Textuelle DSLs – Programmiersprachen zum selbermachen

Arno Haase, arno.haase@haase-consulting.de

Markus Völter, voelter@acm.org, www.voelter.de

Die meisten Softwaresysteme sind so groß, dass ein einzelner Entwickler sie nicht mehr überall in den Implementierungsdetails überblickt. Deshalb brauchen Entwickler ein abstrakteres Vokabular als den Quelltext, um über das System reden zu können. Diese Kernabstraktionen können in UML-Diagrammen oder z.B. in XP-Projekten als „Metapher" vorliegen [1], sie können präzise oder informell definiert sein, aber sie sind notwendig, damit ein System änderbar ist. Ohne dieses gemeinsame Vokabular birgt jedes Gespräch über das System die große Gefahr von unbemerkten Missverständnissen, und die Qualität und Änderbarkeit leidet dem entsprechend [2].

Modellgetriebene Softwareentwicklung [3] hat das Ziel, solche Abstraktionen explizit zu machen, z.B. indem man auf Architekturebene Bausteine und ihr Zusammenspiel beschreibt [4]. Meist denkt man dabei nur an UML-Klassendiagramme, aus denen Infrastrukturcode für das Zusammenspiel von Komponenten generiert wird. Dieser Artikel beschreibt dagegen, wie man das *Verhalten* von Komponenten durch *textuelle* Sprachen beschreiben kann und dabei diese Art von Abstraktionen explizit macht.

Der Artikel gliedert sich grob in drei Teile. Zunächst beschreibt er anhand eines Beispiels, wie man mit Hilfe textueller DSLs das Verhalten eines Systems beschreiben kann. Auf der Basis dieses Beispiels folgt eine Diskussion der Vor- und Nachteile sowie der sinnvollen Einsatzgebiete textueller DSLs, und der Artikel schließt mit einer Reihe von Best Practices.

Beispiel

Grau ist alle Theorie, fangen wir also mit einem Beispiel an. Wir betrachten eine hypothetische Webanwendung, die Kreditforderungen verwaltet und berechnet, in welcher Höhe eine Rückzahlung der Forderung noch zu erwarten ist. Die Anwendung hat eine Web Service-Schnittstelle, und fachliches Reporting spielt eine wichtige Rolle. Abb. 1 zeigt die Architektur des Systems.

Nehmen wir jetzt einmal an, wir haben in einem ersten Schritt die Schnittstellen dieser Komponenten definiert und eine Komponenteninfrastruktur aufgesetzt, sei es durch Generierung oder von Hand mit Spring oder wie auch immer. Wir können uns also jetzt an die Implementierung machen.

Zu GUI und Persistenzschicht wurde an anderer Stelle schon viel gesagt, wir konzentrieren uns hier auf die Geschäftslogik. Die zerfällt für dieses System im Wesentlichen in drei Bestandteile (s. Abb. 2). Da ist zunächst einmal der Teil, der sich um die Datenpflege kümmert, der also Daten vom GUI oder einem Web Service-Aufruf entgegennimmt und in der Persistenzschicht ablegt bzw. Daten aus der Persistenzschicht aufbereitet und an eine der Schnittstellen weiterleitet. Dieser Teil der Geschäftslogik ist eher technischer Natur, er ist für viele Geschäftsanwendungen ähnlich und nicht sehr spannend. Er ist ein Kandidat dafür, teilweise aus einem Domänenmodell generiert zu werden, aber wir betrachten ihn hier nicht näher.

Der zweite Bestandteil der Geschäftslogik sind die Rechenregeln. Sie beschreiben, wie der zu erwartende Restwert einer Forderung ermittelt werden soll, und sie sind aus fachlicher Sicht das Herzstück des Systems. Diese Regeln werden von einer Gruppe von Mathematikern geliefert, die die zugrundeliegenden Modelle fortlaufend weiter entwickeln. Die Schnittstelle für die Berechnung ist recht einfach – eine Forderung mit ihren Sicherheiten und Schuldnern sowie Details zu deren finanzieller Situation geht hinein, und der Restwert kommt als Zahl zurück. Die Regeln sind aber umfangreich und kompliziert, und sie werden fortlaufend weiter entwickelt.

