JAVA Mag inkl. CD

Business Process Modeling Notation »61

S&S

Deutschland €7,50 Österreich €8,60 Schweiz sFr 15,80 12.2009

Javamagazin™

Java • Architekturen • SOA • Agile

www.javamagazin.de

CD-INHALT

DSL Best Practices
Session von der JAX 2009 in voller Länge

DIE HIGHLIGHTS

Klaros Test-management

FitGoodies

Sonar

WEITERE INHALTE

JBoss Portal
eXo Portal
Code-Review-Tools
JSFUnit
HTMLUnit

Alle CD-Infos ab Seite 3

Datenträger enthält Info- und Lehrprogramme gemäß §14 JuSchG

Open-Source-Portalserver
Liferay, eXo, JBoss Portal »53

Kampf dem Fehlerteufel
FindBugs, PMD und Checkstyle im Vergleich »74

Exklusiv für Abonnenten
Spring-Poster

Domain Specific Languages

- Was sind Domänenspezifische Sprachen? »14
- Do-it-yourself-DSLs in Scala »22
- Domänenspezifische Prozesssprachen mit Groovy »28

NetBeans Profiler

Wenn's mal wieder länger dauert »66

Agile Projektteams

Team-Entwicklung und -Design »100

Validierung mit JBoss Seam

The input Seams valid to me »39

Freiheit für die Fachabteilung

Trotz zahlreicher Versuche ist es immer noch nicht gelungen, die bereits legendäre Kluft zwischen den Fachabteilungen und der IT zu schließen oder zumindest signifikant zu verkleinern. Domänenspezifische Sprachen haben das Potenzial, uns diesem Ziel ein Stück näher zu bringen. Groovy bietet dazu vielfältige Möglichkeiten, die in diesem Artikel exemplarisch erläutert werden.

von Wolfgang Pleus

Der typische Entwickler ist täglich umgeben von domänenspezifischen Sprachen (DSLs). Tatsächlich lassen sich typische Programmieraufgaben nur durch DSLs wie SQL oder reguläre Ausdrücke lösen. Zwar sind diese spezifischen Sprachen in ihren Möglichkeiten begrenzt, dafür aber ideal auf die Lösung eines bestimmten Problems ausgerichtet.

Wozu das Ganze?

Bei der Realisierung heutiger IT-Systeme und -Architekturen geht es fast immer um Effizienzsteigerung und Automatisierung. Serviceorientierte Architekturen (SOA), Business Process Management (BPM) und Business Rules Management (BRM) zielen vor allem darauf ab, die hohen Potenziale im Bereich von Unternehmensanwendungen auszunutzen. Eine Schlüsselrolle kommt dabei der Integration von nichttechnisch ausgerichteten Personen zu. Martin Fowler bezeichnet diese auch als Laienprogrammierer [1]. Ein Problembereich ist oft die traditionell starke Trennung von Fachabteilungen und IT. Durch die Einbeziehung der fachlichen Experten in den Entwicklungsprozess über die Anforderungsdefinition und Abnahme hinaus, können erhebliche Potenziale freigesetzt werden, die die Agilität eines Unternehmens nachhaltig steigern können. Initiativen wie 360° SOA/BPM beispielsweise von Oracle oder der Software AG zielen in diese Richtung. Domänenspezifische Sprachen können in diesem Zusammenhang ebenfalls eine wichtige Rolle spielen.

Klassifizierungen

Im Gegensatz zu DSLs werden Programmiersprachen wie Java, C# oder Groovy als General Purpose Languages (GPL) oder auch Hauptsprache bezeichnet, mit denen sich grundsätzlich alle Arten von Programmieraufgaben bewältigen lassen. Domänenspezifische Sprachen werden unterschieden in extern und intern (siehe S.14). Bei der in diesem Artikel vorgestellten DSLs handelt es sich um interne DSLs mit einer fachlichen Ausrichtung.

