www.javaspektrum.de

Ausgabe 2/2022
D € 9,90 · A € 10,90 · CHF 18,20

4 194156 109901 02

# JavaSPEKTRUM

Magazin für professionelle Entwicklung und digitale Transformation

# Teufel im Detail –
## Effizientes Engineering von Anwendungen


## Domain Specific Languages – eine Einführung



Sonderdruck aus
JavaSPEKTRUM 2/2022

Teufel im Detail – Effizientes Engineering von Anwendungen

G30325

## Das 1 x 1 der DSLs

# Domain Specific Languages

Sonderdruck aus
JavaSPEKTRUM 2/2022

Andreas Schäfer

**Das Konzept domänenspezifischer Modellierung mit Domain Specific Languages (DSLs) ist nicht neu, aber dank moderner Tool-Unterstützung durch Language Workbenches ist die Umsetzung heute praktikabler denn je. Dieser Artikel gibt eine Einführung in die DSL-Thematik und deren Umsetzung und fasst zusammen, wann sich ein Einsatz lohnt.**

Wenn wir im Arbeitsalltag Software entwickeln, tun wir das in der Regel mit *General Purpose Languages (GPLs)* [Wiki1] wie zum Beispiel Java. Der Vorteil an GPLs ist, dass sie sehr mächtig sind. Sie sind Turing-vollständig [Wiki2] und technisch orientiert und damit auf beliebige Fachlichkeit anwendbar.

DSLs verfolgen einen anderen Ansatz: Statt möglichst breit einsetzbar zu sein, fokussiert sich eine DSL präzise auf einen bestimmten fachlichen Anwendungskontext. Dieser Anwendungskontext ist die *Domäne* (engl. *Domain*) im Ausdruck *Domain Specific Languages*. Eine DSL möchte eine Domäne mit genau der Terminologie und den Notationen abbilden, die in der Domäne üblich sind.

Als Domäne kommt dabei jegliche (vor allem nicht-triviale) Fachlogik infrage. Hier zur Verdeutlichung ein paar Beispiele für Domänen, für die bereits DSLs entwickelt wurden oder werden:

- Steuerberechnung für Lohnabrechnungen [You1]
- Organisation der kommunalen Verwaltung [You2]
- Kühlungsalgorithmen von Kühlschränken [Vö13]
- Verarbeitung von Sensordaten für automatisiertes Fahren [You3]

Das Ziel dieses Domänenfokus: Diejenigen, die die DSL benutzen, um damit *fachlich getriebene Programme* zu schreiben, sollen *keine Berührungspunkte mit technischen Implementierungsdetails* haben müssen, wenn diese nicht zum Kern der Domäne gehören. Dem Anspruch nach ist für die DSL-Nutzung also kein technischer Hintergrund nötig, sondern vor allem Domänenwissen. Die Fachseite wird somit in die Lage versetzt, selbst Programme lesen, verstehen und schreiben zu können.

Für das Sprachdesign bedeutet das, dass die DSL auf *geeigneten fachlichen Abstraktionen* aufbauen muss, damit fachliche Nutzer die

**Andreas Schäfer** ist seit 2014 IT-Berater und Softwareentwickler bei der IKS GmbH und befasst sich dort vorrangig mit der Entwicklung von Geschäftsanwendungen in Java. Neben Java (vor allem Spring, Hibernate/JPA und Webservices/CXF) liegen seine Interessen in den Bereichen Cloud-Services (AWS), Frontend-Entwicklung (Angular) und der Sprachentwicklung mit JetBrains MPS.
E-Mail: a.schaefer@iks-gmbh.com

in ihrer Domäne gültigen Regeln mit der DSL effizient ausdrücken können. Ein einfaches Beispiel: Ein Grafik-Spezialist, der geometrische Figuren auf dem Bildschirm anordnen möchte [MPS3], kann sich mit

```
Painting - MyDrawing
Output format: java
circle: [ x: 125 | y: 150 | radius: 50 | color: blue ]
square: [ x: 100 | y: 100 | size: 300 | color: green ]
line: [ start point: ( x: 100 | y: 100 ) | end point:
  ( x: 50 | y: 200 ) | color: red ]
```

effizienter und fachlich fokussierter ausdrücken als mit

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Dimension;