Der dritte Bestandteil der Geschäftslogik ist die Überprüfung der Daten auf Plausibilität. Daten, die über das GUI erfasst werden, müssen bestimmte Bedingungen erfüllen, damit das System sie annimmt und in der Datenbank ablegt, z.B. müssen Pflichtfelder gefüllt sein. Es gibt aber dabei auch „weiche“ Randbedingungen, deren Verletzung nur zu einer Rückfrage führt – so sollten bei der Bilanz eines Unternehmens die Summe von Aktiva und Passiva übereinstimmen, aber wenn durch ein Versehen die tatsächliche Bilanz fehlerhaft ist, dann muss das Programm auch die fehlerhafte Bilanz akzeptieren. Die Web Service-Schnittstelle dagegen hat sehr viel schwächere Anforderungen an die Konsistenz von Daten, damit man große Datenmengen ungeprüft importieren und eventuelle Inkonsistenzen anschließend manuell beseitigen kann. Und der Rechenkern prüft die an ihn übergebenen Daten noch einmal besonders streng, bevor er mit einer Berechnung beginnt.

Diese Aufteilung der Geschäftslogik ist natürlich stark vereinfacht. Wir könnten weitere Bestandteile herauslösen und versuchen, sie mit DSLs zu beschreiben, z.B. Reporting oder Security. In jedem Fall bleibt aber ein Rest übrig, für den es sich der Aufwand einer DSL nicht lohnt.

Textuellen DSLs hängt oft der Nimbus des Exotischen an, weil zur Zeit nur wenige Projekte explizit ihre eigenen textuellen DSLs definieren. Es gibt aber eine Reihe von weit verbreiteten DSLs, die man aber oft nicht direkt als solche wahrnimmt.

Ein prominentes Beispiel ist SQL, die universelle DSL für Zugriffe auf relationale Datenbanken. Sie ist recht gut standardisiert und erfreut sich großer Verbreitung. Ein anderes Beispiel ist HTML, die universelle DSL für die Darstellung in Browsern.

Ein Beispiel, das noch näher an der Java-Welt liegt, sind JSPs; sie sind ein Beispiel für eine DSL, die compiliert statt interpretiert wird. Eines ihrer zentralen Features ist die Erweiterbarkeit, sowohl durch eingebetteten Java-Code als auch durch Tag-Libs.

Aber auch die Struts-Konfiguration oder Hibernate-Mappingdateien sind im weiteren Sinne textuelle DSLs. Sie beschreiben in Form von XML einen Aspekt der Funktionalität eines Systems und zeigen, dass nicht nur das Verhalten eines Systems sich gut textuell beschreiben lässt sondern auch strukturelle Aspekte.

Außerdem zeigen diese beiden Beispiele, dass sich Code für textuelle DSLs gut generieren und transformieren lässt: Sie werden oft aus UML-Modellen generiert, und genau so kann man Quelltexte für eigene DSLs im Prinzip leicht aus anderen Modellen erzeugen.

Erster Schritt: Abstraktionen

Vergessen wir für einen Augenblick einmal Java und betrachten die verschiedenen Teile der Geschäftslogik unvoreingenommen. In ihnen geschehen sehr verschiedene Dinge, und man benutzt sehr unterschiedliche Begriffe, um sie zu beschreiben. Wir wollen für die Domänen ja jeweils eine möglichst gut passende Programmiersprache (DSL) finden, und dazu ist der erste Schritt, dass wir ein Grundverständnis dafür bekommen, worum es jeweils geht.

