Mit Stellenmarkt

iX DEVELOPER

4 192045 312906 01

€ 12,90
Österreich € 14,20
Schweiz CHF 25,80
Benelux € 14,80
Italien € 16,80

1/2012

Auf der Heft-DVD:

Über 8 GByte Software für Entwickler

Intel Parallel Studio XE 2011

Visual Studio 2010 Ultimate Edition

Eclipse 3.7.1 für Java-Entwickler

Tasktop Dev 2.1.0

Kaazing WebSocket Gateway

31 Episoden des SoftwareArchitekTOUR-Podcasts

Sponsored Software:
Intel Corp., h. o. Computer

Programmieren heute

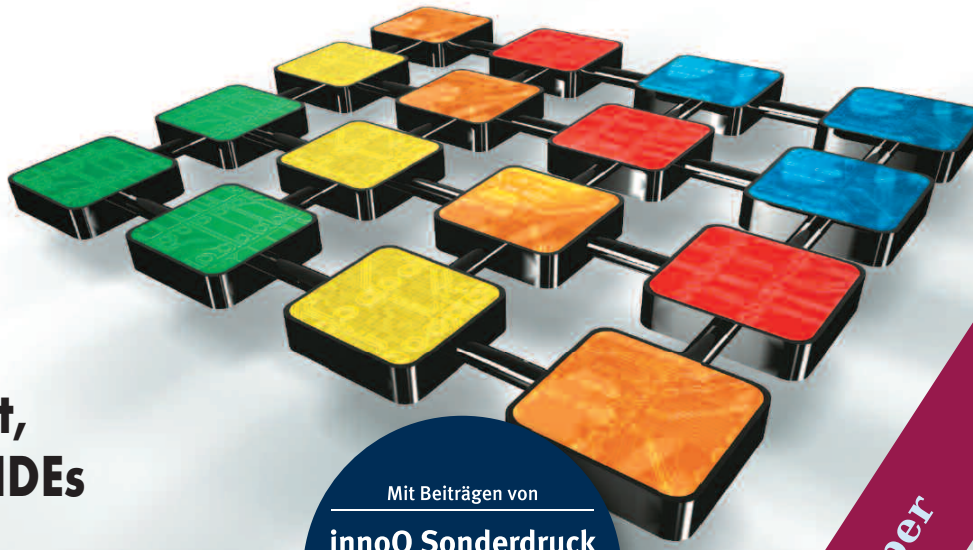
Gewinnspiel:
Notebook zu gewinnen

Aktoren, Dataflow, MapReduce, STM & Co.:

Konzepte paralleler Programmierung, Multithreading bei Java, .Net und C/C++

Agile Softwareentwicklung & ALM:

Continuous Delivery, DVCS, Build, Testing, DevOps, Kanban



Zeitgemäße Webentwicklung:

Serverside JavaScript, HTML5 und Browser-IDEs

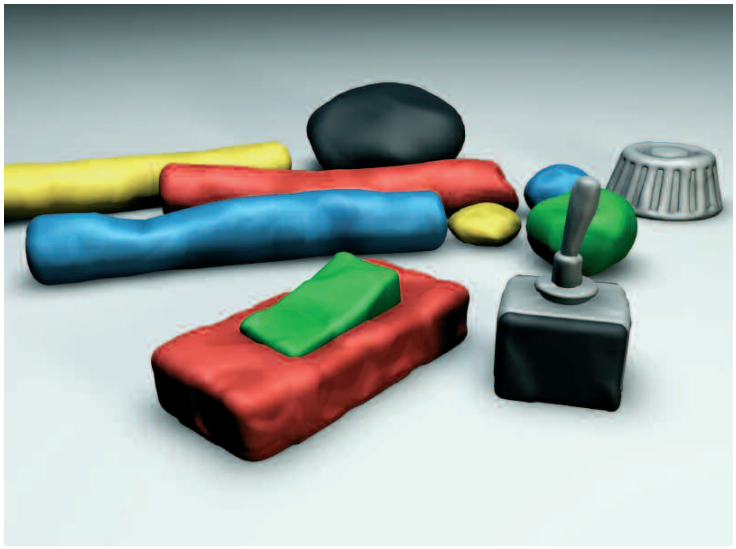
Mit Beiträgen von
innoQ Sonderdruck
Stefan Tilkov
innoQ

