Quellcode zum Download

NEUE SERIE

Fachliche Domain Specific Languages mit Groovy entwickeln

Ist das Groovy?

Groovy eignet sich als dynamische Programmiersprache ausgezeichnet für die Erstellung von fachlich orientierten Domain Specific Languages (DSLs). Der Artikel zeigt, wie eine solche DSL in Groovy definiert werden kann.

von Konstantin Diener

W-JAX Speaker

Eine Domain Specific Language wird zur kompakten Beschreibung von Sachverhalten in einer bestimmten Domäne beziehungsweise einem Fachgebiet verwendet. Für technische Domänen haben diese Beschreibungen mittlerweile einen starken Verbreitungsgrad erlangt und werden von Open-Source-Frameworks wie Apache Camel [1] beispielsweise zur Konfiguration von Messaging-Routen eingesetzt. In diesem Artikel sollen DSLs im Fokus stehen, die der kompakten Beschreibung von Sachverhalten aus einer fachlichen Domäne dienen. Die Definition der DSL erfolgt dabei mit dem Ziel, in puncto Struktur und Vokabular möglichst nah am Sprachgebrauch der entsprechenden Fachleute zu bleiben. So werden diese Softwareanwender in die Lage versetzt, sehr schnell selbstständig die Abläufe oder Strukturen ihrer Fachdomäne in der Anwendung zu beschreiben. Bei der in diesem Artikel als Beispiel beschriebenen Domäne handelt es sich um die Modellierung von Entscheidungsregeln für Immobilienkredite, die unter anderem aus den folgenden Parametern bestimmen, ob ein Kreditangebot unterbreitet wird:

- Einkommenssituation des Antragstellers
- Ausgabensituation des Antragstellers
- Art und Nutzung (vermietet oder selbstgenutzt) der Immobilie

Die Kreditentscheidung wurde für das Beispiel stark vereinfacht und basiert im Wesentlichen auf dem monatlichen, frei verfügbaren Betrag, der dem Antragsteller nach Abzug aller regelmäßigen Ausgaben (Sozialabgaben, Versicherungen, Mieten etc.) und der berechneten Kreditrate

Artikelserie

Teil 1: Definition der Groovy DSL

Teil 2: Integration einer Groovy DSL in eine Java-Anwendung

Abb. 1: Berechnung des Wertes für die Kreditentscheidung

Abb. 2: Berechnung des monatlichen Nettoeinkommens

von seinem Verdienst verbleibt (**Abb. 1**). Wenn dieser Betrag die Deckung der übrigen Lebenshaltungskosten erlaubt, wird der Kredit gewährt, ansonsten nicht.

Berechnungen

Zur Umsetzung der Entscheidungsregeln werden eine interne DSL [2] in Groovy definiert (eine externe DSL würde nicht Groovy als Basis verwenden, sondern eine völlig neue Sprache definieren) und die einzelnen Kreditentscheidungsregeln in Form von Textdateien abgelegt, die ein ebenfalls in Groovy geschriebener JUnit Test auswertet. Die erste Regel stellt eine einfache Berechnung dar und ermittelt das monatliche Nettoeinkommen des Antragstellers (**Abb. 2**). Für die Umsetzung dieser Regel wird eine Textdatei (*monatlichesNettoeinkommen.formel*) mit dem folgenden Inhalt erstellt:

```
monatlichesBruttoeinkommen - Steuern - Sozialabgaben -
                                                sonstigeBelastungen
```

Da es sich bei der vorgestellten Domain Specific Language um eine interne DSL handeln soll, müssen wir die Textdatei im nächsten Schritt als Groovy-Quellcode interpretieren lassen. Dazu nutzen wir die Eigenschaft, dass Groovy die Ausführung von Skripten erlaubt. Diese Skripte sind mehr oder weniger kurze Codeschnipsel, die ohne die Definition einer Klasse implementiert werden können. Die zuvor beschriebene Datei *monatlichesNettoeinkommen.formel* kann als ein solches Groovy-Skript interpretiert werden. Durch die folgenden drei Zeilen in unserem in Groovy geschriebenen JUnit Test führen wir das Skript aus:

```
GroovyShell gs = new GroovyShell()
Script s = gs.parse(new File("monatlichesNettoeinkommen.formel"))
s.run()
```