Der zentrale Begriff der Plausi-Prüfung ist der *PlausiCheck*, der eine *Bedingung* dafür definiert, dass die Daten einer *Entität* gültig sind. Im Falle einer nicht erfüllten Bedingung wird eine *Meldung* erzeugt; Meldungen können *Fehler* oder *Warnungen* sein. Und die PlausiChecks liegen in *Gruppen* vor (z.B. alle Prüfungen für eine Forderung), die immer gemeinsam überprüft werden und ggf. eine Liste von Meldungen zurückliefern.

Ganz anders bei den Rechenregeln: Hier gibt es *Berechnungsvorschriften*, die auf den Daten von Entitäten operieren. Sie bestehen aus *Rechenoperationen*, und sie können sich gegenseitig *aufrufen*. Wir wollen außerdem *Fallunterscheidungen* haben und spezielle *Rundungsregeln* für *Geldbeträge* unterstützen.

Damit eine Sprache eine der Domänen gut und einfach beschreiben kann muss sie diese zentralen Abstraktionen möglichst direkt und unkompliziert unterstützen.

Zweiter Schritt: Semantik

Nachdem wir jetzt eine Vorstellung von den zentralen Abstraktionen haben, können wir daran gehen, ihre genaue Bedeutung (oder Semantik) festzulegen.

Bei der Plausi-Prüfung soll man auf eine Entity eine Gruppe von PlausiChecks anwenden können, die man über ihren Namen auswählt. Als Ergebnis erhält man zwei Listen, eine mit Warn-Meldungen und eine mit Fehlermeldungen der jeweils fehlgeschlagenen Checks. Jeder Check ist ein Ausdruck, der einen boolean liefert; ein Ergebnis von *false* bedeutet, dass die Entity plausibel ist, *true* bedeutet eine fehlgeschlagene Prüfung und erzeugt eine

Meldung. Wenn ein Ausdruck beim Navigieren durch den Objektgraphen über eine null-Referenz läuft, dann ist das erlaubt, und der Wert des Ausdrucks ist *null*.

Rechenregeln sind Funktionen, die jeweils eine Liste von Parametern sowie einen Ergebnistyp haben. Ein Aufrufer sucht über ihren Namen diejenige Rechenregel aus, deren Wert er erhalten will. Das Ergebnis einer Rechenregel wird jeweils durch einen Ausdruck definiert. Fallunterscheidungen bedeuten, dass abhängig von einer Bedingung ein Ausdruck aus einer Liste ausgewählt und ausgewertet wird. Es gibt also keine Abfolge von Befehlen sondern nur Ausdrücke. Es gibt einen gesonderten Datentyp für Geldbeträge, und Geldbeträge werden immer dann automatisch auf zwei Nachkommastellen gerundet, wenn ein Geldbetrag aus einer Regel zurückgeliefert wird. (Das ist etwas künstlich, es mag aber exemplarisch für die eher komplizierteren Anforderungen echter finanzmathematischer Systeme stehen.)

Damit haben wir eine Vorstellung davon, was eine DSL zur Beschreibung von Plausi-Prüfungen oder Rechenregeln tun sollte. Dieses erste Verständnis wird natürlich im Laufe des Projektes wachsen oder sich ändern, aber es ist gut, die Kernabstraktionen für den klar umrissenen Bereich einer DSL schon am Anfang recht gut verstanden zu haben. An dieser Stelle können wir die Sprachkonzepte anhand von Szenarien überprüfen: Wir nehmen konkrete Beispiele, für die wir die Sprachen gerne verwenden würden, und sehen uns im Detail an, wie die Beispiele in der DSL aussehen würden.

Bisher haben wir einige Stunden (bis Tage, wenn das System groß und die Domänen noch unbekannt sind) mit Überlegungen auf Architekturebene verbracht, und wir haben noch keine Zeile Code geschrieben.