public class MyDrawing extends JFrame {
  private JPanel panel = new JPanel() {
    @Override
    protected void paintComponent(Graphics graphics) {
      super.paintComponent(graphics);
      graphics.setColor(Color.blue);
      graphics.drawOval(125, 150, 50, 50);
      graphics.setColor(Color.green);
      graphics.drawRect(100, 100, 300, 300);
      graphics.setColor(Color.red);
      graphics.drawLine(100, 100, 50, 200);
    }
  };

  public void initialize() {
    this.setTitle("MyDrawing");
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.add(panel);
    panel.setPreferredSize(new Dimension(500, 500));
    this.pack();
    this.setVisible(true);
  }

  public static void main(String[] args) {
    MyDrawing canvas = new MyDrawing();
    canvas.initialize();
  }
}
```

Das gilt insbesondere dann, wenn die Struktur der Elemente vorgegeben ist. Im Beispiel wird durch die Auswahl eines `circle` bereits vorgegeben, dass dieser die Properties `x`, `y`, `radius` und `color` enthält und nicht zum Beispiel *size* wie bei einem `square`. Das Beispiel stellt eine *textuelle* DSL dar. Allgemein können DSLs auch *symbolische* (z. B. mathematische Notationen), *tabellarische* oder *grafische* Elemente enthalten.

Wir werden später noch genauer auf die Vorteile von DSLs schauen. Zunächst sollten wir aber die Frage klären, welche Tools man braucht, um eine DSL zu bauen.

## Moderne DSL-Tools

Zunächst eine begriffliche Einordnung: Wenn von DSLs die Rede ist, trifft man oft auf die Unterscheidung zwischen *internen* und *externen* DSLs. *Interne* (auch: *eingebettete*) DSLs sind DSLs, die innerhalb bestimmter GPLs (z. B. Ruby) als eingebettete Konstrukte definiert werden können. Diese Konstrukte sind im Grundsatz mit APIs vergleichbar, bieten aber eine größere syntaktische Freiheit, was die Bezeichnung als DSLs rechtfertigt.

Interne DSLs sind historisch wiederum eine Reaktion auf frühe *externe* DSLs, also DSLs, die *nicht* in GPLs eingebettet sind. In den Anfangstagen externer DSLs mussten diese noch aufwendig manuell kompiliert werden. Da interne DSLs dagegen auf die Kompilierungsmechanismen der sie umgebenden GPL zurückgreifen konnten, fiel dieser Extra-Schritt dort weg, was im Vergleich eine Vereinfachung war. [You4]

Aus heutiger Sicht haben interne DSLs allerdings klare Nachteile:

- Sie sind an die sie umgebende GPL gebunden.
- Die Tool-Unterstützung ist sehr limitiert, insbesondere für die Nutzung der DSL.

Moderne DSLs sind wieder externe DSLs. Ein großer Vorteil moderner externer DSLs ist die gute Tool-Unterstützung (auf die wir gleich zu sprechen kommen). Ein weiterer Vorteil ist, dass externe DSLs eigenständig und damit nicht an eine Umgebungssprache gebunden sind. In moderne DSLs lassen sich sogar leicht bei Bedarf Elemente verschiedener anderer Sprachen einbinden, sodass ein *modularer Aufbau* von Sprachen aus Bestandteilen anderer Sprachen möglich wird.

## Wie sieht nun ein Tool zur DSL-Erstellung aus?

Um DSLs zu bauen, verwendet man sogenannte *Language Workbenches*. Das sind Tools, die vom Aufbau her vergleichbar sind mit IDE-Tools für das Arbeiten mit GPLs, also zum Beispiel IntelliJ oder Eclipse. Sie bauen zum Teil sogar auf den gleichen Plattform-Technologien auf.

Das Besondere an Language Workbenches ist, dass man damit auf einer *Meta-Ebene* arbeitet: Man *nutzt* nicht nur Sprachen, um Programme zu schreiben, sondern *definiert* vor allem eigene Sprachen – die dann wiederum genutzt werden können, um Programme zu schreiben.

Wenn im DSL-Kontext dagegen von *IDEs* die Rede ist, ist oft nicht das Tool selbst gemeint (das ist ja die Language Workbench), sondern diejenigen Hilfs-Mechanismen, die man zusätzlich zu Syntax und Semantik der DSL im Rahmen der Sprachentwicklung selbst definiert, um deren Nutzern den Umgang damit zu erleichtern.

In der Regel wird die gleiche Language Workbench, in der die DSL definiert wird, übrigens nicht nur für die Entwicklung der Sprache verwendet, sondern auch zum Schreiben der fachlichen Programme unter Verwendung der DSL. Zwei in der Praxis häufig eingesetzte Language Workbenches sind:

- JetBrains MPS [MPS1]
- Eclipse Xtext [Xtext]

Die Definition einer DSL in einer Language Workbench besteht typischerweise aus folgenden *Aspekten*.

### Struktur

Hier werden die Sprachkonzepte der DSL und deren mögliche Beziehungen definiert und damit die *abstrakte Syntax* der Sprache.

Diese legt fest, welche Gestalt der sogenannte *Abstract Syntax Tree (AST)* [Wiki3] von Programmen dieser Sprache annehmen kann, also der sich aus der Hierarchie der Anweisungen ergebende Baum. Die in der Struktur definierten Konzepte können Wurzel-Elemente des Baums sein, in Eltern-Kind-Beziehungen mit verschiedener Kardinalität zueinander stehen und andere Konzepte referenzieren.

*Abstrakt* ist hier im Sinne einer Abstraktion von der *konkreten* Sprachsyntax zu verstehen. So sind etwa im Beispiel oben alle Trenn- und Einkapselungszeichen (wie Doppelpunkte und Klammern) und auch die erklärenden Texte („Output format", „circle", ...) reine Editor-Features (siehe nächster Aspekt *Editor*), die ausgetauscht werden können, ohne den AST zu verändern. So könnte etwa die folgende Zeile