Ausgangspunkt der Ausführung von Skripten ist die Klasse *GroovyShell*: Die Methode *parse* liest die DSL-Textdatei ein und kompiliert den Groovy-Code, auf den danach durch das *Script*-Objekt zugegriffen werden kann. Die Ausführung des Skripts führt allerdings zu einem Fehler:

```
groovy.lang.MissingPropertyException: No such property:
        monatlichesBruttoeinkommen for class: monatlichesNettoeinkommen
```

An dieser Exception lässt sich erkennen, wie die Übersetzung des Groovy-Skripts erfolgt ist: Um den Skriptcode wurde eine Klasse „herumgeneriert“, die denselben Namen trägt wie die Skriptdatei. Bei der Ausführung des Skripts kann die Laufzeitumgebung weder in dieser Klasse ein Attribut mit dem Namen *monatliches Nettoeinkommen* noch eine im Skript definierte lokale Variable mit einem entsprechenden Namen finden. Damit

Listing 1

```
Binding b = new Binding()
b.setVariable("monatlichesBruttoeinkommen", 2300.0)
b.setVariable("Steuern", 210.0)
b.setVariable("Sozialabgaben", 400.0)
b.setVariable("sonstigeBelastungen", 160.0)

GroovyShell gs = new GroovyShell(b)
Script s = gs.parse(new File("monatlichesNettoeinkommen.formel"))
def result = s.run()
```

Listing 2

```
class DslTestCase extends GroovyTestCase {

    @Test
    public void testNettoEinkommen() {
        def result = runDsl("nettoEinkommen.formel",
            [
                monatlichesBruttoeinkommen: 2300.0,
                Steuern: 210.0,
                Sozialabgaben: 400.0,
                sonstigeBelastungen: 160.0
                ])

        assert result == 1530.0
    }

    def runDsl(String scriptName, Map p) {
        Binding b = new Binding(p)

        GroovyShell gs = new GroovyShell(b)
        Script s = gs.parse(new File(scriptName))

        def result = s.run()
    }
}
```

die Ausführung des Skripts funktioniert, müssen die im Skript verwendeten Properties vor der Ausführung also „injiziert“ werden. Zu diesem Zweck verwenden wir ein *Binding* (Listing 1).

Die Methode *Script.run()* gibt den Wert zurück, den der letzte Ausdruck des ausgeführten Skripts erzeugt – in unserem Fall das Ergebnis der Berechnung: 1530.0. Um die Skriptausführung in mehreren Testmethoden nutzen zu können, wird in der JUnit-Testklasse eine eigene Methode angelegt und der Aufruf des Skripts entsprechend angepasst (Listing 2). Die einzelnen Properties können in Groovy direkt durch die Schreibweise *<Schlüssel>: <Wert>* ausgedrückt werden. Das Binding besitzt einen Konstruktor, der direkt die resultierende Map entgegennimmt.

Entscheidungen

In unserem Beispiel bestehen die Regeln zur Kreditvergabe zum einen Teil aus Berechnungen und zum anderen Teil aus Entscheidungselementen. Das erste DSL-Beispiel zeigt, dass die Möglichkeit, Berechnungen durchzuführen, durch die Verwendung von Groovy als „Wirtssprache“ ohne nennenswerte Implementierungsarbeiten zur Verfügung steht. Die zweite exemplarische Regel enthält ein Entscheidungselement: Der monatlich frei verfügbare Betrag wird in Abhängigkeit davon berechnet, ob die zu

