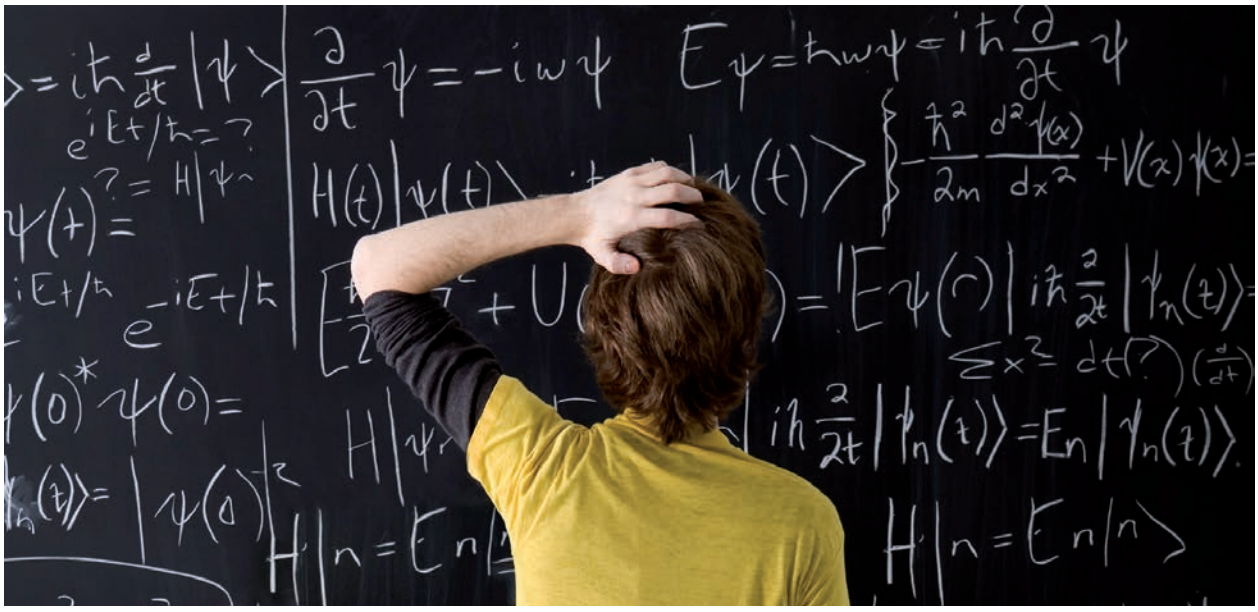
Quellcode auf DVD!



Integration einer Groovy-DSL mit einer Java-Anwendung

DSL-Komposition

Groovy eignet sich als dynamische Programmiersprache ausgezeichnet für die Erstellung von fachlich orientierten Domain Specific Languages (DSLs). Die Anwendung einer solchen DSL erfolgt aber nicht losgelöst, sondern im Allgemeinen im Kontext einer Fachapplikation, die selten selbst in Groovy, sondern eher in Java implementiert ist.

von Konstantin Diener

Im ersten Teil dieser Artikelserie wurde eine in Groovy implementierte Domain Specific Language (DSL) vorgestellt. Die DSL dient der Formulierung von Entscheidungsregeln für Immobilienkredite. Realisiert sind die einzelnen Regeln in Form von Groovy-Skripten wie dem folgenden:

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

Das Zielbild für diesen Artikel ist eine bestehende Fachapplikation (fiktives Beispiel) in Java, die an bestimmten Stellen durch eine mächtige fachliche Konfiguration ergänzt wird. Diese Konfiguration besteht aus Berechnungs- und Entscheidungsregeln, die durch Skripte in der angesprochenen Domain Specific Language realisiert sind (**Abb. 1**). Die Fachapplikation ist der Einfachheit halber als ein in Java implementierter JUnit-Testfall realisiert (Listing 1). Inhaltlich unterscheidet sich die Testfallimplementierung nicht von dem im letzten Artikel vorgestellten Groovy-Pendant. Durch die Syntax der Sprache Java ist sie aber wesentlich weniger kompakt und auch weniger gut lesbar (vor allem die Initialisierung der Property-Liste).

Maven Build

Releases für moderne Geschäftsanwendungen werden im Allgemeinen durch Build-Werkzeuge wie Ant, Maven oder Gradle erzeugt. Im Rahmen der Erweiterung einer solchen Anwendung um eine DSL-Konfiguration müssen die Groovy-Artefakte – wie beispielsweise die DSL-Sprachdefinition (Listing 2) – in den Build-Zyklus dieser Anwendung integriert werden. In unserem Beispiel haben wir einen bestehenden Maven-Build-Prozess, in den die Übersetzung der Groovy-Klassen mittels Eclipse Compiler eingebracht wurde.