```
circle: [ x: 125 | y: 150 | radius: 50 | color: blue ]
```

*bei gleichbleibender abstrakter Syntax* auch so aussehen (es müsste nur der Editor ausgetauscht werden):

```
C(125/150/50/blue)
```

Die Strukturdefinition eines Konzepts beinhaltet also nur dessen *definierende* Eigenschaften und mögliche Beziehungen zu anderen Konzepten im AST. Definition der Struktur des *circle*-Konzepts aus obigem Beispiel (Beispiel aus MPS):

```
concept Circle extends Shape
      implements <none>

instance can be root: false
alias: circle
short description: <no short description>

properties:
x : integer
y : integer
radius : integer

children:
<< ... >>

references:
<< ... >>
```

Die `color` findet man hier deswegen nicht, weil sie zum Parent-Konzept `Shape` gehört (siehe *extends*-Ausdruck). Es handelt sich um einen Blatt-Knoten des AST, der daher keine Kind-Elemente besitzt. Es werden auch keine anderen Knoten referenziert.

### Editor

Hier wird die *konkrete Syntax* der DSL festgelegt, also die konkrete Erscheinung der DSL, in der Programme gelesen und geschrieben werden. Definition des Editors für das *circle*-Beispiel (Beispiel aus MPS):

```
<default> editor for concept Circle
  node cell layout:
     [- circle: [ x: { x } | y: { y } | radius: { radius } |
     # ShapeColor # ] -]
```

Die `ShapeColor` bezieht sich auf eine Eigenschaft des Parent-Konzepts `Shape` und ist daher im Struktur-Beispiel oben nicht sichtbar.

### Statische Semantik/IDE-Features

Es gibt eine Reihe definierbarer Mechanismen, die dazu dienen, die Nutzung der DSL zu erleichtern oder bestimmte Fehler zu vermeiden (und ggf. automatisch zu beheben). Dazu gehören:

- Constraints
- Typsysteme
- Validierungsregeln
- Intentions (Source-Code-Aktionen im aktuellen Kontext)
- Quickfixes (automatische Fehlerbehebungen)
- und viele mehr (Context Actions, Dataflow, ...).

Diese werden wir hier nicht im Detail behandeln, da dies den Rahmen des Artikels sprengen würde.

### Ausführungssemantik (Generator / Interpreter)

Hier wird die Semantik des DSL-Codes festgelegt. Um vom in der DSL geschriebenen Programm zum maschinenausführbaren Code zu kommen, braucht es eine *Übersetzung* zwischen fachlichem Code und technischer Ausführungsplattform. Das passiert entweder durch *Code-Generierung*, also durch *Model-to-Model-Transformation* des DSL-Modells in beispielsweise ein Modell aus ausführbarem GPL-Code, oder durch Bereitstellung eines Interpreters, der die DSL-Anweisungen direkt übersetzt.

Hier ein Generator-Template für die Übersetzung eines `circle` in Java-Code (Beispiel aus MPS):