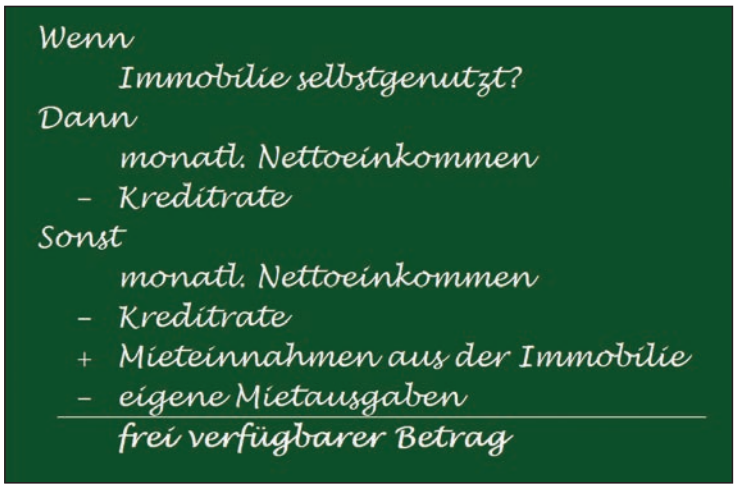

Abb. 3: Berechnung des monatlich frei verfügbaren Betrags

Listing 3

```
Wenn Immobilie.selbstgenutzt,
    Dann:
        Antragsteller.nettoEinkommen - Kreditrate,
    Sonst:
Antragsteller.nettoEinkommen - Kreditrate + Immobilie.monatlicheMiete
- Antragsteller.monatlicheMiete
```

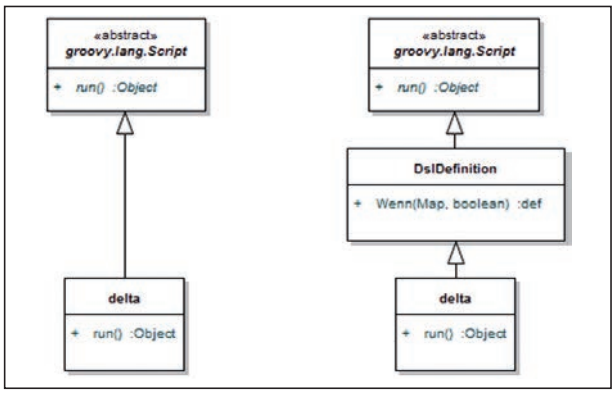

Abb. 4: Klassenhierarchie des Groovy-Skripts

erwerbende Immobilie durch den Antragsteller selbst genutzt oder vermietet werden soll (**Abb. 3**). Für diese Regel wird die Datei *monatlichFreiVerfuegbarerBetrag.formel* mit dem Inhalt aus Listing 3 angelegt.

In der ersten Regel besteht die Berechnung ausschließlich aus einfachen Properties. Die Entscheidungsregel benutzt jetzt für die Immobilie und den Antragsteller zusammengesetzte Properties, um beispielsweise eine monatliche Miete sowohl für die Immobilie als auch für den Antragsteller zu modellieren (Vermeidung von Namenskonflikten). Für diese zusammengesetzten Properties werden die Klassen *AntragstellerObject* und *ImmobilieObject* angelegt:

AntragstellerObject:

```
class AntragstellerObject {

    def nettoEinkommen
    def monatlicheMiete
}
```

ImmobilieObject:

```
class ImmobilieObject {

    boolean selbstgenutzt
    def monatlicheMiete
}
```

Im Kasten „Kunstgriff mit <...*Object*>“ ist der Grund für die seltsam anmutende Namensgebung der beiden Klassen beschrieben. Beim Aufruf der Methode *runDsl* für die Skriptausführung werden die Objekte initialisiert und übergeben (Listing 4).