Artikelserie

Teil 1: Definition der Groovy DSL
Teil 2: Integration einer Groovy DSL in eine Java-Anwendung

Domänenobjekte

Die in den DSL-Regeln formulierten Berechnungen oder Entscheidungen erfolgen auf der Basis von Domänenobjekten wie *Antragsteller* oder *Immobilie*. Beim Aufruf einer Regel durch die Java-Anwendung werden diese Domänenobjekte übergeben. Im letzten Teil des Artikels war das Domänenmodell der Einfachheit halber in Form von Groovy-Klassen implementiert und enthielt die Properties genau in der von den DSL-Skripten benötigten Form. In der Realität ist davon auszugehen, dass eine bestehende Fachapplikation bereits ein Domänenmodell besitzt. Deshalb soll nachfolgend betrachtet werden, wie ein bestehendes Domänenmodell in Form von Java-Klassen mit den Groovy-DSL-Regeln verknüpft werden kann. Dieses beispielhafte Modell besteht aus den drei folgenden Elementen (**Abb. 2**):

- *AmountWithCurrency*: Alle Geldbeträge in der Fachanwendung sind nicht als einfache Fließkommazahlen modelliert, sondern bestehen aus einer Fließkommazahl und einer Währung.
- *Applicant*: Der Antragsteller eines Kredits ist in der Anwendung als Domänenklasse modelliert, die aus dem monatlichen Nettoeinkommen (*takeHomeAmount*) und dem Wohnsitz (*domicile*) besteht.

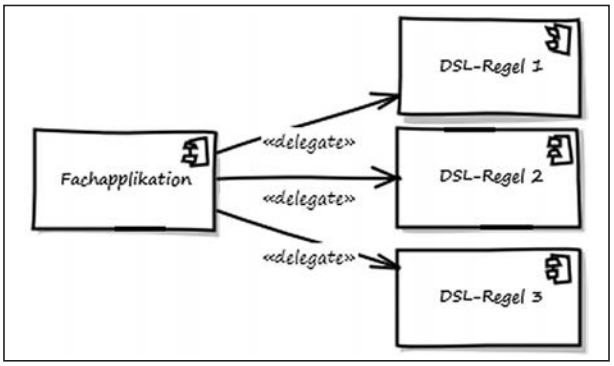


Abb. 1: Integration der DSL-Regeln in eine Fachapplikation

- *RealEstate*: Sowohl die zu erwerbende Immobilie als auch der Wohnsitz des Antragstellers werden über eine Klasse des Domänenmodells realisiert, die neben der Größe und der monatlichen Miete ein Flag zur Kennzeichnung selbstgenutzter Immobilien enthält.

Auch dieses Modell enthält aus Gründen der Übersichtlichkeit nur Attribute, die für die im Beispiel enthaltenen DSL-Skripte relevant sind.

Sollen beispielsweise für die Berechnung des monatlich frei verfügbaren Betrages (Listing 3) Objekte des bestehenden Domänenmodells als Werte für die Properties

Listing 1

```
public class JavaDslTestCase {

  @Test
  public void testNettoeinkommen()
      throws CompilationFailedException, IOException {
    Map<String, Object> p = new HashMap<String, Object>();
    p.put("monatlichesBruttoeinkommen", 2300.0);
    p.put("Steuern", 210.0);
    p.put("Sozialabgaben", 400.0);
    p.put("sonstigeBelastungen", 160.0);

    Object result = runDsl("monatlichesNettoeinkommen.formel", p);

    assertEquals(1530.0, result);
  }

  Object runDsl(String name, Map<String, Object> properties)
      throws CompilationFailedException, IOException {
    Binding b = new Binding(properties);

    CompilerConfiguration conf = new CompilerConfiguration();
    conf.setScriptBaseClass(DslDefinition.class.getName());
    GroovyShell gs = new GroovyShell(b, conf);

    Script s = gs.parse(new File(name));
    return s.run();
  }
}
```

Listing 2

```
abstract class DslDefinition extends Script {

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

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

  def Miete(def monatlicheMiete) {
    return new MietFassade(monatlicheMiete: monatlicheMiete)
  }
}
```

