Lars Corneliussen, Markus Völter

Domänenspezifische Sprachen als Mittel zur Abstraktion in der Software-Entwicklung

Kurze Rede, langer Sinn

Die Anforderungen an Software wachsen stetig. Steigende, tatsächliche Komplexität spiegelt sich häufig in einer großen, undurchdringlichen Codebasis wieder. Kein Wunder, dass der Ruf nach Abstraktion, nach Ausdrucksstärke lauter wird. Mit wenigen Worten mehr sagen; das ist das Ziel einer guten domänenspezifischen Sprache.

Domänenspezifische Sprachen sind formale Sprachen für die Softwareentwicklung. Sie drücken die Konzepte ihrer zugehörigen Domäne wesentlich einfacher und prägnanter aus als herkömmliche Programmiersprachen wie C# oder Java.

Dazu nutzen sie das Vokabular ihrer Fachdomäne und eine Notation, die Sachverhalte aus dieser Domäne geeignet darstellt.

Domänenspezifische Sprachen werden häufig im Kontext modellbasierter Softwareentwicklung (MDSD – Model Driven Software Development) und sprachorientierter Programmierung (LOP – Language Oriented Programming) eingesetzt.

Domänen

Schlägt man bei Wikipedia das Wort „Domäne“ nach, findet sich eine lange Liste von Wortdefinitionen, die sämtlich eine Art von Begrenzung beschreiben. Darunter Bezirk, Gut, Eigentum, Gebiet, Menge, Bereich und Feld.

Domänen im Sinne von DSLs sind eine bestimmte Menge von Sachverhalten und Konzepten. Die korrekte Abgrenzung ist einer der wesentlichen Faktoren für den Erfolg einer DSL. Der Spielraum einer DSL sollte so groß wie nötig und so klein wie möglich sein.

Fachliche und technische Domänen

Die Begriffe horizontal (allgemein) und vertikal (branchenspezifisch) klassifizieren das Einsatzgebiet von Software und sind deshalb zur Kategorisierung von Domänen eher irreführend.

Stattdessen werden technische und fachliche Domänen unterschieden. Eine fachliche Domäne bezieht sich auf die Konzepte, für die eine Software erstellt wird. Beispiele dafür sind ein Warenkorb, Banktransaktionen oder die Vertragsdefinition einer Versicherung.

Eine technische Domäne beschreibt einen Aspekt der Implementierung, des Designs, oder der Architektur eines Softwaresystems. Zum Beispiel die Programmkonfiguration, die Speicherverwaltung oder das Zerlegen von Zeichenketten.

Sprache

In natürlicher Sprache lässt sich prinzipiell alles mit dem Grundwortschatz ausdrücken. Da jedoch die Beschreibung spezifischer Probleme häufig wortreich und schwammig ausfällt, bildet sich in Branchen oder Berufsgruppen eine eigene Terminologie: das Fachvokabular. Dieses drückt dieselben Probleme wesentlich effizienter und eindeutiger aus.

DSLs übertragen diesen Fortschritt auf Programmiersprachen. Statt ein Problem mit einer universellen Programmiersprache zu beschreiben, nutzt eine DSL das Vokabular ihrer Zieldomäne. Da einem Computer keine intelligente Interpretation möglich ist, ergibt sich im Vergleich zu natürlichen Sprachen eine wesentliche Einschränkung. Die Sprache muss eindeutig und in einer von Computern lesbaren Form definiert sein. Solche Sprachen bezeichnet man als formale Sprachen.

Wie das Fachvokabular in der realen Welt ermöglichen DSLs eine einfachere und korrektere Beschreibung der Probleme und Konzepte einer Domäne.

Notation

So wie natürliche Sprachen verbal, schriftlich oder in Gebärden ausgedrückt werden, lassen sich auch Computersprachen auf unterschiedliche Weisen ausdrücken.

Man spricht im Wesentlichen von grafischen und textuellen DSLs. Für beide Notationen gibt es passende Einsatzgebiete. Diese Artikelreihe konzentriert sich jedoch auf textuell notierte DSLs.

Abgrenzung zu universellen Sprachen (GPLs)

GPLs (general purpose languages) sind universelle Computersprachen, mit denen sich Probleme jeder Domäne lösen lassen. Beispiele für solche Sprachen sind C, C# oder Java. Diese Sprachen sind Turing vollständig. Das heißt, dass man beliebige Algorithmen damit programmieren kann.