Prozesssprachen und DSLs

Prozessorientierung wird in der Softwareentwicklung immer relevanter und ist ein wichtiger Baustein serviceorientierter Architekturen. Prozesssprachen haben naturgemäß eine große Nähe zu den Fachabteilungen und stellen eine Art Schnittstelle zwischen diesen und der IT dar. Aus diesem Grund ist es insbesondere an dieser Stelle lohnend, domänenspezifische Sprachen einzusetzen.

Aktuelle Prozesssprachen wie BPEL oder Microsoft XLANG verfügen über grafische Editoren. Wohl nicht zuletzt wegen dieser grafischen Darstellung wird die Ausrichtung dieser Prozesssprachen häufig missverstanden. Es wird eine fachliche Ausrichtung angenommen, da grafische Modelle auf den ersten Blick nachvollziehbar erscheinen. Die Marketingversprechen einiger Anbieter verstärken diesen Eindruck noch. Grafische Repräsentationen sind textuellen jedoch nicht grundsätzlich überlegen, da sie insbesondere bei höherer Komplexität schlecht nachvollziehbar sein

Merkmal	Beschreibung
Relevant	Sie muss einen klaren fachlichen Nutzen haben
Erlernbar	Sie muss intuitiv erlernbar sein
Verbergend	Sie muss Komplexität weitgehend verbergen (Convention over Configuration)
Produktiv	Sie muss schnell zu Ergebnissen führen

Tabelle 1: Qualitätsmerkmale fachlicher DSLs

Quellcode auf CD

können. Prinzipiell handelt es sich um technische DSLs, die es ermöglichen, Prozessdefinitionen über eine Prozessmaschine auszuführen. Die wahre Stärke liegt in den Prozessmaschinen, mit denen langlaufende Abläufe zuverlässig verarbeitet werden können. Die Sprachen als Schnittstelle zu den Anwendern sind aufgrund ihrer technischen Ausrichtung jedoch bestenfalls in der Lage, die Kluft zwischen Fachabteilungen und IT zu verkleinern, jedoch schließen können sie diese nicht.

Um die Kluft deutlich zu verkleinern, können konsequent fachlich ausgerichtete, domänenspezifische Sprachen eingesetzt werden. Diese Sprachen bilden die Begrifflichkeiten der Domänenspezialisten möglichst genau ab und nähern sich dabei so weit wie möglich den Benutzern an. Sie verbergen weitgehend das Standardverhalten der Domäne und werden dadurch einfach in der Anwendung. Dieser Ansatz wird auch als Convention over Configuration bezeichnet [2]. Wichtige Qualitätsmerkmale einer fachlich orientierten DSL sind in Tabelle 1 aufgelistet.

Es ist sehr wichtig, dass der Realisierungsaufwand für eine DSL gering ist und Anpassungen flexibel erfolgen können, damit der Entwicklungsaufwand in Relation zum Nutzen steht. Die Praxis zeigt, dass durch die Einbeziehung der fachlichen Ressourcen der Produktivitätszuwachs beträchtlich ist. Die Entwicklung einer DSL zahlt sich schon bei mittelgroßen Projekten aus.

Ein erster Entwurf

Das wesentliche Merkmal einer DSL ist, dass sie sich konsequent an der fachlichen

Eigenschaften

Wenn eine Eigenschaft einer Klasse ohne Sichtbarkeitsmodifizierer definiert wird, generiert Groovy automatisch ein privates Feld und öffentliche *getter*- und *setter*-Methoden. Diese können bei Bedarf überschrieben werden. Dadurch wird eine intuitive Verwendung von Klasseneigenschaften ohne *get*-und *set*-Präfixe mit gleichzeitiger Steuerung der Sichtbarkeit möglich. Wenn eine Eigenschaft ohne Präfix verwendet wird, so wird die *getter*-Methode aufgerufen. Das funktioniert auch, wenn keine Eigenschaft existiert:

```
class Verbs{
 String name = "Hans"
 public boolean getIstBestandskunde(){
  return true
 }
}

def verbs = new Verbs()
println verbs.getName() // generiert
println verbs.name // Aufruf ohne Präfix
println verbs.istBestandskunde // Aufruf ohne Präfix, keine Eigenschaft
```