Listing 3

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

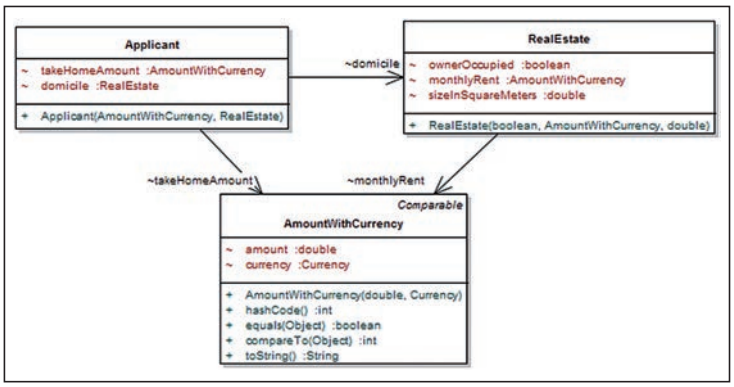


Abb. 2: Beispielhaftes Domänenmodell der Java-Fachapplikation

Immobilie (*RealEstate*) und *Antragsteller* (*Applicant*) verwendet werden, besteht in zweierlei Hinsicht Anpassungsbedarf:

1. Die in den DSL-Skripten verwendeten Properties *monatlichesNettoeinkommen* oder *miete* existieren nicht in den Java-Klassen des Domänenmodells.
2. Die zur Berechnung herangezogenen Geldbeträge sind vom Typ *AmountWithCurrency* und können derzeit nicht mit Rechenoperatoren wie „+“ oder „–“ verknüpft werden.

Da wir weder die Java-Klassen des Domänenmodells noch die DSL-Skripte ändern möchten, bedienen wir uns des Adapter-Musters [1] und legen eine neue Schnittstelle um die Klassen des Domänenmodells. In Java würde man diese Erweiterung über eine Wrapper-Klasse oder einen Proxy realisieren. In Groovy gibt es die Möglichkeit, der Klasse zur Laufzeit die benötigten Properties und Methoden mithilfe der Metaklasse hinzuzufügen (diese Änderungen gelten dann für die komplette VM)

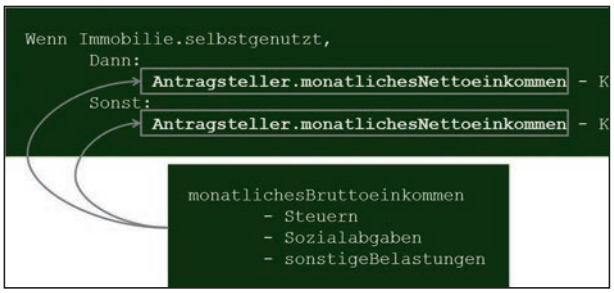


Abb. 3: Abhängigkeiten zwischen den DSL-Regeln

– ein lesenswerter Artikel zu Metaklassen in Groovy findet sich unter [2]. Die für die DSL benötigten Properties werden dabei sowohl 1:1 aus demselben Objekt gelesen (*getMonatlichesNettoeinkommen*), als auch aus verknüpften Objekten abgeleitet (*getMonatlicheMiete*):

```
Applicant.metaClass {
  getMonatlichesNettoeinkommen {
    delegate.takeHomeAmount
  }

  getMonatlicheMiete {
    delegate.domicile.monthlyRent
  }
}
```

Der Groovy-Code fügt der Metaklasse von *Applicant* zwei neue Properties hinzu. Groovy erlaubt es, diese beiden Properties dabei wahlweise als Attribute der Klasse oder indirekt in Form von *get*-Methoden zu realisieren. Da es sich bei dem Codeabschnitt mit den beiden Methoden um eine Closure handelt, wird zum Zugriff auf die modifizierte Java-Klasse nicht *this*, sondern *delegate* verwendet [3].

Listing 4

```
AmountWithCurrency.metaClass {

  plus { AmountWithCurrency operand2 ->
    if (delegate.currency == operand2.currency) {
      new AmountWithCurrency(
        delegate.amount + operand2.amount, delegate.currency)
    }
  }

  minus { AmountWithCurrency operand2 ->
    if (delegate.currency == operand2.currency) {
      new AmountWithCurrency(
        delegate.amount - operand2.amount, delegate.currency)
    }
  }

  div { Number operand2 ->
    new AmountWithCurrency(delegate.amount / operand2, delegate.currency)
  }
}
```