Für die Initialisierung der beiden zusammengesetzten Properties erhält der Konstruktor jeweils die gewünschten Attributwerte in Form von *Named*-Parametern (<*Attributname*>: <*Wert*>). Das Skript kann diese Werte dann über den Punktoperator auslesen, da in Groovy alle Attribute von außen zugreifbar sind. Die Ausführung der Testmethode führt trotz der Definition der zusammengesetzten Properties zu einem Fehler:

```
groovy.lang.MissingMethodException: No signature of method:
  monatlichFreiVerfuegbarerBetrag.Wenn() is applicable for argument types:
        (java.util.LinkedHashMap, java.lang.Boolean) values: [[Dann:1260.0,
                                                    Sonst:1040.0], true] ...
```

Die Laufzeitumgebung versucht erfolglos, in der Skriptklasse *monatlichFreiVerfuegbarerBetrag* eine Methode mit dem Namen *Wenn* zu finden. Genau wie im Fall der Properties muss diese Methode quasi von außen in das Skript injiziert werden. Der einfachste Weg nutzt die Vererbungshierarchie bei der Übersetzung von Groovy-Skripten (**Abb. 4**): Im ersten Beispiel zeigte sich, dass aus dem Skript eine Groovy-Klasse erzeugt wird, die denselben Namen wie die Skriptdatei trägt. Diese Klasse ist im Normalfall von *groovy.lang.Script* abgeleitet (linke Seite der Abbildung). Durch Anpassung der Compiler-Konfiguration kann eine andere Basisklasse für das Skript verwendet werden, die von *groovy.lang.Script* abgeleitet sein muss (rechte Seite der Abbildung):

```
CompilerConfiguration cc = new CompilerConfiguration(
    scriptBaseClass: DslDefinition.class.name)

GroovyShell gs = new GroovyShell(b, cc)
Script s = gs.parse(new File(scriptName))
```

Die verwendete Klasse *DslDefinition* ergänzt die Klassenhierarchie um die benötigte Methode *Wenn*

Listing 4

```
def result = runDsl("monatlichFreiVerfuegbarerBetrag.formel", [
    Immobilie: new ImmobilieObject(
        selbstgenutzt: false,
        monatlicheMiete: 430.0),
    Antragsteller: new AntragstellerObject(
        nettoEinkommen: 1530.0,
        monatlicheMiete: 650.0),
    Kreditrate: 270.0])

assert result == 1040.0
```

Listing 5

```
abstract class DslDefinition extends Script {

    def Wenn(Map m, boolean condition) {
        if (condition) {
            m.Dann
        }
        else {
            m.Sonst
        }
    }
}
```

Es können beliebig viele Argumente als Schlüssel-Wert-Paare übergeben werden.

(Listing 5). Die Methode nimmt als Parameter eine Bedingung vom Typ *boolean* entgegen. Die beiden Elemente *Dann* und *Sonst* werden auf *Named*-Parameter [3] abgebildet, die die erweiterte Form der dynamischen Parameterlisten in Java sind: Es können beliebig viele Argumente als Schlüssel-Wert-Paare übergeben werden. In der Methodendefinition wird eine Map definiert, die immer an erster Stelle in der Parameterliste stehen muss und die *Named*-Parameter aufnimmt. Wie in Groovy üblich, kann auf die Elemente dieser Map mit dem Punktoperator zugegriffen werden. Die runden Klammern beim Aufruf einer Methode sind in Groovy optional, sobald die Methode mindestens einen Parameter hat. Aus Gründen der Lesbarkeit wurde im DSL-Skript auf die Klammern verzichtet. In manchen Situationen soll eine Entscheidung nicht nur zwei Alternativen haben, wie ein aus Programmiersprachen bekanntes *if-then-else*, sondern beliebig viele – wie im Beispiel in Listing 6. Diese Art der Entscheidungen lassen sich durch eine ähnliche DSL-Methode realisieren wie für die Wenn-Dann-Sonst-Regeln:

```
def Falls(Map m, def key) {
   m[key]
}
```

Die einzelnen Alternativen werden der Methode als *Named*-Parameter übergeben, und der erste Parameter dient als Schlüssel, um die entsprechende Alternative auszuwählen und deren Wert zurückzuliefern.