Dritter Schritt: Syntax

Die am weitesten verbreiteten Syntax-Stile für DSLs sind UML und XML: UML hat sich als Basis für das Generieren von Datenmodellen und Komponentenbeziehungen bewährt, und viele leistungsfähige Konfigurationsdateien (z.B. Struts oder Hibernate) bestehen aus XML.

Sowohl UML als auch XML sind in erster Linie zum Beschreiben von statischen, strukturellen Aspekten von Systemen. Zum Beschreiben des Verhaltens eines Systems benötigt man dagegen Ausdrücke, also Bestandteile, die beschreiben, wie sich ein Wert aus anderen Werten ergibt.

Hier zeigt sich die Gefahr, sich zu früh auf eine Syntax festzulegen. Man kann zwar im Prinzip sowohl in UML als auch in XML Ausdrücke einbetten, aber die primäre Abstraktion des Modells ist damit unwiderruflich strukturell. Das verstellt leicht den Blick auf die vielfältigen Ausdrucksmöglichkeiten einer textuellen Sprache.

Auf der Grundlage unseres Verständnisses der Domänen können wir jetzt daran gehen, eine Syntax zu definieren. Es ist wichtig, dass eine DSL eine einfache und intuitive Syntax hat, aber noch wichtiger ist eine klare und durchgängige Semantik. Wenn man sich erst

einmal auf eine Syntax festgelegt hat, dann denkt man über die Domäne leicht nur noch anhand dieser Syntax nach und übersieht Konzepte, die es in der Syntax noch nicht gibt.

Aber wir haben ja inzwischen ein gutes Grundverständnis von Plausi-Prüfugen und Rechenregeln, so dass wir an die Syntax gehen können.

```
// LISTING 1
PlausiGruppe SchuldnerGui <Schuldner> {
  Fehler  "namePflichtfeld": name == null;
  Fehler  "nameLaenge": name.length <3 || name.length > 50;
  Warnung "hausnummer": adresse.hausnummer == null;
  Warnung "aktivaPassiva“: bilanz.summeAktive !=
    bilanz.summePassiva;
}
PlausiGruppe SchuldnerB2B <Schuldner> {
  Fehler  "namePflichtfeld": name == null;
  Warnung "vornamePflichtfeld": vorname == null;
}
```

Listing 1 zeig exemplarisch unsere Syntax für Plausiprüfungen. Jede Plausigruppe beginnt mit dem Schlüsselwort *PlausiGruppe*, gefolgt vom Namen der Gruppe und der Entity, die überprüft wird. Danach folgt eine Liste der einelnen PlausiChecks, die jeweils mit dem Schlüsselwort *Fehler* oder *Warnung* anfangen. Danach kommt die Fehlermeldung und, durch Doppelpunkt getrennt, der Ausdruck, der überprüft werden soll.

```
// LISTING 2
double ortsFaktor (Schuldner s):
  switch (s.adresse.stadt) {
    case "Pusemuckel": 0.5;
    default: 0.8;
  };

... // viele, komplizierte Regeln!

betrag restWert (Forderung f):
  ortsFaktor (f.hauptSchuldner) * f.nominalwert;
```

In Listing 2 steht ein Beispiel für die Syntax der Rechenregeln. Sie ist an die Syntax von Java-Methoden angelehnt und beginnt mit dem Rückgabewert einer Regel, gefolgt von ihrem Namen und der Parameterliste. Anschließend steht ein Ausdruck, der den Wert der Regel festlegt.

Die erste der beiden Regeln enthält ein Beispiel für die Syntax einer Fallunterscheidung: In unserer DSL kann man einen switch auch über einen String als Parameter durchführen. Und die zweite Regel illustriert den einfachen Umgang mit Geldbeträgen: Man kann einen beliebigen double-Wert als *betrag* zurückgeben, und er wird dabei automatisch auf zwei Nachkommastellen gerundet.