Listing 5

```
public class GroovyFormula {

  final String name;
  final String code;
  private Script script;

  public GroovyFormula(String name, String code,
                       CompilerConfiguration compilerConfiguration) {
    super();
    this.name = name;
    this.code = code;
    this.valuePlaceholders = valuePlaceholders;

    GroovyShell groovyShell = new GroovyShell(compilerConfiguration);
    script = groovyShell.parse(this.code);
  }

  public synchronized Object execute() {
    return script.run();
  }
}
```

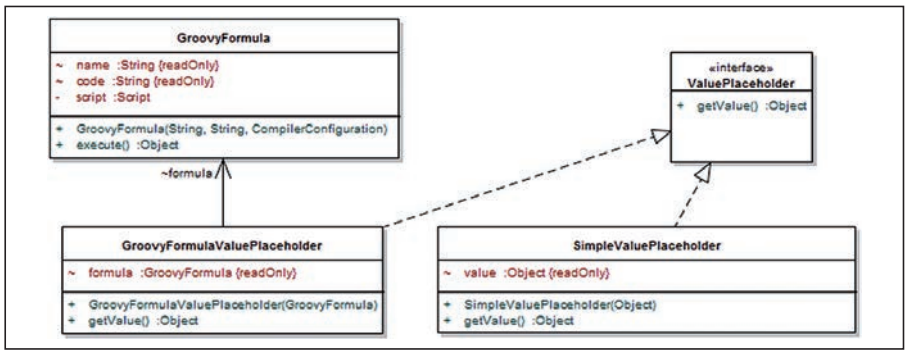


Abb. 4: Klassenmodell für Platzhalter in DSL-Skripten

Um eine nahtlose Integration des Java-Domänenmodells mit den DSL-Skripten zu erreichen, müssen Objekte der Klasse *AmountWithCurrency* noch in Rechenoperationen verwendbar sein. Groovy bietet generell die Möglichkeit, Operatoren zu überladen und so beispielsweise mit komplexen Objekten zu rechnen [4]. Das Überschreiben von Operatoren geschieht durch Methoden, die einer bestimmten Namenskonvention entsprechen (*plus*, *minus*, *multiply*, *div*). Auch diese Methoden ergänzen wir über die Metaklasse von *AmountWithCurrency* (Listing 4).

Im gewählten Beispiel soll es nur möglich sein, zwei *AmountWithCurrency*-Objekte zu addieren oder zu subtrahieren – eine Zahl kann nicht addiert oder subtrahiert werden. Aus Gründen der Übersichtlichkeit wurde die Behandlung von Operanden mit abweichender Währung im Beispiel ausgespart. Die dritte Methode implementiert die Division von Geldbeträgen durch eine einfache Zahl. Das Ergebnis aller Operator-Methoden ist wieder ein Geldbetrag vom Typ *AmountWithCurrency*.

Auflösen von Properties und Formeln

Bislang wurden alle DSL-Regeln einzeln durch die Java-Anwendung (in unserem Beispiel realisiert durch den Unit Test) aufgerufen, obwohl zwischen den einzelnen Regeln Abhängigkeiten bestehen: So wird in einer Regel das Nettogehalt des Antragstellers berechnet, und dieser Wert geht in die Berechnung des monatlich frei verfügbaren Betrags ein (**Abb. 3**).

Im Idealfall würden diese Abhängigkeiten wie Properties automatisch aufgelöst und die referenzierten Regeln ausgeführt. Die Properties und Referenzen auf andere Regeln wären sozusagen Platzhalter in den DSL-Regeln, deren Wert bei Bedarf ermittelt und der Regel zur Verfügung gestellt wird. Dieses Platzhalterkonzept bilden wir in einem Klassenmodell ab (**Abb. 4**):

- *ValuePlaceholder*: Die Schnittstelle abstrahiert alle Arten von Platzhaltern für DSL-Regeln und enthält nur die Methode *getValue*, die den Wert des Platzhalters zurückliefert.
- *SimpleValuePlaceholder*: Property-Werte werden durch einen Wrapper in den DSL-Skripten bereitgestellt, der die *ValuePlaceholder*-Schnittstelle implementiert.
- *GroovyFormula*: Bislang war das Laden und Ausführen der Groovy-Skripte direkt im JUnit-Test implementiert. Um beliebige Regeln bei Bedarf auszuführen, wird das Groovy-Skript in einer Klasse gekapselt, die auch die Registrierung der Skript-Basisklasse *DslDefinition* übernimmt (Listing 5).
- *GroovyFormulaValuePlaceholder*: Die Platzhalterimplementierung für Groovy-Formeln ruft in ihrer *getValue*- die *execute*-Methode des assoziierten *GroovyFormula*-Objekts auf (Listing 6).