Domänenfunktionen

Bislang enthält die Domain Specific Language ausschließlich aus Programmiersprachen bekannte Elemente, die ein weniger technisches Aussehen erhalten haben. Im nächsten Schritt soll die Miete für das zu erwerbende oder zu bauende Objekt verifiziert werden: Nur Mieten von unter zehn Euro pro Quadratmeter sollen als gültig angesehen werden, damit der Antragsteller die Situation nicht schönen kann – in einem realen System würde vermutlich ein Abgleich anhand von Postleitzahl und Größe mit den Daten aus dem Mietspiegel stattfinden. Die Regel wird wie folgt in einer DSL-Regel formuliert:

```
Miete Immobilie.monatlicheMiete für Immobilie.größe angemessen
```

Für diese Regel wurden zwei Eigenschaften der Sprache Groovy genutzt: zum einen die Möglichkeit, in Bezeichnern auch Umlaute verwenden zu dürfen, zum anderen die bereits bekannten optionalen Klammern. Werden keine Klammern oder Punkte ausgelassen, sieht der Ausdruck wie folgt aus:

```
Miete(Immobilie.monatlicheMiete).für(Immobilie.größe).angemessen
```

Der Groovy Parser geht hier davon aus, dass *Miete* eine Methode ist, deren Argumente durch Kommata getrennt folgen. Da kein Komma folgt, wird nur ein Argument zugeordnet und der Ausdruck nach dem nächsten Leerzeichen als neuer Methodenaufruf (*für*) interpretiert. Nach demselben Muster erfolgt die Zuordnung des Ausdrucks *Immobilie.größe* als Argument der Methode *für*. Auf den letzten Ausdruck folgen keine

Listing 6

```
Falls (Beruf,
 Renter:
  monatlicheRentenzahlungen - Steuern - sonstigeBelastungen,

 Pensionär:
  monatlichePensionszahlungen - Steuern - Sozialabgaben - sonstigeBelastungen,

 Student:
  monatlichesBruttoeinkommen - sonstigeBelastungen,

 Angestellter:
  monatlichesBruttoeinkommen - Steuern - Sozialabgaben - sonstigeBelastungen,

 Beamter:
  monatlichesBruttoeinkommen - Steuern - sonstigeBelastungen
)
```

Listing 7

```
abstract class DslDefinition extends Script {
   ...
   def Miete(def monatlicheMiete) {
      new MietFassade(monatlicheMiete: monatlicheMiete)
   }
}
```

Listing 8

```
class MietFassade {

   def monatlicheMiete

   MietPruefer für(def groesse) {
      new MietPruefer(monatlicheMiete: monatlicheMiete, groesse: groesse)
   }
}
```

Argumente, weshalb der Parser von einer Property (*angemessen*) ausgeht.

Das Element *Miete* wird als Methode in der Klasse *DslDefinition* ergänzt (Listing 7). Diese Methode liefert als Rückgabewert ein Objekt vom Typ *MietFassade* (Listing 8) zurück. Die eigentliche Prüfung findet dann in der Klasse *MietPruefer* statt (Listing 9): Das Skript liest die Property *angemessen* aus. Diese Property ist in der Klasse nicht direkt implementiert und wird über eine Callback-Methode realisiert, die ebenfalls eine Groovy-Spezialität ist: Beim Auslesen aller Properties wird die Methode *getProperty* aufgerufen, sofern sie vorhanden ist. Hat die referenzierte Property den Namen *angemessen*, wird der aktuelle Wert ermittelt und zurückgegeben, andernfalls der übliche Weg eingeschlagen, um Properties aufzulösen; wobei das „@“ zwingend erforderlich ist, damit keine endlose Rekursion entsteht – andernfalls würde nämlich wieder die Methode *getProperty* für die Auflösung des Properties aufgerufen.