Domäne ausrichtet. Technische Aspekte haben sich dieser Tatsache unterzuordnen. Um das zu gewährleisten, müssen die Fachexperten kontinuierlich und von Anfang an in die Sprachdefinition einbezogen werden. Ein Vorgehen nach Scrum eignet sich dazu sehr gut, da es die Fachseite in der Product-Owner-Rolle aktiv einbezieht. Wie natürliche Sprachen bestehen auch domänenspezifische Sprachen aus Subjekt, Verb und Objekt. Die über die DSL aufgerufenen Services stellen gewissermaßen die Subjekte dar. Verben und Objekte sind Teil der DSL-Definition. Sie beginnt mit der Bestimmung der fachlich relevanten Objekte. Die konkrete Ausprägung kann naturgemäß in jeder Domäne unterschiedlich sein. Hier sind unterschiedliche Abstraktionen von Streams bis zu konkreten Fachentitäten denkbar. In unserem Beispiel beginnen wir mit einem Domänenmodell aus dem Versicherungsumfeld. Das Modell verfügt über die Entitäten *Antrag* (Application), *Police* (Policy) und *Versicherungsnehmer* (Insuree).

Bei der Analyse hat sich jedoch ergeben, dass diese technische Sicht für die Fachanwender nicht adäquat ist. Diese Benutzer sind nur daran interessiert, ob ein Antragssteller Bestandskunde ist und ob er oder sie versichert werden kann. Daraus leiten sich die Eigenschaften *istBestandskunde* und *istVersicherungsfaehig* ab. Alle anderen Informationen werden von der DSL verborgen.

Um einen Versicherungsprozess abzuschließen, werden die Verben *berechneBeitrag*, *erstellePolice* und delegiere von der DSL bereitgestellt. Zur Ablaufsteuerung werden zudem die Kontrollanweisungen *falls*, *fallsNicht* und *sonst* verwendet, da einfache Wenn-/Dann-Konstruktionen und -Schleifen für die Fachanwender in der Regel problemlos anzuwenden sind. Es fällt auf, das die domänenspezifische Sprache in der Umgangssprache der Fachanwender formuliert ist, auch wenn das technische Modell in einer anderen Sprache modelliert ist. Das ist sehr typisch für eine fachliche DSL, da sie nur so intuitiv und leicht erlernbar ist.

Um dem Umfang des Artikels gerecht zu werden, ist der Umfang der DSL natürlich sehr stark vereinfacht. Ein vollständiges Beispiel zeigt Listing 1.

Während der Analyse könnte sich herausstellen, dass ein Antragsprozess, wenn er nicht durch eine Delegation unterbrochen wird, immer mit einer Police endet. In diesem Fall könnte der Prozess wie in Listing 2 vereinfacht werden. Die Policenerstellung wird dabei gemäß dem Convention-over-Configuration-Paradigma [2] zum integralen Bestandteil der DSL. An diesem Beispiel wird deutlich, wie sehr die Ausprägung einer domänenspezifischen Sprache von ihrem Kontext abhängig ist.

Listing 1: Insurance Sales Language

```
prozess (start: "neuer antrag") {
 berechneBeitrag tarif:1.23
  fallsNicht (istBestandskunde) {
   falls (istVersicherungsfaehig) {
    erstellePolice kopien: "VN, VP, BZ"
   }
   sonst{
    delegiere an: "teamleiter"
   }
  }
 sonst{
  erstellePolice kopien: "VN"
 }
}
```

Listing 2: Vereinfachter Prozess

```
prozess (start: "neuer antrag") {
 berechneBeitrag tarif:1.23
  fallsNicht (istBestandskunde) {
   fallsNicht (istVersicherungsfaehig) {
    delegiere an: "teamleiter"
   }
  }
}
```