Dem JUnit-Test wird ein Stück Code hinzugefügt, der die Formeln lädt und zusammen mit den festen Property-Werten in einer *Map* ablegt (Listing 7).

In den vorangegangenen Beispielen wurden die Werte der Properties über das Binding den DSL-Skripten direkt zur Verfügung gestellt. Wenn dieser Mechanismus beibehalten wird, müssten auch die Ergebnisse der referenzierten Regeln vor der Skript-Ausführung berechnet und bereitgestellt werden. Das würde be-

Listing 6

```
public class GroovyFormulaValuePlaceholder implements ValuePlaceholder {

  final GroovyFormula formula;

  public GroovyFormulaValuePlaceholder(GroovyFormula formula) {
    super();
    this.formula = formula;
  }

  @Override
  public Object getValue() {
    return formula.execute();
  }
}
```

Listing 7

```
...
  valuePlaceholders.put("Kreditrate", toAmountPlaceholder(270.0));

  String[] formelFiles = new File("./").list(new FilenameFilter() {

    @Override
    public boolean accept(File directory, String filename) {
      return filename.endsWith(FORMEL);
    }
  });

  CompilerConfiguration compilerConfiguration = new
                                                CompilerConfiguration();
  compilerConfiguration.setScriptBaseClass(DslDefinition.class.getName());

  for (String formelFile : formelFiles) {
    String code = IOUtils.toString(new FileInputStream(
      new File(formelFile)), "UTF-8");
    String formulaName = formelFile.replace(FORMEL, "");
    GroovyFormula groovyFormula = new GroovyFormula(
        formulaName, code, compilerConfiguration);
    valuePlaceholders.put(formulaName,
      new GroovyFormulaValuePlaceholder(groovyFormula));
  }

  ValuePlaceholder vp =
      valuePlaceholders.get("monatlichFreiVerfuegbarerBetrag");
  Object result = vp.getValue();

  assertEquals(new AmountWithCurrency(1140.0, EURO_CURRENCY),
                                                        result);
}
...

  private ValuePlaceholder toAmountPlaceholder(double d) {
    return new SimpleValuePlaceholder(new AmountWithCurrency(d,
                                                EURO_CURRENCY));
  }
```

deuten, dass der Aufrufer des Skripts – in diesem Fall die Klasse *GroovyFormula* – alle relevanten Regeln vorab ermitteln würde (beispielsweise durch eine Analyse des Abstract Syntax Tree). Um diesen Aufwand zu verhindern, stellen wir den DSL-Skripten nicht mehr direkt die Werte der einzelnen Platzhalter, sondern die Platzhalter selbst zur Verfügung. Dieses Vorgehen hat zunächst zur Folge, dass sich die Platzhalter nicht mehr wie bisher bequem über ihren Namen auslesen lassen, sondern auf dem Platzhalterobjekt immer die Methode *getValue* aufgerufen werden muss (was in Groovy gleichbedeutend mit einem Zugriff auf die Property *value* ist):

```
monatlichesBruttoeinkommen.value - Steuern.value - Sozialabgaben.value
  - sonstigeBelastungen.value
```

Um zu vermeiden, dass die Lesbarkeit und Verständlichkeit der DSL-Skripte durch die vielen *.value*-Ausdrücke

Listing 8

```
GroovyShell groovyShell = new GroovyShell(compilerConfiguration)
Script script = groovyShell.parse(this.code)
script.metaClass.getProperty { String name ->
  ValuePlaceholder valuePlaceholder = valuePlaceholders[name]
  if (valuePlaceholder) {
    valuePlaceholder.value
  }
  else {
    delegate.@"$name"
  }
}
```

Listing 9

```
class ScriptDslifier {

  private final Script script

  public ScriptDslifier(Script script) {
    this.script = script
  }

  public void dslify(Map<String, ValuePlaceholder> valuePlaceholders) {
    script.metaClass.getProperty { String name ->
      ValuePlaceholder valuePlaceholder = valuePlaceholders[name]
      if (valuePlaceholder) {
        valuePlaceholder.value
      }
      else {
        delegate.@"$name"
      }
    }
  }
}
```

verringert wird, müssen wir etwas an der Art und Weise ändern, wie der Zugriff auf Platzhalter und deren Werte erfolgt. Bislang wurde der Zugriff nach dem Muster „Gib mir den Wert zum angegebenen Namen“ an das Binding delegiert. Um die ursprüngliche, kompakte und lesbare Form der Skripte wiederherzustellen, ändert sich das Muster zu „Gib mir den Wert des Platzhalters zum angegebenen Namen.“ Wir fügen einen Zwischenschritt in das Verfahren zum Ermitteln von Werten ein, der den Platzhalter auflöst und dessen Wert zurückliefert.