Vierter Schritt: Integration in die Anwendung

Jetzt, wo wir eine konrete Vorstellung von den textuellen DSLs haben, brauchen wir noch einen Parser für sie sowie eine Laufzeitumgebung, um sie auszuführen. Der Parser lässt sich ohne großen Aufwand mit einem Parser-Generator wie JavaCC [5] oder ANTLR [6] erzeugen.

Außerdem braucht man noch einen Interpreter. Das klingt zunächst vielleicht kompliziert und nach viel Aufwand, aber ein Interpreter ist ja nichts anderes als eine Reihe von Java-Klassen, die die fachlichen Operationen der DSL enthalten. Eine detaillierte Anleitung dazu sprengt den Rahmen dieses Artikels, aber auf Basis eines Parser-Generators ist ein Interpreter schnell geschrieben und lässt sich ohne großen Aufwand pflegen.

Schließlich brauchen wir noch eine Java-Schnittstelle, durch die der Rest der Anwendung die Funktionalität aufrufen kann, die in der DSL definiert ist. Listing 3 zeigt, wie diese Schnittstelle für den Plausi-Prüfer aussehen kann; bei der Prüfung werden die Fehler und Warnungen an die übergebenen Listen angehängt.

Hinter dem Interface liegt eine konkrete Implementierung, die bei der Initialisierung eine Datei mit Plausi-Definitionen einliest und sie dann bereitstellt. Der Rest des Systems kann sich jetzt von einer Factory – oder per Dependency Injection – eine konkrete Instanz des PlausiCheckers holen und mit ihrer Hilfe beliebige Überprüfungen durchführen.

```
// LISTING 3
public interface PlausiChecker {
  void check (Object entity, String checkName,
    List fehler, List warnings);
}
```

Lohnt sich das?

Nachdem wir jetzt an einem Beispiel gesehen haben, wie textuelle DSLs in der Praxis aussehen, können wir uns der Frage widmen, ob sich der Aufwand lohnt. Die Frage ist berechtigt, denn eine DSL bedeutet Aufwand für ein Projekt.

Zunächst analysiert und formalisiert man die Domänen, wobei die dabei gewonnenen Einsichten auch ohne DSL nützlich sind. Vor allem aber entwirft man eine Syntax, schreibt einen Interpreter (oder Compiler) dafür, und schließlich schreibt und pflegt man Quelltexte in einer neuen Sprache, mit der man weniger vertraut ist als mit Java und für die es weniger gute Toolunterstützung gibt (Stichwort Refactoring-Unterstützung). Welche Vorteile stehen dem gegenüber, die den Aufwand rechtfertigen?

Zunächst einmal gilt natürlich, dass sich der Aufwand nicht immer lohnt. Insbesondere bei kleinen Projekten kann der Aufwand den Nutzen leicht übersteigen, und man muss im Einzelfall abwägen. Aber der Einsatz von textuellen DSLs zur Beschreibung von Geschäftslogik bringt eine Reihe von Vorteilen gegenüber dem Ausprogrammieren z.B. in Java.

Der wichtigste Vorteil besteht darin, dass man mit der DSL auf der gleichen Abstraktionsebene programmiert, auf der die Anforderungen definiert sind. Der Code ist kompakt, er ist leicht zu verstehen und funktioniert so, wie man es erwartet.

Bis zu einem gewissen Grad kann man das auch durch ein gutes Schneiden von Methoden und Klassen erreichen. Speziell bei fein granularen Konzepten stößt man dabei aber oft an einen Punkt, wo man mit der Syntax von Java in Konflikt kommt. Ein typisches Beispiel ist der PlausiChecker, in dem man quer durch den Objektgraphen navigieren kann, ohne sich über NullPointerExceptions Gedanken zu machen. Ein anderes Beispiel ist die Möglichkeit, bei den Rechenregeln einen switch() über Strings durchzuführen oder eine kompakte Spezialsyntax für Rechenoperationen über die Bilanzdaten mehrere Jahre hinweg einzuführen.