```
template reduce_Circle
input Circle

parameters
<< ... >>

content node:
{
  Graphics g = null;
  <TF {
    ->$g.setColor(Color.->$red);
    ->$g.drawOval($10, $10, $10, $10);
  } TF>
}
```

Die mit `$` markierten Werte sind Platzhalter, die im *Inspector* (erweiterte Editor-Ansicht) mit Knoten-Properties verknüpft werden können. Die ersten beiden Werte innerhalb von `drawOval` werden zum Beispiel durch `node.x` und `node.y` ersetzt.

Nicht jede DSL muss übrigens zwingend in GPL-Code übersetzt werden. Ziel einer Generierung kann zum Beispiel auch eine XML-Konfigurationsdatei oder eine Textdatei zur Dokumentation sein. Im Extremfall kommen DSLs sogar ganz ohne Übersetzung aus und dienen zum Beispiel der reinen Visualisierung von Domänenregeln.

Gerade die im letzten Abschnitt genannten Stichwörter *Code-Generierung* und *Model-to-Model-Transformation* rufen oft Assoziationen zum *MDSD*-Ansatz hervor, daher schauen wir uns nun kurz das Verhältnis der Ansätze zueinander an.

## DSLs und MDSD

*MDSD* (*Modellgetriebene Softwareentwicklung*, engl. *Model-Driven Software Development*) kann als Oberbegriff angesehen werden, unter den DSLs als Spezialkategorie fallen.

Klassische MDSD-Ansätze arbeiten oft mit UML-basierter Modellierung und darauf aufbauender Code-Generierung oder nutzen Modellierungs-Frameworks wie Eclipse EMF [EMF]. DSLs haben sich

in der Entwicklungsgeschichte von MDSD als ein sinnvolles Element zur Weiterentwicklung des Ansatzes herauskristallisiert.

Klassische MDSD-Ansätze haben insofern weiterhin ihre Daseinsberechtigung, als dass sie sich gut für technisch getriebene und standardisierte Anwendungsfälle eignen, bei denen die Arbeitsweise gut zu den Modellstrukturen der Standards passt und die Implementierung von DSLs somit ein redundanter Mehraufwand wäre. Der klare Vorteil von DSLs dagegen liegt in der weitaus größeren Flexibilität hinsichtlich Syntax und IDE-Unterstützung.

Mit modernen Language Workbenches sind textuelle, symbolische, tabellarische und grafische Syntax beliebig miteinander kombinierbar. Die Syntax kann exakt auf die Domäne zugeschnitten werden. In klassischer MDSD ist man dagegen an die Syntax der Modellierungs-Standards gebunden.

Ähnliches gilt für die IDE-Unterstützung, da bei DSLs auch zum Beispiel kontextabhängige Aktionen exakt nach domänenspezifischen Regeln definiert werden können.

DSLs bieten also interessante Anwendungsmöglichkeiten in der domänenspezifischen Modellierung. Aber schauen wir noch einmal genauer hin: Wann genau können wir DSLs einsetzen?

## Vorteile

- *Kürze*: DSL-Code ist verglichen mit entsprechendem GPL-Code kürzer und damit übersichtlicher. Dies hilft der Produktivität und Wartbarkeit.
- *Effizienz*: Syntax und IDE-Unterstützung der DSL können so gestaltet werden, dass fachliche Sprachbenutzer damit sehr effizient arbeiten können.
- *Entkopplung*: Durch die *Separation of concerns* von Fachlichkeit und technischer Plattform entkoppelt man deren Lebenszyklen. Die fachlichen Modelle können zum Beispiel sehr langlebig sein, während die technische Plattform austauschbar bleibt. Technische Optimierungen können auf Generator-Ebene erfolgen, ohne dass sich der fachliche Code ändert.
- *Qualität*: Durch das Sprachdesign kann die technische und fachliche Korrektheit von DSL-Programmen erzwungen werden, was die Anzahl möglicher Fehler und somit die Qualität der Programme erhöht.