Das Binding hat in den vorangegangenen Beispielen dafür gesorgt, dass die von uns übergebenen Schlüssel-Wert-Paare behandelt wurden, als wären sie Attribute der kompilierten Groovy-Skript-Klasse. Zur Umstellung des Verfahrens benötigen wir eine Callback-Methode, die jedes Mal aufgerufen wird, wenn eine Property dieser Skript-Klasse ausgelesen werden soll. Zu diesem Zweck gibt es in Groovy die Methode *getProperty* [5]. Diese Methode überschreiben wir auf der Metaklasse des *Script*-Objekts, das von der *GroovyShell* erzeugt wird (Listing 8).

Die Methode verwendet die *Map*, in der alle Platzhalterobjekte enthalten sind, und versucht, einen passenden Platzhalter anhand des Namens zu finden. Existiert ein solcher Platzhalter, wird dessen Wert zurückgegeben, andernfalls der „normale“ Groovy-Prozess zum Auflösen von Properties aufgerufen. Wichtig ist hierbei, dass in diesem Fall vor den Namen der Property ein @-Zeichen eingefügt wird: Andernfalls wird wiederum die *getProperty*-Methode aufgerufen, und es entsteht eine Endlos-Rekursion.

Das Parsen und Erstellen des *Script*-Objekts ist in der Java-Klasse *GroovyFormula* gekapselt und müsste um das Überschreiben von *getProperty* auf der Metaklasse erweitert werden. Da Metaklassen aber nur in Groovy existieren, wird dieser Teil des Codes in eine Klasse *ScriptDslifier* (Listing 9) ausgelagert und von *GroovyFormula* (Listing 10) lediglich aufgerufen.

Regeln als Properties von Domänenobjekten

Die Umstellung des Umgangs mit Platzhaltern in DSL-Skripten ermöglicht die Verwendung von Platzhaltern, die entweder feste Werte enthalten oder deren Wert durch die Ausführung einer anderen Regel erzeugt wird. Was mit dieser Methode noch nicht funktioniert, ist, dass die Properties von Domänenobjekten durch Regeln ermittelt werden können: Das monatliche Nettoeinkommen des Antragstellers ist ja eigentlich keine einfache Property im Skript, sondern eines Domänenobjekts (in diesem Fall der Klasse *Applicant*). Damit auch die Klasse *Applicant* Properties haben kann, deren Wert nicht direkt vergeben, sondern durch die Ausführung einer Regel ermittelt wird, lässt sich hier, analog zur Klasse *Script*, im letzten Abschnitt die Methode *getProperty* auf der Metaklasse überschreiben:

```
Applicant.metaClass.getProperty { String name ->
  ValuePlaceholder valuePlaceholder = valuePlaceholders[name]
  ...
}
```

Kämen noch weitere Klassen des Domänenmodells hinzu, die diese „regelhinterlegten Properties“ unterstützen sollen, wäre für jede dieser Klassen ein entsprechender Codeschnipsel zu schreiben. Diesen redundanten Code möchten wir gerne vermeiden und lagern die Anreicherung in eine Groovy-Klasse *BusinessObjectDslifier* aus (Listing 11).

Die Verwendung des *ScriptDslifiers* erfolgt automatisch in der Klasse *GroovyFormula* – jedes Groovy-Skript ist mit dem Mechanismus zur Auflösung von Platzhaltern versehen. Ein ähnlich elegantes Verfahren für die Vorbereitung von Domänenobjekten lässt sich über den *SimpleValuePlaceholder* realisieren (Listing 12): Wird bei der Erzeugung eines solchen Platzhalterobjekts dem Konstruktor eine Platzhalter-*Map* übergeben, erfolgt eine Anreicherung des Domänenobjekts, andernfalls wird es ohne Veränderung übernommen.