Listing 3: Script-Erweiterung über ExpandoMetaClass

```
static void runDSL(def rawScript, Context ctx) {
  def shell = new GroovyShell()
  Script script = shell.parse(rawScript)

  // Script-Klasse erweitern
  ExpandoMetaClass emc = new ExpandoMetaClass(script.class, false)
 emc.prozess = {Map params, Closure cl ->
   cl.delegate = new Verbs(ctx)
   cl.resolveStrategy = Closure.OWNER_FIRST
   cl()
 }

 emc.initialize()
 script.metaClass = emc

 script.run() // Script ausführen
}
```

Listing 4: berechneBeitrag-Verb

```
public void berechneBeitrag(Map params){
  def tarif = params==null?2.0d:params['tarif']
  ctx.state.policy.premium = tarif * 200
  println "Beitrag ist ${ctx.state.policy.premium} Euro"
 }
```

Der Rahmen

Groovy [3] stellt eine ideale Möglichkeit zur Realisierung von domänenspezifischen Sprachen dar. Aufgrund der nahtlosen Java-Integration können alle Merkmale der Plattform genutzt werden. Darüber hinaus bietet Groovy leistungsfähige Merkmale zur Dynamisierung, die für die einfache Erstellung von DSLs erforderlich sind. Dazu gehören u. a. benannte Parameter, Eigenschaften, Closures, Metaklassen, Interzeptoren und Zugriff auf den abstrakten Syntaxbaum (AST). Eine formale Sprachnotation in der Backus-Naur-Form (BNF) oder der Einsatz von Parser-Generatoren ist nicht erforderlich. Bei dem Prozess aus Listing 1 handelt es sich grundsätzlich um ein Groovy Script, das über die *GroovyShell* ausgeführt werden kann. Die DSL-Verben gehören jedoch noch nicht zum Sprachumfang von Groovy. Jede Groovy-Klasse kann über eine *ExpandoMetaClass* erweitert werden (Kasten:

Metaklassen

Über Metaklassen kann jede Groovy-Klasse erweitert werden. So lassen sich Methoden zur Laufzeit zu einer Klasse hinzufügen:

```
class AnyClass{}
def any = new AnyClass()

def emc = new ExpandoMetaClass
                    (any.class,false)
emc.prozess = {Map params-> println
              "Hi ${params['name']}"}
emc.initialize()
any.metaClass = emc

any.prozess(name:"Max")
                    // Ausgabe -> Hi Max
```

„Metaklassen"). Dieses Merkmal wird genutzt, um die Methode *prozess* zur *Script*-Klasse hinzuzufügen (Listing 3).

Die Methode *prozess* erwartet eine beliebige Parameterliste sowie ein Closure als Übergabeparameter (Kasten: „Closures und Delegates"). Dadurch wird die Struktur *prozess(start:"neuer antrag"){...}* unterstützt.

Verben

Grundsätzlich könnten auf diese Art alle DSL-Verben dem Script hinzugefügt werden. Da der *prozess*-Methode ein Closure übergeben wurde, gibt es einen einfacheren Weg. Jedes Closure verfügt über einen Delegaten. Wenn eine Methode innerhalb des Closures aufgerufen wird, wird versucht, diesen Aufruf über den Delegaten zu verarbeiten. In Listing 3 wird die Klasse *Verbs* als Delegate zu-

Closures und Delegates

Ein Closure ist ein explizit definierbarer Codeblock:

```
// Einfacher Closure
Closure c = {println "Done"}
c() // Ausgabe -> Done
```

Dieser Block kann einer Variable vom Typ *Closure* zugewiesen und somit als Übergabeparameter für Methoden verwendet werden:

```
// Methode mit Closure als Parameter
def falls(boolean condition, Closure c){
 if(condition) c()
}
// Methodenaufruf mit Closure
falls(true){
 println "Done"
} // Ausgabe -> Done
```

Closures können durch Delegates erweitert werden. Ein Methodenaufruf innerhalb des Closure wird an den Delegaten weitergeleitet:

```
// Delegate-Klasse
class Delegate{
 def doIt(){
  println "Done"
 }
}
// Closure mit Delegate
Closure cl = {doIt()}
cl.delegate = new Delegate()
cl() // Ausgabe -> Done
```