Datenträger enthält **Info- und Lehrprogramme** gemäß § 14 JuSchG

IT-Hypes unter der Lupe:

Mobile Web
Platform as a Service
Programmiersprachen-Trends

präsentiert von: heise Developer www.heise-developer.de



Interne DSLs: Programmiersprachen in Programmiersprachen einbetten

Formbar

Stefan Tilkov

In den letzten Jahren hat es sich eingebürgert, bei domänenspezifischen Sprachen zwei Varianten zu unterscheiden: interne und externe. Dabei gewinnen interne DSLs insbesondere durch die steigende Popularität dynamisch typisierter Sprachen zunehmend an Bedeutung.

Obwohl es zwischen gängigen Programmiersprachen wie C++, Java, C#, Python oder Perl in der Mächtigkeit, den unterstützten Programmierparadigmen, im Laufzeit- und Entwicklungsverhalten gewaltige Unterschiede gibt, haben sie eines gemeinsam: Sie sind – zumindest prinzipiell – als sogenannte General-Purpose-Sprache allgemeingültig für nahezu beliebige Zwecke einsetzbar. Im Gegensatz dazu sind domänenspezifische Sprachen (Domain Specific Languages, DSLs) auf einen einzelnen Bereich, eine Domäne, zugeschnitten.

Eine solche Domäne kann fachlich sein, wie die Definition von Produkten in einem Versicherungssystem, oder technisch, wie die Beschreibung von Strukturen einer grafischen Benutzeroberfläche. Diese Unterscheidung zeigt bereits, dass es eine beliebige Menge solcher Sprachen entlang des gesamten Spektrums – von spezifischen, für einen einzigen Anwendungsfall erstellten bis hin zu generischen, in vielen verschiedenen Projekten einsetzbaren – geben kann. Wichtig ist in jedem Fall, dass die Sprache es vereinfacht, die domänenspezifischen Konzepte auszudrücken, weil sie genau dafür erarbeitet wurde.

In den meisten Fällen sind DSLs zwar spezifisch, aber in ihrer Mächtigkeit eingeschränkt: Häufig enthalten sie nicht alle Kontrollstrukturen, verzichten auf Möglichkeiten zur Definition eigener Datentypen oder machen andere Abstriche gegenüber dem, was Entwickler aus „normalen“ Programmiersprachen gewohnt sind.

Kurz und knapp

Wenn sie weniger mächtig sind, was spricht dann überhaupt für das Erstellen einer DSL? Der Hauptvorteil besteht darin, dass die spezifische Sprache es besser ermöglicht, Sachverhalte einfach, knapp und präzise auszudrücken, als es mit dem generischen Mechanismus denkbar wäre. Je spezifischer die DSL ist, desto geringer ist die Distanz zwischen dem, was der Entwickler ausdrücken will, und dem, was er dazu hinschreiben muss. Im Idealfall liest sich das Programm wie eine Beschreibung des Problems beziehungsweise der Aufgabenlösung, wie man sie im Protokoll einer Diskussion mit einem Anwender beschreiben würde.

Es lässt sich trefflich darüber streiten, ob Fachexperten ein Programm in einer für ihre Domäne entwickelten DSL sogar selber schreiben könnten. Nach Meinung des Autors ist das praktisch nie der Fall. Aber zumindest die Qualitätskontrolle durch den Fachexperten sollte einfacher werden, wenn er das Programm lesen kann, weil es in seiner eigenen Sprache ähnlichen Form notiert ist.