- *Domänennahe Nutzerunterstützung*: Fehlermeldungen und Hinweise können domänenspezifisch gestaltet werden, sodass diese für fachliche Benutzer intuitiv verständlich sind.
- *Analysierbarkeit*: Durch ihre dichte Semantik sind DSL-Programme leicht analysierbar, was zum Beispiel für technische Validierungen genutzt werden kann.
- *Fokus*: Da der DSL-Code sich auf die Domäne fokussiert, ist das Denken beim Schreiben der Programme auf die Domäne gerichtet. Auch das gemeinsame Durchgehen des DSL-Codes im Team ist fokussierter.
- *Besseres Domänenverständnis*: Der Prozess der Sprachentwicklung schafft (manchmal bei allen Projektbeteiligten) ein besseres Verständnis der Domäne.
- *Gemeinsames Domänenverständnis*: Das Verständnis der Domänenbegriffe wird bei allen Beteiligten auf ein gemeinsames Niveau gebracht (ubiquitäre Sprache), was der besseren Kommunikation dient.
- *Ermöglichen von Handlungsfähigkeit*: Domänen-Experten werden durch die Entwicklung einer DSL oft erst in die Lage versetzt, sich an der Implementierung zu beteiligen.

## Einsetzbarkeit von DSLs

Zunächst einmal: Domänen können durchaus technischer Natur sein. Ein einfaches Beispiel für eine technische sogenannte *Utility-DSL* ist eine DSL, die die Erstellung von Artefakten mit einer bestimmten Struktur, zum Beispiel WDSLs für SOAP-Webservices [IKS], vereinfacht. Eine solche Utility-DSL könnte ich als Entwickler selbst in meinem Arbeitsalltag einsetzen, um mir einige Aufgaben zu erleichtern. In diesem Fall hätte ich sowohl die Rolle des Sprachentwicklers als auch die des Sprachbenutzers inne.

In fachlicheren, weniger technischen Domänen sind die Rollen oft nicht mehr identisch. Idealerweise sind es Projektbeteiligte der Fachseite, die die fachlichen Programme schreiben. Die Entwickler-Seite ist dann für die Entwicklung der DSL und insbesondere der Generatoren zuständig.

Dabei benötigt der Sprachentwicklungsprozess eine enge Zusammenarbeit der Entwickler- mit der Fachseite, da ein Verständnis der Domäne für ein gutes Sprachdesign entscheidend ist. Insbesondere müssen für die Sprachstruktur geeignete fachliche Abstraktionen gefunden werden. Daher beinhaltet die DSL-Entwicklung die Definition einer ubiquitären Sprache im Sinne von Domain-Driven Design. Letztlich kann sogar die DSL selbst als Formalisierung einer solchen ubiquitären Sprache angesehen werden.

Um zu verdeutlichen, wann der Einsatz einer DSL im fachlichen Kontext sinnvoll sein kann, bietet sich eine kurze Übersicht von Vorteilen und Herausforderungen [Vö13] an (siehe Kästen).

## Herausforderungen

- *Aufwand*: Die DSL-Konstruktion ist mit initialem Mehraufwand verbunden, der in der Projektplanung berücksichtigt werden muss.
- *Kenntnisse*: Die DSL-Konstruktion erfordert Kenntnisse sowohl geeigneter Tools als auch von Best Practices der Sprachentwicklung. Aktuell sind diese Kenntnisse noch wenig verbreitet.
- *Kommunikation*: Es ist viel Kommunikation zwischen verschiedenen Rollen nötig, vor allem Sprachentwicklern und Domänenexperten. Die Organisationsstruktur muss dies hergeben.
- *Eindeutiges Domänenverständnis*: Die fachlich Beteiligten müssen ein gemeinsames Verständnis der Domäne haben. Widersprüchliche Auffassungen von Begrifflichkeiten erschweren die Formalisierung.
- *Wartung*: Auch eine DSL muss im Lauf der Zeit gewartet und weiterentwickelt werden.