Aus dieser konzeptionellen Nähe zu den Anforderungen ergibt sich, dass textuelle DSLs eine gute Basis für die Kommunikation mit Fachleuten sind. Wir haben oft anhand der DSL-Quelltexte über unklare Punkte in den Anforderungen diskutiert, und teilweise haben sogar „nichtprogrammierende" Fachleute die Quelltexte geschrieben und Bugs in ihnen gefixt.

Außerdem kann man textuelle DSLs teilweise so „aufbohren", dass man aus ihnen z.B. Hilfetexte generieren kann. Oder man kann z.B. in den Rechenregeln Informationen für fachlich motivierte Datenmigrationen unterbringen.

Und schließlich kann man z.B. mit Eclipse relativ leicht eine IDE für eigene DSLs bauen, die schon während des Editierens fachliche Constraints überprüft sowie Syntax-Highlighting und andere Features einer modernen Entwicklungsumgebung bietet. Der Aufwand für eine solche IDE lohnt sich natürlich nur für vergleichsweise große Projekte, dafür bietet sie dann aber auch einen erheblichen Mehrwert.

DSLs, Agilität und JUnit-Tests

Textuelle DSLs erwecken, wie überhaupt modellgetriebene Softwareentwicklung, leicht den Eindruck, dass man mit ihnen wasserfallartig vorgeht und sich den Weg zu späten Kurskorrekturen im Sinne agiler Softwareentwicklung verbaut. Schließlich investiert man vorab Zeit in die Analyse und baut dann noch einen Interpreter, bevor man wieder unmittelbar an Kundenanforderungen arbeitet. Außerdem gibt es ja keine Gewähr, dass man bei der Vorabanalyse die Domäne „richtig und vollständig" versteht – eher im Gegenteil, die Erfahrung zeigt, dass Details oft erst „unterwegs" klar werden.

Das sind wichtige Bedenken, und man sollte auf keinen Fall einfach wild drauflos für alles und jedes neue DSLs einführen. Eine wichtige Voraussetzung für jede Art modellgetriebener Entwicklung ist, dass man zumindest ein Grundverständnis von der Domäne hat, die man abstrahieren will. Alles andere ist zumindest fahrlässig. Andererseits erfordert jede Art des Programmierens, dass man zumindest ungefähr versteht, worum es geht.

Der Aufwand für das Erstellen einer neuen DSL inklusive Interpreter liegt, etwas Übung vorausgesetzt, in der Größenordnung einiger Stunden bis Tage. Wenn die Domäne so klein ist, dass sich dieser Aufwand klar nicht lohnt, dann sollte man ihn natürlich nicht treiben.

Ansonsten erweisen sich in unserer Erfahrung die textuellen DSLs als bemerkenswert stabil über die Projektlaufzeit. Speziell am Anfang erweisen sich oft Erweiterungen der Sprachen als nützlich, aber in den allermeisten Fällen zerbrechen sie nicht die Kernabstraktionen der Sprachen und erfordern oft nicht einmal Anpassungen an bestehenden Quelltexten. In unserer Erfahrung erreichen sie typischerweise recht schnell einen stabilen Stand, der dann ziemlich robust selbst gegenüber radikalem Refactoring im Rest der Anwendung ist.

Ein wichtiger Bestandteil agiler Entwicklung mit DSLs sind auch automatisierte JUnit-Tests. Zum einen sollte man die Sprachbestandteile der DSL selbst durch JUnit-Tests abgesichern. Dabei hat es sich bewährt, kleine Quelltexte direkt als String-Konstanten in die JUnit-Testklassen einzubetten und aus diesen Stringkonstanten heraus zu parsen. Diese Tests stellen sicher, dass der Interpreter so funktioniert wie er soll.