Listing 10

```
public class GroovyFormula {

  final String name;
  final String code;
  private Script script;
  private final Map<String, ValuePlaceholder> valuePlaceholders;

  public GroovyFormula(String name, String code,
      CompilerConfiguration compilerConfiguration,
      Map<String, ValuePlaceholder> valuePlaceholders) {
    super();
    ...
    this.valuePlaceholders = valuePlaceholders;

    GroovyShell groovyShell = new GroovyShell(compilerConfiguration);
    script = groovyShell.parse(this.code);
  }

  public synchronized Object execute() {
    new ScriptDslifier(script).dslify(valuePlaceholders);
    return script.run();
  }
}
```

Listing 11

```
class BusinessObjectDslifier {

  private final Object businessObject;

  public BusinessObjectDslifier(Object businessObject) {
    this.businessObject = businessObject;
  }

  public void dslify(Map<String, ValuePlaceholder> valuePlaceholders) {
    businessObject.metaClass {

      getProperty { String name ->
        ValuePlaceholder vp = valuePlaceholders[name]
        if (vp) {
          vp.value
        }
        else {
          def metaProperty =
           delegate.metaClass.getMetaProperty(name)
          if (metaProperty) {
            metaProperty.getProperty(delegate)
          }
          else {
            delegate.@"$name"
          }
        }
      }
    }
  }
}
```

Performance

Groovy als Programmiersprache auf der Java Virtual Machine hängt der Ruf an, besonders langsam zu sein. Tatsächlich gilt es bei der Ausführung von Groovy-Skripten einige einfache Grundregeln zu beachten, damit die Regelausführung nicht zum Performance-Killer der Fachapplikation wird.

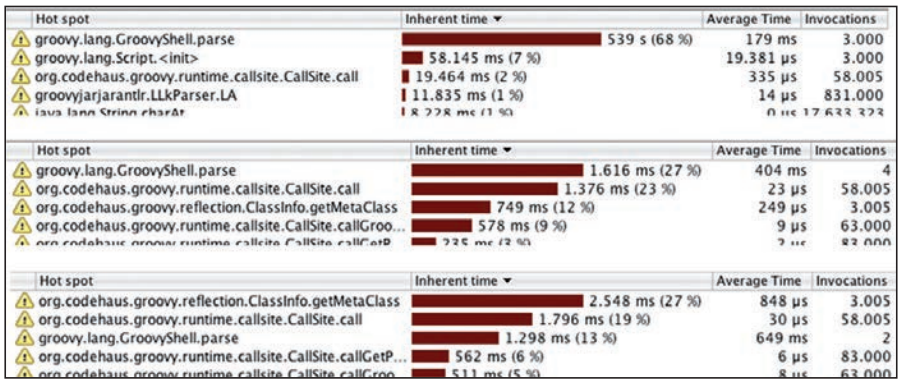
Bei der Ausführung von Groovy-Skripten ist das Parsen und Kompilieren des Skript-Strings eine sehr teure Operation, weshalb die Klasse *GroovyFormula* das *Script*-Objekt nur einmalig im Konstruktor erstellt. Zur Verdeutlichung wurde die Ausführungszeit von verschiedenen Implementierungsalternativen gemessen und verglichen. Für die Messung wurde die Regel für den monatlich frei verfügbaren Betrag ausgeführt (die wiederum die Regel für das monatliche Nettogehalt verwendet): Die obere Alternative in **Abbildung 5** zeigt die Ausführungszeit ohne eine Zwischenspeicherung dieses Objekts, die mittlere die Zeit mit einer Erstellung im Konstruktor und Zwischenspeicherung. In dieser zweiten Messung fällt auf, dass die Methode *parse* der Klasse *GroovyShell* viermal aufgerufen wurde, obwohl nur

Listing 12

```
public class SimpleValuePlaceholder implements ValuePlaceholder {

  final Object value;

  public SimpleValuePlaceholder(Object value) {
    this(value, null);
  }

  public SimpleValuePlaceholder(Object value,
      Map<String, ValuePlaceholder> valuePlaceholders) {

    super();
    this.value = value;

    if (valuePlaceholders != null) {
      new BusinessObjectDslifier(value).dslify(valuePlaceholders);
    }
  }
  ...
}
```

Listing 13

```
public synchronized Object execute() {
  if (script == null) {
    script = groovyShell.parse(this.code);
  }

  new ScriptDslifier(script).dslify(valuePlaceholders);
  return script.run();
}
```