Benannte Parameter

Einer Groovy-Methode können benannte Parameter in beliebiger Reihenfolge als Map übergeben werden. Durch die Implementierung von sinnvollen Standardwerten werden Parameter optional. So werden verschiedene Aufrufvarianten ermöglicht:

```
def berechneBeitrag(params){
 def tarif = params==null?2.0:params['tarif']
 println "Tarif: ${tarif}"
}

berechneBeitrag(tarif:1.23)
berechneBeitrag tarif:1.23
berechneBeitrag()
```

Abstrakter Syntaxbaum (AST)

Jedes Groovy-Programm wird während der Kompilierung in einen abstrakten Syntaxbaum (AST) überführt. Groovy erlaubt die Manipulation des AST durch ein entsprechendes API. Beispielsweise können Methodenaufrufe umbenannt, ersetzt oder entfernt werden. Dadurch ergeben sich sehr weitreichende Möglichkeiten der Manipulation bereits zum Zeitpunkt der Kompilierung. Zusammen mit Metaklassen und Interzeptoren können Groovy-Implementierungen so über den ganzen Lebenszyklus verändert werden.

Interzeptoren

Groovy erlaubt es, alle Methodenaufrufe und Eigenschaftenzugriffe abzufangen. Das wird erreicht durch die Implementierung der Schnittstelle *GroovyInterceptable* und Überschreiben von *invokeMethod*, *getProperty* und *setProperty*:

```
import org.codehaus.groovy.runtime.InvokerHelper
class AnyClass implements GroovyInterceptable{
  def props = [:]
  def invokeMethod(String name, args){
    System.out.println "Invoking method $name."
    def metaClass = InvokerHelper.getMetaClass(this)
    return metaClass.invokeMethod(this,name,args)
  }
  def getProperty(String name){
    System.out.println "Getting property $name."
    return props[name]
  }
  void setProperty(String name,value){
    System.out.println "Setting property $name."
    props[name]=value
  }
  void doIt(){
   System.out.println "Method doIt called."
  }
}

def any = new AnyClass()
any.doIt() // Ausgabe -> Invoking method doIt. Method doIt called
any.x=5 // Ausgabe -> Setting property x.
assert any.x==5 // Ausgabe -> Getting property x.
```

Um eine Rekursion zu vermeiden, wird *System.out.println* anstelle von *println* verwendet. Zudem wird die initiale Methode innerhalb von *invokeMethod* über die Metaklasse aufgerufen.

gewiesen. Alle weiteren Verben werden innerhalb dieser Klasse implementiert.

Der Zustand des Prozesses wird vom Prozess verborgen. Verwaltet wird er von der Klasse *Context*, die vor der Ausführung mit dem Anfangszustand (Application und Policy) initialisiert wird. Nach der Prozessausführung kann der Endzustand aus dem Kontext ausgelesen werden. Listing 4 zeigt eine exemplarische Implementierung des *berechneBeitrag*-Verbs.

Das Verb greift auf den Kontext und die übergebenen Parameter (Kasten: „Benannte Parameter") zu, um die Operation auszuführen. Die Ergebnisse werden wiederum im Kontext gespeichert. Verben zur Ablaufsteuerungen benötigen zudem ein Closure als Parameter. Listing 5 zeigt das am Beispiel des *falls*-Verbs.

Hier wird das Closure in Abhängigkeit der Bedingung ausgeführt. Da das *falls*-Verb mit dem *sonst*-Verb zusammenarbeitet, muss der Zustand der Operation im Kontext zwischengespeichert werden. Andere Konstrukte wie Schleifen können auf diese Art ebenfalls realisiert werden. Eigenschaften wie *istBestandskunde* werden als Groovy-Eigenschaften realisiert (Kasten: „Eigenschaften"). Dadurch wird eine intuitive Syntax ohne *get*- und *set*-Präfixe erreicht.