Außerdem haben sich Black Box-Tests bewährt, die die „echten" DSL-Quelltexte verwenden, die mit dem fertigen System zum Einsatz kommen sollen. Diese Tests überprüfen, dass der DSL-Code z.B. die richtigen, tatsächlich gewollten Rechenregeln enthalten. Solche relativ grob-granularen Tests sähen genau so aus, wenn die Rechenregeln fest in Java programmiert wären (s. Listing 4).

```
// LISTING 4
public void testRestwert () {
  Forderung f = createTestForderung (...);
  RechenEngine re = RechenEngineFactory.create ();
  assertEquals (28450.35,
      re.getResult ("restWert", f), 0.00001);
}
```

Best Practices

Beim Entwickeln textueller DSLs ist das wichtigste Werkzeug ein Parser-Generator; JavaCC[5] und ANTLR [6] sind in der Java-Welt am weitesten verbreitet. Für ANTLR gibt es ein Eclipse-Plugin [7], das neben Syntax-Highlighting auch eine Integration in den Build-Mechanismus bietet.

Ein weiteres wichtiges Werkzeug beim Arbeiten mit textuellen DSL ist ein ganz normaler Texteditor. Suchen und Ersetzen mit regulären Ausdrücken ist ein leistungsfähiges Werkzeug, mit dem man erstaunlich viel tun kann. Und jeder, der das schon einmal mit einem UML-Werkzeug versucht hat, wird die Robustheit eines textuellen diff oder merge zu schätzen wissen.

Außerdem entwickelt die Firma JetBrains unter dem Namen „Meta Programming System" einen IDE-Baukasten für eigene, textuelle DSLs. Das Tool ist noch nicht released, sieht aber

vielversprechend aus [8]. Ein guter Überblick über die aktuellen Trends beim Tool Support für DSLs findet sich bei Martin Fowler [9].

Zu guter Letzt haben wir noch eine Liste mit Best Practices zusammengestellt, die sich beim Arbeiten mit textuellen DSLs bewährt haben.

Das erfolgreiche Erstellen guter DSLs beginnt beim Schneiden der Domänen. Es hat sich bewährt, für verschiedene Aspekte des Systems verschiedene DSLs zu verwenden, die sich möglichst wenig überlappen (also „orthogonal" sind). Die Versuchung ist oft groß, z.B. Plausiprüfungen in das Datenmodell aufzunehmen oder Reportdefinitionen in die Rechenregeln, aber die resultierenden DSLs sind einfacher, robuster und beständiger, wenn sie jeweils nur eine einzige Domäne abdecken müssen, die möglichst wenige und klar umrissene Schnittstellen zu anderen Domänen hat. Eine typische Schnittstelle ist z.B. die Definition der Entitäten – viele DSLs greifen darauf zurück.

Innerhalb der Domänen sind gut verstandene Kernabstraktionen das Wichtigste. Dabei ist es oft eine Hilfe, wenn man sich früh das Interface überlegt, durch das der Rest der Anwendung den DSL-Code aufruft. Was soll hineingereicht werden? Was soll zurückgeliefert werden? Außerdem sollte man sich für jede DSL auf möglichst wenige Abstraktionen beschränken und statt dessen versuchen, mit möglichst wenigen Konzepten alles Nötige zu beschreiben. Eine DSL mit wenigen gut kombinierbaren Kernkonzepten ist einfach und oft auch besonders ausdrucksstark als eine DSL.

Eine weitere Best Practice ist, sich zunächst auf möglichst wenige, einfache Sprachelemente zu beschränken, die möglichst unabhängig von einander sind. Auf diese Weise erhält man einen stabilen und einfachen Sprachkern, der die Chance hat, sich nicht zu ändern. Syntaktischer „Zucker" macht die Sprache unübersichtlicher und schwieriger zu ändern. Man könnte z.B. für die Plausi-Checks die Sprachelemente „minLength" und „maxLength" definieren, um die minimale und maximale Feldlänge von Strings zu überprüfen. Wenn man so für jeden Spezialfall die Sprachdefinition erweitert, bläht man aber schnell die Sprache auf, ohne wirklich etwas zu gewinnen. Wenn die DSL sich im Laufe der Zeit bewährt hat und stabil ist, kann man – vorsichtig und maßvoll – syntaktische Abkürzungen einführen, aber dabei ist weniger fast immer mehr.