Intern versus extern

Für die Erstellung einer DSL stehen dem Entwickler zwei grundsätzliche Mittel zur Verfügung. Das erste davon ist die Definition und Implementierung einer Sprache mit eigener Syntax und Semantik. Die notwendigen Komponenten für die Verarbeitung von Programmen in dieser Sprache – Validierer, Interpreter, Compiler, Parser und so weiter – werden dabei in der Regel mit den gleichen Verfahren und Werkzeugen umgesetzt, die auch bei der Entwicklung allgemeingültiger Sprachen zum Einsatz kommen (wie lexx/yacc, flex/bison, ANTLR oder Xtext). In dem Fall spricht man von **externen DSLs** und meint damit, dass für die Verarbeitung der DSL Mechanismen zum Einsatz kommen, die aus Sicht der eingesetzten Hauptprogrammiersprache extern sind.

Im Gegensatz dazu steht die zweite Variante, die Realisierung einer **internen DSL**. Hier wird kein eigener Compiler oder Interpreter entwickelt. Stattdessen nutzt man die dafür in der ohnehin eingesetzten Programmiersprache enthaltenen Mechanismen – die DSL wird in eine Wirtssprache eingebettet, bleibt also intern. Programme, die in einer internen DSL ausgedrückt sind, sind immer auch gültige Programme in der Wirtssprache.

Listing 1 und 2 zeigen dasselbe Beispiel, die Beschreibung eines finiten Automaten (State Machine), bei Listing 1 umgesetzt mit einer hypothetischen externen DSL, bei Listing 2 eingebettet in die Programmiersprache Ruby, die hier stellvertretend für aktuelle dynamische Sprachen steht. In-

formationsgehalt und Semantik sind in beiden Listings gleich, aber die Syntax unterscheidet sich, und daraus ergibt sich eine grundsätzlich andere Implementierungsstrategie.

Metaprogrammierung und Makros

Unter Metaprogrammierung versteht man die Entwicklung von Programmen, die ihrerseits Programme erstellen oder modifizieren. Ein simples Shell-Skript, das auf Basis von Kommandozeilenparametern ein oder mehrere Quellcode-Dateien erzeugt, ist damit ein ebenso gutes Beispiel für Metaprogrammierung wie ein ausgefeilter, konfigurierbarer Codegenerator. Beide Ansätze sind aus Sicht der Programmiersprache extern – für das Erzeugen des Codes verwendet man eine andere Sprache (*sh* im ersten Fall, die Konfigurationssprache des Generators im zweiten). Bei einer internen DSL kommt für die Metaprogrammierung die Sprache selbst zum Einsatz. Im Folgenden seien einige der dafür existierenden Mechanismen am Beispiel unterschiedlicher Sprachen illustriert.

Am schwächsten schneiden in der Beziehung Sprachen wie Java ab: Bei ihnen gibt es keine Unterstützung für das Erstellen einer eigenen Syntax über das Erstellen von Klassen, Interfaces und Methoden hinaus. Um die Effekte zu approximieren, kann man auf einige Muster zurückgreifen: Ein Fluent-Interface bezeichnet eine Schnittstelle, die eine Verkettung von Methodenaufrufen so unterstützt, dass insgesamt ein sprechender, „fließender“ Eindruck entsteht. Dazu geben die Methoden normalerweise jeweils wieder das Objekt zurück, für das sie aufgerufen wurden. In einer statisch typisierten Sprache lässt sich eine gute IDE-Unterstützung erreichen, wenn die Methoden durch die Rückgabe mit unterschiedlichen Typen signalisieren, dass die Aktionen jeweils als Nächstes möglich sind. Schließlich ist es über die in Java 5 eingeführten statischen Imports zu realisieren, Klassenmethoden ohne Umwege direkt aufzurufen. Ein Beispiel einer solchen Fluent-API findet sich für die Domäne „Testen“ beim Einsatz der Bibliotheken Mockito und Hamcrest:

```
when(mockedList.contains(argThat(isValid()))).thenReturn("element");
```

Man kann durchaus hinterfragen, ob man hier von einer DSL sprechen sollte, allerdings ist das mittlerweile üblich. Unabhängig von der Art der Implementierung ist eine so gestaltete Programmierschnittstelle für den Entwickler ohne Frage angenehm.

