Jenseits des Tellerrands, Teil 5: Textuelle domänenspezifische Sprachen

Programmiersprachen zum Selbermachen

VON MARKUS VÖLTER UND ARNO HAASE

Softwareentwicklung ist weit mehr als Programmierung und Programmierung ist weit mehr als das bloße Beherrschen von Programmiersprachen und Werkzeugen. Die Artikel dieser Serie möchten wichtige Themen, aktuelle Fragen und auch grundsätzliche Probleme aufgreifen, beleuchten und „die gängigen Antworten" immer wieder in Frage stellen. Subjektivität ist dabei keineswegs verpönt, sondern das notwendige Salz in der technischen Einheitssuppe.
Unsere Branche leidet an einer Pervertierung der Prioritäten: In vielen Projekten setzen sich die klügsten Entwicklerköpfe überwiegend mit der Beherrschung technologischer Probleme auseinander. Dabei gießen sie immer wieder das bekannte Rad in neue Frameworks – die sie dann im nächsten Release wegen „noch besserer Technologie" neu entwickeln oder durch eine angesagte Open-Source-Bibliothek ersetzen. Das eigentliche Herz der Anwendung, die Fachlichkeit und die Domäne, wird der zweiten (meist unerfahrenen) „Programmiercharge" überlassen; dabei steckt genau hier der bleibende Geschäftswert. Eine Möglichkeit, um dem fachlichen Kern seinen gebührenden Platz einzuräumen, besteht darin, eine „fachliche Programmiersprache" zur Verfügung zu stellen: eine Domain Specific Language.
Johannes Link
(johannes.link@andrena.de)

Die meisten Softwaresysteme sind so groß, dass ein einzelner Entwickler sie nicht mehr 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 definiert oder informelle „Umgangssprache" sein, aber sie sind notwendig, damit ein System änderbar bleibt. Ohne dieses gemeinsame Vokabular birgt jedes Gespräch über das System die große Gefahr von unbemerkten Missverständnissen und die Qualität und Änderbarkeit leidet dementsprechend [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 an UML-Klassendiagramme, aus denen Infrastrukturcode für das Zusammenspiel von Komponenten generiert wird. Wir zeigen aber, wie man das Verhalten von Komponenten durch textuelle Sprachen beschreiben kann. Dabei werden diese Abstraktionen explizit und direkt nutzbar.

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-Services-Schnittstelle und fachliches Reporting spielt eine wichtige Rolle. Abbildung 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: Wir können uns 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

Jenseits des Tellerrands

Was bisher geschah und Sie zukünftig erwartet

- Teil 1: Das nächste Java?
- Teil 2: Typisierung in Java und darüber hinaus
- Teil 3: Warum gibt es eigentlich Softwareentwicklungsprojekte?
- Teil 4: Softwarearchitektur: eine kritische Bestandsaufnahme
- **Teil 5: Textuelle domänenspezifische Sprachen**
- Teil 6: Moderne System- und Abnahmetests

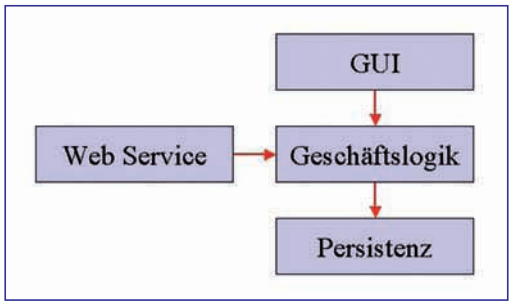

Abb. 1: Architektur des Beispielsystems

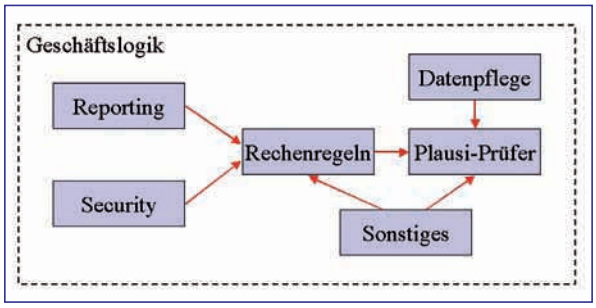

Abb. 2: Bestandteile der Geschäftslogik

Wesentlichen in drei Bestandteile (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, und 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 sind aus fachlicher Sicht das Herzstück des Systems. 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 umfangreich und kompliziert und werden fortlaufend weiterentwickelt.

