Expression, gevolgd door ofwel een Expressions, ofwel het einde van de input". Met andere woorden: Expressions bestaat uit het één of meer keer voorkomen van Expression. Iedere Expression is zelf ook weer een production die een waarde oplevert, en de resultaten worden bij elkaar opgeteld en geretourneerd door de methode die JavaCC voor ons zal genereren.
De volgende twee productions definieren een Expression, die bestaat uit een enkele regel in ons script, en een Value, die bestaat uit ofwel een getal, ofwel een property waarvan de waarde moet worden uitgelezen uit test.props.

De tokens worden gedefinieerd in een variant op reguliere expressies

```
int Expression() :
{ int val1, val2, score; }
{
  (val1 = Value() <LESSTHAN> val2 = Value()
   <COLON> score = Value())
  { return val1 < val2 ? score : 0; }
}

int Value() :
{ Token t = null; }
{
  t = <NUMBER>
  { return Integer.parseInt(t.image); }
  | t = <PROPERTY>
  { return Integer.parseInt(
         props.getProperty(t.image)); }
}
```

Nadat we CreditScore.jj hebben geschreven, kunnen we hem vertalen naar Java-code en vervolgens compileren en uitvoeren:

```
javacc CreditScore.jj
javac  CreditScore.java
java   CreditScore
```

Voor een extreem eenvoudig taaltje als dit was de inzet van een parsergenerator natuurlijk overkill. Maar met dit startpunt kunnen eenvoudig uitbreidingen worden toegevoegd. Probeer bijvoorbeeld zelf eens conditionele expressies te implementeren, zodat de volgende scriptcode kan worden geïnterpreteerd.

```
if schuld < inkomen {
  Score 10
  if hypotheek < inkomen {
        Score 7
  }
}
else {
  Score -2
}
```

Wanneer de taal complexer wordt, zul je er normaal gesproken overigens voor kiezen om de .jj file zo beperkt mogelijk te houden; het doel van de parser is dan slechts om een objectboom te genereren die vervolgens door normale Java-code verder kan worden verwerkt. De source code van JavaCC zelf is hier een mooi voorbeeld van.

Eigen scripttaal toevoegen aan Java 6

JavaCC ondersteunt Java versies vanaf 1.2. Maar ben je een early adopter en maak je al gebruik van Java 6, dan wil je natuurlijk dat je gloednieuwe DSL kan worden ingezet via de JSR 223 API. Gelukkig is ook dat niet zo moeilijk. Als onderdeel van de API wordt een AbstractScriptEngine geleverd waarin het grootste deel van het werk al gedaan is. Voor het implementeren van de ScriptEngine interface moeten dan nog vier methods worden geïmplementeerd: createBindings voor het tot stand brengen van een binding tussen scriptvariabelen en de scripting engine, getFactory om de bijbehorende ScriptEngineFactory te vinden, en twee varianten van eval. Met enkele kleine aanpassingen aan onze parser, zou de implementatie van ScriptEngine.eval(String, StringContext) er als volgt uit kunnen zien:

```
public Object eval(Reader reader,
         ScriptContext context)
         throws ScriptException {
  try {
    CreditScore cs = new CreditScore(reader);
         return cs.Expressions();
  }
  catch (ParseException e) {
    throw new ScriptException(e);
  }
}
```

Hoe zorgen we nu dat de rest van de wereld onze scripting engine ook inderdaad kan gebruiken? Daartoe moeten we om te beginnen de ScriptEngineFactory implementeren. Deze bevat twaalf methods, waarvan de meeste gelukkig eenvoudig genoeg zijn in te vullen.
Scripting engines worden geregistreerd middels het standaard JAR service provider mechanisme. Dat komt er in dit geval op neer dat je in de JAR-file waarin je de implementatie verspreidt, in de folder /META-INF/services, een tekstbestand opneemt met de naam javax.script.ScriptEngineFactory, met daarin één regel tekst bestaande uit de fully-qualified name van de factory class. Vervolgens kan iedere applicatie die over deze JAR-file beschikt, onze scripting engine gebruiken zoals eerder in dit artikel beschreven.

Conclusie

Wanneer behoefte is aan een domeinspecifieke taal om data aan te leveren die op run-time door een applicatie moet kunnen worden geïnterpreteerd, is XML vaak 'de weg van de minste weerstand', maar is in veel gevallen niet de optimale oplossing. In Java 6 is het inzetten van een bestaande scripttaal erg eenvoudig. Het zelf bouwen van een beperkte (special-purpose) taal is wellicht niet triviaal, maar toch een stuk eenvoudiger dan veel ontwikkelaars denken.
Natuurlijk is het doel van dit artikel niet om ontwikkelaars ertoe aan te sporen om voor ieder wissewasje een eigen taal te gaan definiëren. Daarvoor is JavaCC toch nog net wat te complex. Maar het kan zeker geen kwaad om dit stukje gereedschap in de gereedschapskist te hebben. Daarnaast is het ook zeker een interessante gewaarwording om eens vanaf de andere kant tegen een compiler aan te kijken. De aldus opgedane ervaring kan altijd van pas komen, zelfs wanneer je uiteindelijk besluit om je probleem toch maar met XML op te lossen. «

XML heeft serieuze beperkingen als mens-naar-computer interface