Listing 1: State-Machine-Beschreibung in einer externen DSL

```
cancel:
  transitions from submitted to cancelled,
accept:
  transitions from received to accepted,
               from checking to checked
```

Listing 2: State-Machine-Beschreibung in einer Ruby-basierten internen DSL

```
event :cancel do
  transitions :from => :submitted, :to => :cancelled
end
event :accept do
  transitions :from => :received, :to => :accepted
  transitions :from => :checking, :to => :checked
end
```

Dynamisch bevorzugt

Das DSL-Konzept ist in dynamischen Sprachen extrem verbreitet, allen voran in Ruby, dem es einen Großteil seiner Popularität verdankt. Das populäre Webframework Ruby on Rails verwendet DSLs an diversen Stellen, zum Beispiel für die Definition des Routings (der Abbildung von Request-URIs auf Code), für die Beschreibung von Abhängigkeiten zu Bibliotheken oder für die Definition von Beziehungen zwischen Modell-Entitäten:

```
class Office < ActiveRecord::Base
  belongs_to          :region
  has_one             :manager
  has_many            :employees
  has_and_belongs_to_many :categories
end
```

An dem Beispiel lässt sich gut illustrieren, dass es zwei Lesarten für ein solches Codefragment gibt: Aus fachlicher Sicht sieht der Leser die Definition einer Struktur, die man auch mit einem Fachexperten besprechen kann (um etwa herauszufinden, ob es für jedes Büro tatsächlich nur einen Manager gibt). Aus Implementierungssicht handelt es sich um gültigen Ruby-Code: Was wie ein Schlüsselwort aussieht (*has_many*, *belongs_to* usw.), ist tatsächlich der Aufruf einer statischen Methode, die zur Laufzeit implizit den Code für die Unterstützung der Assoziation erzeugt.

Repräsentativ für ein anderes, häufig eingesetztes Muster ist Builder, eine populäre Bibliothek zum Erzeugen von XML-Dokumenten. Die Struktur des Quellcodes in Listing 3 entspricht der des zu erzeugenden XML. Aus Code-Sicht ist dabei interessant, dass die auf dem *xml*-Objekt aufgerufenen Methoden überhaupt nicht definiert sein müssen – der Aufrufer verwendet einfach die Namen der gewünschten XML-Elemente. Das ermöglicht Rubys *method_missing*-Mechanismus: Das ist der Name einer Methode, die aufgerufen wird, wann immer man eine nicht implementierte Methode verwendet. Sie bekommt den Namen der nicht gefundenen Methode sowie deren Argumente übergeben, und Builder benutzt diese Informationen, um sich die zu erzeugenden XML-Elemente zu merken.

Das Builder-Beispiel zeigt einen Vorteil interner DSLs gegenüber externen: Es ist jederzeit möglich, auf die Konstrukte der Wirtssprache zurückzugreifen (im Beispiel auf Instanzvariablen, Schleifen, Bedingungen und diverse Methoden).