Ein Anhaltspunkt, an dem DSLs zu erkennen sind, ist ihr Abstraktionslevel. Eine DSL beschreibt gewöhnlich was passieren soll, nicht wie es genau passieren soll. Sprachen wie zum Beispiel Erlang oder Fortran, die spezielle (technische) Probleme wie Parallelität oder Berechnungsoptimierung fokussieren, werden häufig ebenfalls als DSLs bezeichnet. Ihre universelle Einsetzbarkeit verbietet jedoch diesen Schluss.

Auch echte DSLs werden häufig zu komplex, weil sie zu viele Eventualitäten abdecken. Das YAGNI-Prinzip (You Ain’t Gonna

Need It) hilft bei der richtigen Abgrenzung der Problemdomäne. Je strikter sie abgegrenzt ist, desto einfacher und effektiver wird die DSL.

Sollte an einer Stelle in einer DSL volle Flexibilität nötig sein, ist es besser für solche Ausnahmen einen Ausstieg in eine .NET-Bibliothek zu wählen. Alternativ können auch Skriptsprachen wie IronRuby oder IronPython angesprochen werden.

Bei DSLs unterscheidet man weiter interne (oder eingebettete) und externe DSLs.

Interne DSLs

Interne DSLs sind solche, die in eine andere Sprache eingebettet werden. Besonders beliebte Hostsprachen (Gastgebersprachen) dafür sind erweiterbare Sprachen wie XML, UML, Ruby, oder speziell in der .NET-Welt auch Boo.

Beispiele für interne DSLs in .NET sind XAML für WPF oder Binsor (Boo-DSL zur Konfiguration eines Windsor Containers).

In C# und Java spricht man von „Fluent Interfaces“, wenn der Aufbau des Codes an eine DSL erinnert. Ob man diese Form wirklich DSLs nennen kann ist strittig. C# und Java sind syntaktisch nicht wirklich erweiterbar, weswegen man bei der konkreten Syntax von DSLs wenig Einflussmöglichkeiten hat.

Der Vorteil interner DSLs ist die etwas geringere Einstiegshürde. Parser und Compiler sind bereits vorhanden und werden mittels Metaprogrammierung um weitere Sprachkonstrukte ergänzt. Bestehendes Wissen über die Hostsprache kann genutzt werden.

Der wesentliche Nachteil ist, dass eine interne DSL nicht auf der grünen Wiese entsteht, sondern auf die Konzepte der Hostsprache aufbaut. Dies schränkt die Freiheit im Sprachdesign ein. Interne Sprachen sind immer eine Erweiterung (wenn auch in Form von Beschränkungen), jedoch niemals eine Reduktion der Hostsprache. Weiter ergibt sich eine suboptimale Syntax, da sich das „Rauschen“ der Hostsprache nicht vollständig unterdrücken lässt.

Sollte im Gegenzug kein Wissen über die Hostsprachen bestehen, muss dieses zuerst erarbeitet werden. Bei zum Beispiel UML oder Ruby bedeutet dies einen nicht unerheblichen Lernaufwand.

Externe DSLs

Wenden wir uns also den „echten“ DSLs, genannt „externe DSLs“ zu. Diese haben eine frei definierte, für die Domäne passende Syntax.

Externe domänenspezifische Sprachen sind nichts Neues in der Softwareentwicklung. Dafür finden sich zahlreiche Beispiele, unter denen Reguläre Ausdrücke (RegEx) wohl das berühmteste ist. Auch bei SQL, CSS und XPath handelt es sich um externe DSLs. Diese DSLs adressieren jedoch meist technische Domänen für bestimmte, weitverbreitete Zielplattformen.

DSLs in Kundenprojekten zu erstellen und damit sowohl fachliche Aspekte als auch sehr konkrete technische Gesichtspunkte einer Implementierung zu adressieren ist jedoch erst seit wenigen Jahren in Mode. Dies lässt sich damit erklären, dass Werkzeuge zur effizienten Implementierung und Verwendung von DSLs erst seit kurzer Zeit verfügbar sind und sich teilweise noch im Betastadium befinden.

Externe, Interne und GPLs im Vergleich

Versenden einer E-Mail per C# (GPL):

```
var m = new
MailMessage("sender@example.org"
, "recipient@example.org");
m.Subject = "Dies ist der
Betreff";
m.Body = "Hier ist die
Nachricht.";

SmtpClient smtp = new
SmtpClient("smpt.example.org",
25);
smtp.Send(m);
```

Ein in XML eingebettetes DSL-Programm zur Versendung von E-Mails könnte wie folgt aussehen:

```
<sendmail
  from=“sender@example.org“
  to=“recipient@example.org“
  subject=“Dies ist der Betreff“
>
  Hier ist die Nachricht.
</sendmail>
```

Der deklarative Ansatz mit Xml ist schon deutlich lesbarer als die C#-Variante. Das „wie“ wird von der Verarbeitenden Software ausgeführt.