Aspekte

Jede Domäne hat spezifische Aspekte, die immer ähnlich ablaufen. Dazu gehören die Prozessverfolgung (Tracking) oder die Fehlerbehandlung. Idealerweise sollten diese Aspekte von den Benutzern der DSL verborgen sein. Groovy erlaubt es, beliebige Methodenaufrufe abzufangen (Kasten: „Interzeptoren"). So lassen sich Aspekte sehr einfach realisieren. Listing 6 zeigt eine Interceptor-Implementierung, die anstelle der Verbenklasse als Closure Delegate registriert werden kann.

Dieser Interceptor erlaubt eine zentrale Fehlerbehandlung sowie die Messung der Ausführungszeit einzelner Verben. Durch die Implementierung allgemeiner Aspekte lässt sich die Komplexität der DSL weiter verringern.

Beschränkungen

Wie schon erwähnt, stellt die DSL eigentlich ein Groovy Script dar. Die DSL kann also um Groovy-Befehle erweitert werden. Für manche mag das ein Vorteil sein. Falls die DSL zu limitiert sein sollte, lässt sich das Ziel schließlich durch ein wenig Groovy erreichen. Andererseits führt das mittelfristig eher zu schlecht wartbaren DSL-Scripten. Besser ist es, die DSL iterativ um neue Merkmale zu erweitern, falls sie sich als zu limitiert erweisen sollte. Wie lässt sich aber verhindern, dass der geübte DSL-Anwender Groovy-Befehle verwendet? Zu diesem Zweck muss der Sprachumfang der Hautsprache (in diesem Fall Java und Groovy) eingeschränkt werden. Sehr einfach geht das, indem analog der *prozess*-Methode weitere Methoden der

ExpandoMetaClass überschrieben werden (Listing 7).

Diese Prüfungen werden erst zur Laufzeit sichtbar, da die *println*-Methode erst aufgerufen werden muss. Seit Version 1.6.1 verfügt Groovy zudem über Möglichkeiten des Zugriffs auf den abstrakten Syntaxbaum (Kasten: „Abstrakter Syntaxbaum"). Dadurch sind ebenfalls Prüfungen zur Kompilierungszeit möglich. Durch die Realisierung einer *Visitor*-Klasse können alle Methodenaufrufe geprüft werden. Listing 8 zeigt eine exemplarische Implementierung. Diese wirft eine Ausnahme, falls entweder *println* oder *exit* im DSL-Script verwendet werden. So wird beispielsweise ein Aufruf von *System.exit(0)* verhindert. Dieser Visitor erfordert noch Hilfscode (siehe Heft-CD).

Listing 5: falls-Verb

```
public void falls(boolean condition, Closure closure) {
 ctx.lastCondition = condition
 if(condition){
   closure.delegate = new Verbs(ctx.copy())
   closure.resolveStrategy = Closure.DELEGATE_FIRST
   closure()
 }
}
```

Listing 6: Interceptor für Aspektimplementierungen

```
public class VerbsInterceptor implements GroovyInterceptable{
  private Verbs verbs
  VerbsInterceptor(Verbs verbs) {
    this.verbs = verbs
  }
  def invokeMethod(String name, args){
    def ret
    def start = System.currentTimeMillis()
    try{
      ret = verbs.metaClass.invokeMethod(verbs,name,args)
    }
    catch(Throwable t){
      System.out.println "Logging error: ${t.getMessage()}"
    }
    def stop = System.currentTimeMillis()
    System.out.println "Tracking ${name} execution time: ${stop-start} ms"
    return ret
  }
}
```

Listing 7: Einschränkungen durch ExpandoMetaClass

```
static void addRuntimeConstraints(ExpandoMetaClass emc){
  emc.println = {String msg ->
    throw new Exception("println is not allowed in ISL.")
}
```

Listing 8: Einschränkungen durch AST-Zugriff

```
public class ConstraintVisitor extends CodeVisitorSupport {
  def constraintVerbs =['exit','println']
 void visitMethodCallExpression(MethodCallExpression expression) {
   ConstantExpression method = expression.getMethod()
   if(constraintVerbs.contains(method.getValue())){
    throw new Exception("${method.getValue()} is not allowed");
   }
   super.visitMethodCallExpression(expression)
 }
}
```