Grüße aus der Lisp-Welt

Als letztes Beispiel dient Clojure, ein Lisp-Dialekt für die JVM (siehe auch [a]). Wie alle Derivate dieser zweitältesten Programmiersprache verfügt Clojure über die Eigenschaft, dass Programme selbst mit Datenstrukturen der Programmiersprache ausgedrückt werden. Diese Eigenschaft, Homoikonizität genannt, macht es besonders leicht, Programme zu schreiben, die Programme schreiben – die wörtliche Definition von Metaprogrammierung. Schließlich muss ein Programm, das ein Programm erzeugen will, nur eine Datenstruktur generieren. Der wichtigste Unterschied zu fast allen anderen Sprachen ist, dass sich dabei auch die Evaluierungsreihenfolge ändern lässt. Die Transformationsmöglichkeit ist in den Compiler integriert, sodass dieser zur Compile-Zeit ein Stück Code mit vom Benutzer definierten Makros in ein anderes transformieren kann. Prinzipiell ließen sich dazu die in den Standardbibliotheken verfügbaren Mechanismen zum Erzeugen von Listen und anderen Strukturen verwenden, zur Vereinfachung verfügen jedoch fast alle Lisp-Systeme über einen Mechanismus, der eine Art Template-Ansatz ermöglicht: Man ergänzt eine Schablone, die den zu generierenden Code beschreibt, um die dynamischen Elemente.

Listing 3: XML-Generierung mit Ruby und der Builder-DSL

```
xml.feed :xmlns=>'http://www.w3.org/2005/Atom' do
  xml.title   @author
  xml.link    :rel=>'self'
  xml.link    :href=>url_for(:action=>'posts', :path=>nil)
  xml.id      :href=>url_for(:only_path=>false,
                             :action=>'posts', :path=>nil)
  xml.updated Time.now.iso8601
  xml.author  { xml.name @author }
  @entries.unshift @parent if @parent
  @entries.each do |entry|
    xml.entry do
      xml.title   entry.title
      xml.link    :href=>url_for(entry.by_date)
      xml.id      entry.atomid
      xml.updated entry.updated.iso8601
      xml.author  { xml.name entry.author.name } if entry.author
      xml.summary do
        xml.div :xmlns=>'http://www.w3.org/1999/xhtml' do
          xml << entry.summary
        end
      end
    if entry.summary
      xml.content do
        xml.div :xmlns=>'http://www.w3.org/1999/xhtml' do
          xml << entry.content
        end
      end
    end
  end
end
```

Listing 4: Hypothetisches Testgerüst für die Addition in Clojure

```
(describe "Addition"
  (testing "of integers"
    (it "computes small sums"
      (= 3 (+ 1 2)))
    (it "computes large sums"
      (= 7000 (+ 3000 4000))))
  (testing "of floats"
    (it "computes small sums"
      (> 0.00001 (Math/abs (- 0.3 (+ 0.1 0.2)))))
    (it "computes large sums"
      (> 0.00001 (Math/abs (- 3000.0 (+ 1000.0 2000.0)))))))
```

Der Makroansatz ist bei Clojure idiomatisch, er wird auch für die Implementierung vieler wichtiger Bibliotheksfunktionen verwendet. Da die gesamte Syntax auf den gleichen Datenstrukturen beruht, ist ein selbst definiertes Konstrukt nicht von einem eingebauten Schlüsselwort (einer „Special Form“ in Lisp-Terminologie) zu unterscheiden. Listing 4 zeigt die Definition eines Testszenarios mit dem Lazytest-Framework, dass diese Verwischung von Grenzen demonstriert.

Gerade in der Lisp-Gemeinde wird die Diskussion um interne DSLs häufig belächelt – nicht, weil sie diese nicht für sinnvoll hält, sondern weil sie diese im Gegenteil schon seit Jahrzehnten als normales Mittel der Entwicklung einsetzt.

Nimmt man es ganz genau, hat der Autor bei keinem der bislang genannten Beispiele tatsächlich eine neue Syntax eingeführt. Dazu braucht man ein „richtiges“ Lisp, zum Beispiel Common Lisp, Scheme oder Racket. In diesen Sprachen ist es möglich, über Reader-Makros wirklich neue syntaktische Konstrukte zu definieren, die im ersten Schritt dann in die Lisp-Datenstrukturen überführt werden.