Hot spot	Inherent time	Average Time	Invocations
groovy.lang.GroovyShell.parse	539 s (68 %)	179 ms	3.000
groovy.lang.Script.<init>	58.145 ms (7 %)	19.381 µs	3.000
org.codehaus.groovy.runtime.callsite.CallSite.call	19.464 ms (2 %)	335 µs	58.005
groovyjarjarantlr.LLkParser.LA	11.835 ms (1 %)	14 µs	831.000
java.lang.String.charAt	8.228 ms (1 %)	0 µs	17.633.323

Hot spot	Inherent time	Average Time	Invocations
groovy.lang.GroovyShell.parse	1.616 ms (27 %)	404 ms	4
org.codehaus.groovy.runtime.callsite.CallSite.call	1.376 ms (23 %)	23 µs	58.005
org.codehaus.groovy.reflection.ClassInfo.getMetaClass	749 ms (12 %)	249 µs	3.005
org.codehaus.groovy.runtime.callsite.CallSite.callGroo...	578 ms (9 %)	9 µs	63.000
org.codehaus.groovy.runtime.callsite.CallSite.callGetP...	235 ms (3 %)	2 µs	83.000

Hot spot	Inherent time	Average Time	Invocations
org.codehaus.groovy.reflection.ClassInfo.getMetaClass	2.548 ms (27 %)	848 µs	3.005
org.codehaus.groovy.runtime.callsite.CallSite.call	1.796 ms (19 %)	30 µs	58.005
groovy.lang.GroovyShell.parse	1.298 ms (13 %)	649 ms	2
org.codehaus.groovy.runtime.callsite.CallSite.callGetP...	562 ms (6 %)	6 µs	83.000
org.codehaus.groovy.runtime.callsite.CallSite.callGroo...	511 ms (5 %)	8 µs	63.000

Abb. 5: Messungen der DSL-Performance

zwei Regeln ausgeführt worden sind. Durch die Erzeugung des *Script*-Objekts im Konstruktor von *GroovyFormula* wurden alle DSL-Skripte übersetzt und nicht nur die relevanten. In unserem Beispiel ändern wir *GroovyFormula* so ab, dass die Erstellung *lazy* erfolgt (Listing 13).

Die Ausführungszeit dieser Implementierungsalternative ist in der unteren Messung in **Abbildung 5** abzulesen. Wie erwartet, erfolgen jetzt auch nur noch zwei Aufrufe von *GroovyShell.parse*. Dass die Methode *GroovyFormula.execute synchronized* ist, erschwert die Parallelisierung des Codes. Eine Umstellung an dieser Stelle sorgt ebenfalls für eine Optimierung der Laufzeiteigenschaften. Weiteres Optimierungspotenzial für die Performance bietet mit Sicherheit die Implementierung der *Dslifier*-Klassen. Darüber hinaus ist es durch die Auflösung von Platzhaltern durch die *getProperty*-Methode möglich, auch Regelergebnisse in einem Cache zu halten.

Fazit

In vielen Unternehmen existieren heute Java-(Enterprise-)Fachapplikationen, die durch eine mächtige, flexible Konfiguration in Form von DSL-Regeln besser an die Bedürfnisse der verwendeten Fachabteilungen angepasst werden können. In diesem Artikel stand deshalb die Integration von DSL-Skripten in eine Java-Fachapplikation im Vordergrund. Diese fiktive Fachapplikation wurde beispielhaft durch einen in Java geschriebenen JUnit-Testfall realisiert. Um die Java- und die Groovy-Klassen in einem gemeinsamen Build zu übersetzen, findet im Beispiel der Eclipse Compiler als Maven Plug-in Verwendung. Der weitere Artikel zeigte, wie ein bestehendes Domänenmodell mit den DSL-Regeln verknüpft sowie Properties und Regeln automatisch aufgelöst und an Domänenobjekte angereichert werden können. Welche Punkte im Hinblick auf die Performance bei der Regelausführung berücksichtigt werden sollten, wurde ebenfalls beleuchtet. Die Quellcodes zum Artikel stehen in Form eines Maven-Projekts auf javamagazin.de zum Download bereit. Die vorgestellte Integration stellt eine gute Möglichkeit dar, die Flexibilität und Kompaktheit von Groovy in einer Enterprise-Anwendung zu nutzen, ohne die gesamte Anwendung auf Groovy portieren zu müssen.

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 Methodiken wie Scrum oder Kanban.

Links & Literatur

[1] Erich Gamma, et al.: „Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software“, Addison-Wesley; 2001

[2] http://groovy.codehaus.org/ExpandoMetaClass

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

[4] http://groovy.codehaus.org/Operator+Overloading

[5] http://groovy.codehaus.org/Using+invokeMethod+and+getProperty