Außerdem ist es oft hilfreich, in der DSL eine Möglichkeit vorzusehen, Java-Code aufzurufen. Man kann dazu z.B. ein Function-Interface im Stil von Listing 5a definieren. Wenn ein konkreter DSL-Quelltext dann ein Sprachelement braucht, das es noch nicht gibt – z.B. das aktuelle Datum – dann kann man das Feature als Java-Klasse implementieren (s. Listing 5b) und in den DSL-Quelltext einbinden und es anschließend verwenden – hier mit vorangestelltem Hochkomma (s. Listing 5c).

```
// LISTING 5a
public interface Function {
  Object getValue (Object[] params);
}
```

```
// LISTING 5b
public class Now implements Function {
  public Object getValue (Object[] params) {
    return new java.util.Date ();
  }
}

// LISTING 5c
ImportFunction now := meinpackage.Now;
...
Fehler  "zukuenftigerStichtag": stichtag > ’now ();
```

Zum Thema Performance empfehlen wir das übliche Vorgehen: Zunächst eine einfache, robuste und änderbare Lösung bauen und anschließend bei Bedarf messen und optimieren. Viele Systeme verbringen den größten Teil ihrer Zeit mit I/O, und es hat sich in unserer Erfahrung gezeigt, dass die geringere Performance durch interpretierten Code oft nicht ins Gewicht fällt. Wenn das doch der Fall sein sollte, dann kann man für eine DSL von einem Interpreter auf einen „Cross-Compiler“ umstellen, der aus der DSL Java-Code generiert.

Als letzte Best Practice empfehlen wir, sich gedanklich einen Baukasten an Sprachelementen zuzulegen, aus dem man nach Bedarf eine DSL zusammensetzt. Dabei ist es hilfreich, nicht nur objektorientierte Sprachen zu kennen, sondern auch funktionale oder logische Sprachen. Es gibt inzwischen eine Reihe von solchen Sprachelementen, die sich als leistungsfähig erwiesen haben und die sich gleichzeitig gut kombinieren lassen. Solche Sprachelemente sind z.B. Expressions, ein Typsystem, Polymorphie, aber auch Backtracking oder Lazy Evaluation. Der interessierte Leser sei auf [10] verwiesen, das viele Aspekte des Bauens von Programmiersprachen ausführlich behandelt.

Fazit

Textuelle DSLs sind zwar kein Allheilmittel, aber sie erlauben es, auf der Abstraktionsebene der Domäne zu programmieren. Der initiale Aufwand für das Definieren einer DSL inklusive Interpreter liegt typischerweise bei einigen Stunden bis Tagen, und anschließend ist der Aufwand für ihre Pflege typischerweise gering. Bei gut gewählten Kernabstraktionen bleiben sie im Gegenteil oft über längere Zeit stabil, selbst bei größeren Refactorings. Sie sind somit als stabiles Fundament für einzelne Teile eines Systems geeignet, auch wenn das Gesamtsystem sich in vielerlei Hinsicht weiter entwickelt.

References

[1] Kent Back, Cynthia Andres, eXtreme Programming Explained, Addison-Wesley, 2000

[2] Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003

[3] Thomas Stal, Markus Völter, Modellgetriebene Softwareentwicklung, dpunkt Verlag, 2005

[4] Markus Völter, Arno Haase, „Architektur ohne Hype“, Java Magazin 11/2005

[5] https://javacc.dev.java.net/

[6] http://www.antlr.org/

[7] http://antlreclipse.sourceforge.net/

[8] http://www.jetbrains.com/mps/

[9] http://www.martinfowler.com/articles/languageWorkbench.html

[10] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullmann, Compilerbau Teil 1, Oldenbourg, 1999