DSLs versus APIs

Wann genau spricht man nun von einer DSL, wann schlicht von einer Bibliothek oder einer API? Die Grenze zwischen Bibliothek und DSL ist fließend und die Diskussion in den meisten Fällen müßig. Das Java-Beispiel illustriert das Problem. Man kann aber auch mit Fug und Recht behaupten, dass jede gute API so entworfen sein sollte, dass sie einsetzende Programme lesen lassen wie die Fachbeschreibung eines Domänenexperten.

Das Argument, dass es sich bei internen DSLs schlicht nur um Bibliotheken und APIs handelt, ist in Diskussionen praktisch und vielseitig einsetzbar. Man kann es verwenden, um interne DSLs abzuqualifizieren: Diese sind überhaupt keine „richtigen“ DSLs, weil sie eben nur die Mechanismen der Sprache nutzen. Es ist aber auch geeignet, um ein Argument gegen DSLs zu entkräften: Domänenspezifische Sprachen erzeugen, wenn sie intern sind, weder mehr noch weniger Einarbeitungs- oder Wartungsprobleme als andere Bibliotheken. Schließlich lassen sie sich benutzen, um die Unterschiede zwischen Programmiersprachen herauszuarbeiten: Wie mächtig eine Sprache ist, kann man unter anderem daran festmachen, wie einfach es ist, sie mit Bibliotheken um Konstrukte zu erweitern, die aussehen, als wären sie eingebaut.

Innerhalb des Entwicklungsprozesses kommt internen DSLs gerade durch die Nähe zu APIs eine besondere Rolle zu. In Programmiersprachen, die sich durch entsprechende Mechanismen auszeichnen, ist es absolut üblich, die Programmiersprache kontinuierlich dem Problem entgegen-

wachsen zu lassen: Muster, die man bei der Problemlösung erkennt, werden zu Bestandteilen der Sprache, in der man die Lösung formuliert. Leicht idealisiert bedeutet das, dass man zu jedem Zeitpunkt die perfekte Programmiersprache verwendet, nämlich genau die, die am besten auf die Aufgabe passt. Das mag nun revolutionärer klingen, als es ist – schließlich wendet man auch bei einer noch so inflexiblen Programmiersprache dieselben Prinzipien an, wenn man eine vom fachlichen Code verwendete Bibliothek während der Entwicklung Schritt für Schritt erweitert.

Wachstum von unten

Programmiersprachen passen besonders gut zu einer solchen Philosophie, wenn sie die iterative Entwicklung auch mit anderen Mechanismen unterstützen. Je nach Überzeugung oder dem Grad ideologischer Verblendung kann man dazu zwingend ein dynamisches Typsystem oder eine REPL (Read-Eval-Print Loop) zählen.

Beim Entwurf einer internen DSL gibt es eine Reihe wiederkehrender Muster. Das vielleicht wichtigste davon ist die Trennung eines logischen Modells, das eine bestimmte Semantik hat beziehungsweise anstößt, von der konkreten Syntax, mit der man es ausdrückt. Ein Beispiel dafür sind UI-Bibliotheken, die eine Reihe von Abstraktionen einführen, um die Fachdomäne „Benutzerschnittstelle“ abzubilden – Fenster und Panels, verschiedene Arten von Layouts, diverse Kontrollelemente, Nachrichten, Eventhandler und so weiter. Eine bestimmte Konfiguration von ineinander verschachtelten Fenstern und Kontrollelementen ist damit eine Instanz eines solchen Modells, deren Erstellung Aufgabe des Entwicklers ist.

Andere Beispiele für Modelle aus technischen Domänen sind Abhängigkeitsgraphen eines Build-Management-Werkzeugs, Statusmaschinen und Workflows oder Regelwerke für Plausibilisierungen: In diesen Fällen kann es sinnvoll sein, das inhaltliche Modell zunächst unabhängig von einer DSL zu entwerfen und erst später eine möglichst angenehme Syntax dafür zu definieren.

Alles in Ordnung?