- *Verwaltung von DSLs*: Hat sich die Entwicklung von DSLs in einer Organisation einmal etabliert, müssen die einzelnen DSLs so organisiert werden, dass Überlappungen vermieden werden (z. B. durch Modularisierung), um die Übersichtlichkeit zu wahren.
- *Bindung*: Hat einmal ein Investment in DSLs stattgefunden und wurden dadurch bestimmte Strukturen geschaffen, kann sich eine Organisation an diese gebunden fühlen, was künftige Umstrukturierungen erschweren kann.
- *Fehlende Tool-Interoperabilität*: Ein Wechsel zwischen verschiedenen DSL-Tools ist nicht ohne größeren Aufwand möglich, da es keine Interoperabilität zwischen den gängigen Language Workbenches gibt.
- *Rollen*: Die Firmenkultur muss kompatibel sein mit den Rollen, die Entwicklern und Domänenexperten bei der Nutzung von DSLs zukommen.

Es gibt also sowohl klare Vorzüge als auch Punkte, die beachtet werden müssen. Oft sind wohl aktuell noch die fehlenden Kenntnisse rund um DSLs ein Hindernis. Schauen wir uns also noch an, wie man tiefer ins Thema einsteigen kann.

## Ressourcen zum Thema

Eine umfassende Einführung zu DSLs liefert Markus Völters Buch „DSL Engineering" [Vö13]. Viele der hier angerissenen Punkte lassen sich dort noch mal im Detail nachlesen. Außerdem werden viele Design- und Implementierungs-Richtlinien gegeben, unterfüttert mit praktischen Beispielen.

Wer sich näher mit MPS als Language Workbench beschäftigen möchte, findet im „Fast Track to MPS" [MPS2] Erklärungen und Tutorials, die sich gut zum Einstieg eignen. Ein paar von mir in MPS geschriebene kleine Beispielsprachen finden sich auch im IKS-GitHub [IKS].

Eine der Besonderheiten von MPS ist der *projektionale Editor*. Das Konzept findet sich sowohl in der Sprach-Definition als auch in der Sprach-Nutzung und bedeutet, dass man *nicht Text editiert, sondern direkt die Knoten des AST*. Die Trennung zwischen Struktur und Editor spielt also auf beiden Ebenen (DSL-Definition und -Nutzung) eine wichtige Rolle. Letztlich sind die Sprachen, mit denen die Aspekte (Struktur, Editor, Generator, ...) in MPS definiert werden, auch wieder eigene DSLs. MPS baut also intern auf der gleichen Art von Konzepten auf, die man damit definieren kann. Ein Vorteil des projektionalen Editors ist, dass dadurch die Notwendigkeit des Parsens (die man bei textuellen Programmen hätte) wegfällt. Die Nutzung des projektionalen Editors ist allerdings zunächst gewöhnungsbedürftig, da man im Gegensatz zu Text-Editoren nicht frei herumeditieren kann, sondern an die vorgegebene Sprachsyntax gebunden ist.

Im Gegensatz zu MPS nutzt Xtext das Konzept des projektionalen Editors übrigens nicht, sondern setzt stattdessen auf textuelle Editoren und Parser.

## Literatur und Links

**[EMF]** https://www.eclipse.org/modeling/emf
**[IKS]** https://github.com/iks-gmbh-playground/jetbrains-mps-samples
**[MPS1]** https://www.jetbrains.com/mps
**[MPS2]** https://www.jetbrains.com/help/mps/fast-track-to-mps.html
**[MPS3]** https://www.jetbrains.com/help/mps/shapes-an-introductory-mps-tutorial.html
**[Vö13]** M. Völter, DSL Engineering, dslbook.org, 2013
**[Wiki1]** https://de.wikipedia.org/wiki/General_Purpose_Language
**[Wiki2]** https://de.wikipedia.org/wiki/Turing-Vollständigkeit
**[Wiki3]** https://de.wikipedia.org/wiki/Syntaxbaum#Abstrakte_Syntaxbäume
**[Xtext]** https://www.eclipse.org/Xtext
**[You1]** https://youtu.be/q56wzLQkEho
**[You2]** https://youtu.be/MWahVlmYEKE
**[You3]** https://youtu.be/k_WEzDdtarw
**[You4]** https://youtu.be/ZfdAwV0HlEU