Ausblick

Eine DSL, wie sie bisher vorgestellt wurde, kann u. a. zur Prozessabbildung im Rahmen einer serviceorientierten Architektur eingesetzt werden. Dabei sind weitere Aspekte wichtig, insbesondere die Verben, z. B. *berechneBeitrag* sind direkt in der Prozess-DSL implementiert. In einer echten Unternehmensumgebung würden diese Aufrufe an Serviceimplementierungen delegiert. Diese Services könnten zur Laufzeit ermittelt und z. B. über einen ESB angebunden oder über OSGI oder Spring injiziert werden.

Eigenschaften, z. B. *istVersicherungsfaehig*, sind häufigen Änderungen unterworfen. Daher wären sie gute Kandidaten für die Abbildung über eine Business Rules Engine. Der Beispielprozess wird manuell aktiviert, denkbar wäre auch eine Aktivierung über eine Nachrichten-Subscription einer JMS-Queue oder über einen Scheduler. Eine effiziente Ressourcenverwaltung kann durch einen Zustandsdienst erreicht werden, der den Prozesskontext verwaltet. Unterstützung für langlaufende Prozesse wird über Persistenzpunkte ermöglicht, die als weitere Aspekte implementiert werden können. Bewährt hat sich auch die Bereitstellung einer Workbench, zur Unterstützung der DSL-Anwender. Diese reichen vom Syntax-Coloring einfacher Texteditoren, bis zu Eclipse-Plug-ins zur vollständigen Integration in einen Freigabeprozess. Grafische Repräsentationen über BPMN oder UML-Profile sind ebenfalls denkbar.

Jenseits der Technik

Unabhängig von den technischen Details empfiehlt es sich die organisatorischen Implikationen im Blick zu behalten. Durch die konsequente Einbeziehung der Fachabteilungen verändert sich das Selbstverständnis sowohl in den Fachabteilungen als auch in der IT. Die prozesszentrische Sicht führt zu neuen und veränderten Rollen. Die Fachabteilungen werden als Laienprogrammierer in die Entwicklung einbezogen, während die IT sich stärker auf Serviceentwicklung und Infrastrukturthemen fokussiert. Das kann Unsicherheit und Spannungen erzeugen – übrigens nicht nur in den Fachabteilungen. Um die aktive Mitarbeit aller Beteiligten zu gewährleisten, ist es daher essenziell, ihnen die Prozessstrategie und mögliche Auswirkungen auf die tägliche Arbeit zu erklären. Das ist eine Managementaufgabe. Daher ist es wichtig, das Management als starken Partner zu haben und frühzeitig Einigkeit in Bezug auf die gemeinsame Strategie herzustellen.

Fazit

DSLs gewinnen insbesondere im Bereich der Prozessautomatisierung vermehrt an Bedeutung. Insbesondere Groovy stellt aufgrund seiner dynamischen Merkmale eine ideale Implementierungstechnologie dazu dar. Eine besondere Bedeutung kommt der engen Zusammenarbeit mit den Fachabteilungen während des DSL-Designs zu. Das Ergebnis sind fachlich ausgerichtete Sprachen, die dazu beitragen können, die Kluft zwischen Fachabteilungen und IT deutlich zu verkleinern. ■

Wolfgang Pleus arbeitet als Technologieberater, Autor und Trainer im Bereich serviceorientierter Architekturen. Seit über 15 Jahren unterstützt er internationale Unternehmen bei der Realisierung komplexer Geschäftslösungen auf der Basis von Java EE und .NET. Er ist Mitglied im Accelsis SOA/BPM Competence Team.

Links & Literatur

[1] Lay Programmers: http://martinfowler.com/articles/languageWorkbench.html#InvolvingNon-programmers

[2] Convention over Configuration: http://en.wikipedia.org/wiki/Convention_over_configuration

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