DSLs sind in Mode, und gerade die interne Variante wird im Moment häufig verwendet, um die Stärken einer Programmiersprache herauszustellen. Dabei ist es leicht, über das Ziel hinauszuschießen und etwas zu erstellen, das zwar auf den ersten Blick beeindruckt, in der Praxis jedoch mehr Probleme als Lösungen erzeugt. Nicht umsonst gilt in der Lisp-Welt als erste Regel für Makros, dass man sie vermeiden soll, wenn es eine alternative Lösung gibt. Ganz so drastisch muss man die Dinge nicht sehen, man sollte sich aber durchaus auch der Nachteile bewusst sein:

- Für Domänenexperten, die selbst Quellcode entwickeln wollen, sind interne DSLs deutlich schwieriger verwendbar als externe, und zwar nicht nur wegen der syntaktischen Einschränkungen, sondern vor allem durch die Fehlermeldungen, die aus der Werkzeugkette der Wirtssprache stammen. Zielgruppe für interne DSLs sind daher in der Regel Entwickler, die auch die Wirtssprache beherrschen.
- Insbesondere bei DSL-Implementierungen, die die Fähigkeiten der Wirtssprache ausreizen, besteht das Risiko, dass es für einen Entwickler bei einem Fehler schwierig ist, die Ursache herauszufinden.
- Je nach Programmiersprache und verwendetem Ansatz ist die Kombinierbarkeit von Konstrukten aus einer DSL mit den sonstigen Mechanismen beeinträchtigt.

Den Nachteilen interner DSLs stehen naturgemäß auch einige Vorteile gegenüber:

- Für die Entwicklung einer internen DSL sind keinerlei zusätzliche Werkzeuge notwendig, die über die für die eingesetzte Programmiersprache verfügbaren hinausgehen, die Werkzeugkette wird also komplett wiederverwendet.
- Es ist zu jedem Zeitpunkt möglich, die Konstrukte der Wirtssprache zu verwenden, wenn die Mächtigkeit der internen DSL nicht ausreicht.
- Durch die Kombinationsmöglichkeit mit den Strukturen der Wirtssprache lässt sich ein hoher Grad von Abstraktion mit dem jeweils dafür am besten geeigneten Mittel erreichen.

Fazit

Wenn eine DSL entwickelt wird, um das Leben des Entwicklers zu vereinfachen und nicht dazu, dass ein Fachexperte Code erstellen kann, drängt sich der interne Ansatz auf. Konstrukte, die eine höhere Ausdrucksstärke der Sprache ermöglichen, sind ein zentraler Faktor hinsichtlich der Produktivität und Qualität der formulierten Lösung. Vielleicht werden die besten internen DSLs von den Entwicklern erstellt, die den Begriff noch nie gehört haben oder sich nicht dafür interessieren, weil die domänenspezifische Ausprägung ohnehin ein Kernidiom ihrer Programmiersprache ist. Für alle anderen gilt: Bevor man sich auf die Komplexität der Erstellung einer externen DSL einlässt, sollte man auf jeden Fall prüfen, ob sich der Anwendungsfall nicht auch mit den Mittel der ohnehin eingesetzten Programmiersprache abbilden lässt. (ane)

STEFAN TILKOV

ist Geschäftsführer und Principal Consultant bei der innoQ Deutschland GmbH und beschäftigt sich dort mit Architekturen verteilter Systeme.

Onlinequellen

[a] Stefan Tilkov; Gelungene Mischung. Clojure: Ein pragmatisches Lisp für die JVM
http://heise.de/-1030144

[b] Stefan Tilkov, Markus Völter; SoftwareArchitekTOUR-Podcast, Episode 27: Interne DSLs (auf Heft-DVD)
http://heise.de/-1167843

Literatur

[1] Martin Fowler; Domain Specific Languages; Addison-Wesley 2010

[2] Paul Graham; On Lisp. Advanced Techniques for Common Lisp; Prentice-Hall 1994; www.paulgraham.com/onlisp.html

Alle Links: www.ix.de/ix1116122