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 Summen von Aktiva und Passiva übereinstimmen. Aber wenn durch ein Versehen die tatsächliche Bilanz fehlerhaft ist, dann muss das Programm die fehlerhafte Bilanz akzeptieren. Die Web-Services-Schnittstelle dagegen hat sehr viel schwächere Anforderungen an die Konsistenz von Daten. So kann man große Datenmengen ungeprüft importieren und eventuelle Inkonsistenzen anschließend manuell beseitigen. Der Rechenkern prüft die an ihn übergebenen Daten noch einmal

Anzeige

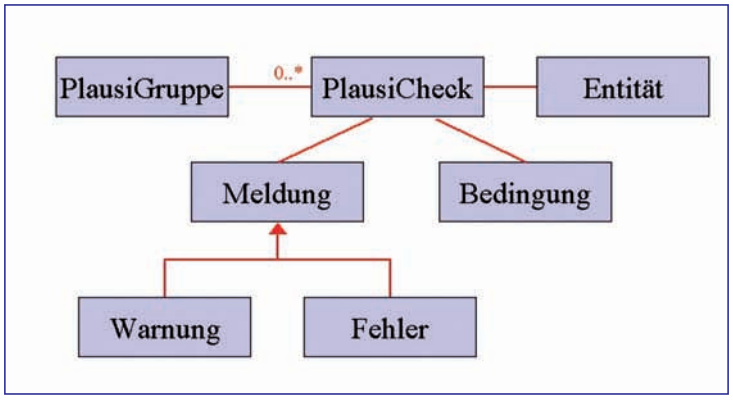

Abb. 3: Bestandteile der Plausi-Prüfungen

besonders streng, bevor er mit einer Berechnung beginnt.

Erster Schritt: Abstraktionen

Vergessen wir für einen Augenblick einmal Java und betrachten die verschiedenen Teile der Geschäftslogik unvoreingenommen. Sie tun verschiedene Dinge und man benutzt unterschiedliche Begriffe, um sie zu beschreiben. Wir wollen für die Domänen 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 (Abb. 3).

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

Damit eine Sprache eine Domäne gut und einfach beschreiben kann, muss sie deren zentrale 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- 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. Diese 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.)

Beispiele für textuelle DSLs

Textuellen DSLs hängt oft der Nimbus des Exotischen an, weil zurzeit nur wenige Projekte explizit ihre eigenen textuellen DSLs definieren. Es gibt aber eine Reihe von weit verbreiteten DSLs, die man oft nicht 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.
Noch näher an der Java-Welt liegen JSPs; sie sind ein Beispiel für eine DSL, die kompiliert statt interpretiert wird. Eines ihrer zentralen Features ist die Erweiterbarkeit, sowohl durch eingebetteten Java-Code als auch durch Tag-Libs.
Auch die Struts-Konfiguration oder Hibernate-Mapping-Dateien sind im weiteren Sinne textuelle DSLs. Sie beschreiben in Form von XML einen Aspekt der Funktionalität eines Systems und zeigen, dass sich nicht nur das Verhalten eines Systems, sondern auch strukturelle Aspekte gut textuell beschreiben lassen. Außerdem zeigen diese beiden Beispiele, dass man Code für textuelle DSLs gut generieren und transformieren kann: Sie werden oft aus UML-Modellen generiert und genau so kann man Quelltexte für eigene DSLs im Prinzip leicht aus anderen Modellen erzeugen.

UML, XML und DSLs

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 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 geeignet. Zum Beschreiben des Verhaltens eines Systems benötigt man dagegen Ausdrücke (Expressions), die darstellen, wie sich ein Wert aus anderen Werten ergibt.
Hier besteht 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.

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

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 inzwischen ein gutes Grundverständnis von Plausi-Prüfungen und Rechenregeln, sodass wir an die Syntax gehen können.

Listing 1 zeig exemplarisch unsere Syntax für Plausi-Prüfungen. Jede Plausi-Gruppe beginnt mit dem Schlüsselwort *PlausiGruppe*, gefolgt vom Namen der Gruppe und der Entity, die überprüft wird. Danach folgt eine Liste der einzelnen PlausiChecks, die jeweils mit dem Schlüsselwort *Fehler* oder *Warnung* anfangen. Darauf kommt die Fehlermeldung und nach einem Dop-

Listing 1

Syntax für Plausi-Prüfungen

```
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 2

Syntax für Rechenregeln

```
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;
```

Anzeige

pelpunkt der Ausdruck, der überprüft werden soll.

In Listing 2 sieht man 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. 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.

Der wichtigste Vorteil besteht darin, dass man mit der DSL auf der gleichen Abstraktionsebene programmiert, auf der die Anforderungen definiert sind.

Vierter Schritt: Integration in die Anwendung

Da wir jetzt eine konkrete Vorstellung von den textuellen DSLs haben, brauchen wir noch einen entsprechenden Parser sowie eine Laufzeitumgebung, um sie auszuführen. Der Parser lässt sich ohne großen Aufwand mit einem Parser-Generator wie JavaCC [5] oder ANTLR [6] erzeugen (siehe Kasten).

Außerdem braucht man noch einen Interpreter. Das klingt zunächst kompliziert und nach viel Aufwand, aber ein Interpreter ist nichts anderes als eine Reihe von Java-Klassen, welche 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. Der folgende Quelltext 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 der Klasse *PlausiChecker* holen und mit ihrer Hilfe beliebige Überprüfungen durchführen.