Für das Beispiel ist die Regel für die Mietprüfung in der Klasse *MietPrüfer* hartkodiert. In einer realen Fachanwendung würde an dieser Stelle beispielsweise ein Service aufgerufen, der die Miete anhand von Postleitzahl und Größe mit den Daten aus dem Mietspiegel abgleicht.

Kunstgriff mit „<...Object>“

Es mutet etwas seltsam an, dass die Klasse, in der die Eigenschaften eines Antragstellers modelliert sind, nicht auch *Antragsteller* heißt. Die Ursache für diesen Kunstgriff liegt in der Tatsache, dass „Antragsteller“ im DSL-Skript groß geschrieben ist, um eine bessere Lesbarkeit für den Fachexperten zu ermöglichen. Hieße die Klasse *Antragsteller*, würde die Laufzeitumgebung den Ausdruck *Antragsteller.monatlichesNettoeinkommen* als Zugriff auf das statische Attribut (Klassenattribut) *monatlichesNettoeinkommen* der Klasse *Antragsteller* interpretieren. Aus diesem Grund bekam die Klasse einen anderen Namen als die Property im Skript. Alternativ könnte die Property im Skript umbenannt oder klein geschrieben werden, was aber möglicherweise die Lesbarkeit zugunsten einer einfacheren Implementierung verringert hätte.

Fazit

In mehreren Stufen wurde im Laufe des Artikels eine beispielhafte Domain Specific Language für die Formulierung von Kreditentscheidungsregeln definiert. Die Implementierung erfolgte auf Basis von Groovy, das den ersten Schritt, die Modellierung von Berechnungen, gleich out of the Box mitbringt. Darüber hinaus wur-

Listing 9

```
class MietPruefer {

    def monatlicheMiete
    def groesse

    boolean angemessen() {
        (monatlicheMiete / groesse) < 10
    }

    def getProperty(String name) {
        if (name == "angemessen") {
            angemessen()
        }
        else {
            this.@"$name"
        }
    }
}
```

den Entscheidungselemente und Domänenfunktionen beschrieben, mit denen tatsächliche Fachbegriffe Einzug in die Sprache halten. Es zeigt sich, dass Groovy vor allem auch wegen vieler optionaler Sprachelemente (zum Beispiel runde Klammern und Punkte bei Methodenaufrufen) sehr flexible Möglichkeiten zur Definition von DSLs bietet, die nahe an der Sprache der Fachexperten bleiben. Obwohl sich die DSL der natürlichen Sprache bereits angenähert hat, soll aber nicht verschwiegen werden, dass an dieser Stelle noch Weiterentwicklungspotenzial besteht. Schließlich muss der Fachanwender die von der DSL unterstützten Formulierungen immer noch sehr genau kennen – „wo kommt ein Punkt hin und wo nicht?“ Begleitend wurde gezeigt, wie die Integration der DSL-Skripte mit dem übrigen Programmcode realisiert werden kann. In der nächsten Ausgabe wird dann die Integration mit einer Java-Applikation genauer betrachtet. Die Quellcodes zum Artikel stehen in Form eines Maven-Projekts auf http://javamagazin.de zum Download bereit.

Konstantin Diener ist Leading Consultant bei der COINOR AG und im Bereich Wertpapierprozesse tätig. Er beschäftigt sich seit über zehn Jahren mit der Java-Plattform. Sein aktuelles Interesse gilt vor allem Business-Rules-Management-Systemen und Domain Specific Languages sowie agilen Methoden wie Scrum oder Kanban.

Links & Literatur

[1] http://camel.apache.org/dsl.html

[2] http://martinfowler.com/bliki/DomainSpecificLanguage.html

[3] http://groovy.codehaus.org/Statements