```
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, entwickelt 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 Tool-Unterstützung gibt (Stichworte Debugging und 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. Das kann so weit gehen, dass Domänenexperten direkt die DSL schreiben oder zumindest lesen, aber der Nutzen ist da, auch wenn ausschließlich Programmierer die DSL verwenden.

Bis zu einem gewissen Grad kann man das auch durch gutes Schneiden von Methoden und Klassen erreichen. Speziell bei feingranularen Konzepten stößt man dabei aber oft an einen Punkt, an dem 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 tex-

Tool-Support

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-Plug-in [7], das neben Syntax-Highlighting auch eine Integration in den Build-Mechanismus bietet.
Speziell für einfache DSLs kann man oft die Syntax auch so wählen, dass weit verbreitete Mittel wie XML, Key-Value-Paare und reguläre Ausdrücke ausreichen. Eine so entstandene Syntax ist zwar vielleicht weniger elegant als eine frei definierte, aber nichtsdestoweniger kann sie eine Domäne abstrakt beschreiben und das Schreiben des Parsers erfordert weniger Vorwissen.

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 ausgeliefert, sieht aber viel versprechend aus [8]. Ein guter Überblick über die aktuellen Trends beim Tool-Support für DSLs findet sich bei Martin Fowler [9].

tuelle 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 „nicht programmierende" 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 in den Rechenregeln Informationen für fachlich motivierte Datenmigrationen unterbringen und schließlich 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 Zeit in die Analyse und baut dann noch einen Interpreter, bevor man wieder unmittelbar an Kundenanforderungen arbeitet. Das kann man natürlich zu einem beliebigen Zeitpunkt in der Entwicklung tun, wenn man einen Teil entdeckt, den man gut abstrahieren kann. In jedem Fall investiert man aber in Infrastruktur, ohne dadurch unmittelbar sichtbare Anforderungen abzudecken. Außerdem gibt es keine Gewähr, dass man bei der Vorabanalyse die Domäne „richtig und vollständig" versteht – eher im Gegenteil, die Erfahrung zeigt, dass Details sehr 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 nicht lohnt, dann sollte man ihn natürlich nicht treiben.

Ansonsten erweisen sich nach 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. Die DSL erreicht typischerweise recht schnell einen stabilen Stand, der dann ziemlich robust selbst gegenüber radikalem Refactoring im Rest der Anwendung ist. Das liegt daran, dass die Kernabstraktionen einer Domäne, die ja in der DSL festgehalten werden, eben recht stabil sind.

Ein wichtiger Bestandteil agiler Entwicklung mit DSLs sind auch automatisierte JUnit-Tests. Zum einen sollte man die Sprachbestandteile der DSL selbst durch JUnit-Tests absichern. 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 Blackbox-Tests bewährt, welche 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 enthält. Solche relativ grobgranularen Tests sähen genau so aus,

Javamagazin IMPRESSUM

Verlag:
Software & Support Verlag GmbH

Anschrift der Redaktion:
Java Magazin
Software & Support Verlag GmbH
Kennedyallee 87
D-60596 Frankfurt am Main
Tel. +49 (0) 69 6300890
Fax. +49 (0) 69 63008989
redaktion@javamagazin.de
www.javamagazin.de

Chefredakteur: Sebastian Meyen
Redaktion: Nicole Bechtel, Alexander Neumann
Redaktionsassistenz: Sebastian Brück, Pierre Minnieur, Anna Pietras, Alexander Schmidt
Leitung Grafik & Produktion: Jens Mainz
Layout, Titel: Dominique Bergmann, Jessica Demirkaya, Melanie Hahn, Daniel Hartung, Jens Mainz, Sissy Mertens, Michel Michiels-Corsten, Maria Rudi
Illustration: Viktor Naimark
Tel. 0177 4366235

Autoren dieser Ausgabe:
Ulrike Böttcher, Daniel Cloutier, Dirk Frischalowski, Thilo Frotscher, Alexander Greif, Arno Haase, Oliver Ihns, Claudia Kellermann, Johannes Link, Sebastian Meyen, Jürgen Moors, Frank Müller, Denis Petrov, Chris Rupp, Broder Schümann, Alexander Schunk, Markus Völter, Christine Walz, Matthias Weßendorf, Christian Weyer, Tobias Wolf, Eberhard Wolff, Lars Wunderlich

Anzeigenverkauf:
Software & Support Verlag GmbH
Patrik Baumann
Tel. +49 (0) 69 6300890
Fax. +49 (0) 69 63008989
pbaumann@javamagazin.de

Es gilt die Anzeigenpreisliste Nummer 8

Pressevertrieb:
IPV Inland Presse Vertrieb GmbH
Tel.+49 (0) 40 237110, www.ipv-online.de/

Druck: PVA Landau
ISSN: 1619-795X

Abo-Service:
Software & Support Verlag GmbH
Tel. +49 (0) 69 6300890
Fax +49 (0) 69 63008989
www.javamagazin.de/service/

Abonnementpreise der Zeitschrift:

Inland:	12 Ausgaben	€ 69,–
Europ. Ausland:	12 Ausgaben	€ 79,–
Studentenpreis (Inland)	12 Ausgaben	€ 59,–
Studentenpreis (Ausland):	12 Ausgaben	€ 69,–

Einzelverkaufspreis:

Deutschland:	€ 6,50
Niederlande:	€ 7,80
Österreich:	€ 7,50
Schweiz:	sFr 12,70

Erscheinungsweise: monatlich

© 2005 Software & Support Verlag GmbH

Alle Rechte, auch für Übersetzungen, sind vorbehalten. Reproduktionen jeglicher Art (Fotokopie, Nachdruck, Mikrofilm oder Erfassung auf elektronischen Datenträgern) nur mit schriftlicher Genehmigung des Verlages. Jegliche Software auf der Begleit-CD zum *Java Magazin* unterliegt den Bestimmungen des jeweiligen Herstellers. Eine Haftung für die Richtigkeit der Veröffentlichungen kann trotz Prüfung durch die Redaktion vom Herausgeber nicht übernommen werden. Honorierte Artikel gehen in das Verfügungsrecht des Verlags über. Mit der Übergabe der Manuskripte und Abbildungen an den Verlag erteilt der Verfasser dem Herausgeber das Exklusivitätsrecht zur Veröffentlichung. Für unverlangt eingeschickte Manuskripte, Fotos und Abbildungen keine Gewähr. Java™ ist ein eingetragenes Warenzeichen der Sun Microsystems Inc.

Die Website des Java Magazins wird gehostet von Host Europe (www.hosteurope.de).

wenn die Rechenregeln fest in Java programmiert wären:

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

Best Practices

Zu guter Letzt haben wir noch eine Liste mit Best Practices zusammengestellt, die sich beim Arbeiten mit textuellen DSLs als sinnvoll erwiesen 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. Plausi-Prüfungen 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 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? Wenn zunächst nicht deutlich ist, wie dieses Interface aussieht, kann man mit einem ersten Entwurf beginnen, den man später per Refactoring verfeinert.

Eine weitere Best Practice ist, sich zunächst auf möglichst wenige, einfache Sprachelemente zu beschränken, die möglichst unabhängig voneinander sind. So 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 definieren:

```
public interface Function {
 Object getValue (Object[] params);
}
```

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

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

und in den DSL-Quelltext einbinden und es anschließend verwenden – hier mit vorangestelltem Hochkomma:

```
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 gleichzeitig gut kombinieren lassen. Solche Sprachelemente sind z.B. Expressions, ein Typsystem, Polymorphie, aber auch Backtracking oder Lazy Evaluation. Der interessierte Leser sei in diesem Kontext auf [10] verwiesen.

Fazit

Textuelle DSLs sind zwar kein Allheilmittel, aber sie erlauben es, auf der Abstraktionsebene der Domäne zu programmieren. Die anfängliche Investition für das Definieren einer DSL inklusive Interpreter liegt typischerweise bei einigen Stunden bis Tagen; anschließend ist der Aufwand für ihre Pflege meist gering. Gut gewählte Kernabstraktionen bleiben oft über längere Zeit stabil, selbst bei größeren Refactorings. DSLs sind somit als stabiles Fundament für einzelne Teile eines Systems geeignet, auch wenn das Gesamtsystem sich in vielerlei Hinsicht weiterentwickelt.

Arno Haase arbeitet als freiberuflicher Berater. Seine Tätigkeitsschwerpunkte liegen dabei auf Softwarearchitektur, modellgetriebener Entwicklung, agilen Methoden und Projekt-Reviews.
Markus Völter arbeitet als freiberuflicher Berater für Softwaretechnologie und -Engineering. Seine Schwerpunkte liegen dabei auf Softwarearchitektur, modellgetriebener Entwicklung und Middleware.

Links & Literatur

[1] Kent Beck, 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, 2005

[4] Markus Völter, Arno Haase: Architektur ohne Hype. Jenseits des Tellerrands, Teil 4: Software-Architektur – eine kritische Betrachtung, in *Java Magazin* 11.2005

[5] javacc.dev.java.net

[6] www.antlr.org

[7] antlreclipse.sourceforge.net

[8] www.jetbrains.com/mps/

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

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