Eine Fluent-API mit C# könnte folgendermaßen aussehen:

```
Mail.From(“sender@example.org“)
  .About(“Dies ist der Betreff“)
  .With(„Hier ist die
Nachricht.“)

.SendTo(„recipient@example.org“)
;
```

Hierbei handelt es sich um ganz normalen objektorientierten C#-Code. Es kommt lediglich ein etwas angepasstes API-Design zum Einsatz. Durchaus eine valide Möglichkeit, der Unübersichtlichkeit in GPLs ein Schnippchen zu schlagen.
Wesentlich kompakter und hübscher ist jedoch folgende externe DSL:

```
from sender@example.org about
„Dies ist der Betreff“ send
„Hier ist die Nachricht“ to
recipient@example.com
```

Sämtliche Redundanz ist beseitigt; die Notation konzentriert sich auf die Essenz einer E-Mail.

Die Syntax wirft jedoch weitere Fragen auf: Sind Umbrüche im Mailtext erlaubt und wie werden sie kodiert? Was, wenn der Betreff Anführungszeichen enthalten soll? Im weiteren Verlauf des Artikels wird noch näher auf den Prozess des Sprachdesigns eingegangen.

Motivation

Jahrelange Projekterfahrung zeigt, dass sich der Initialaufwand für die Implementierung einer DSL inklusive der benötigten Verarbeitungswerkzeuge schon bei mittelgroßen Projekten auszahlt.

Zusätzlich dazu kann die Qualität der produzierten Software maßgeblich verbessert werden. DSLs unterstützt die Einhaltung von allgemein anerkannte Prinzipien der Softwareentwicklung, wie Separation of Concerns oder Don’t Repeat Yourself.

Komplexität verbergen

Das wohl wesentliche Ziel von DSLs ist es, Komplexität zu verbergen. Verbergen, weil tatsächliche Komplexität sich nicht weg diskutieren lässt. Der einzige Weg, wirkliche Komplexität beherrschbar zu machen ist die Zerteilung in überschaubare Happen von Software.

Abstraktion von Plattform und Technik

In herkömmlichen Anwendungen wird der Code, der tatsächlich Fachlichkeit beschreibt, von

schematischem Architekturcode ertränkt. Zwischen der Deklaration von Namespaces, Klassen, Eigenschaften und Methoden, Logging, Fehlerbehandlung, und Sicherheitsabfragen finden sich verstreut kleine Fragmente fachlicher Logik.

Die Trennung von Fachlichkeit und Technik sorgt für einen ersten Zugewinn an Verständlichkeit. Eine DSL beschreibt die Fachlichkeit, während die Kopplung zur konkreten Programmiersprache, zur Plattform und zu einzusetzenden Frameworks so spät wie möglich, meist im Interpreter oder Codegenerator erfolgt. Hier unterstützen DSLs die SoC (Separation of Concerns).

Dies vereinfacht das isolierte Aktualisieren einzelner Komponenten oder ermöglicht die Migrationen auf eine andere Zielplattform ohne das DSL-Programm anpassen zu müssen.

Redundanz vermeiden

Eines der wichtigsten Prinzipien der Softwareentwicklung ist DRY (Don't repeat yourself). Die Doppelterstellung und –Pflege von Logik ist aufwändig und anfällig für Fehler. Allerdings lässt sie sich häufig nicht vermeiden, besonders wenn unterschiedliche Komponenten und Plattformen im Spiel sind. Zum Beispiel wenn in einer mehrschichtigen Applikation auf dem Client Validierungen oder Berechnungen vorgezogen werden sollen, um den Server zu entlasten. Der Server wird sich nicht auf den Client verlassen wollen. Meist werden nicht die gleichen technischen Klassen auf Server und Client verwendet. Eine Codedopplung ist dann nicht zu vermeiden. Unter Einsatz von Validierungsregeln in DSL-Pogrammen, können diese durch verschiedene Adapter oder Codegenerierung eine beliebige Anzahl von technischen Ausführungen desselben Konzeptes bereitstellen.

Domänenexperten einbinden

Eine DSL fokussiert die Fachlichkeit einer Problemdomäne. Sie sollte zusammen mit den Experten der Domäne entwickelt werden, damit sie auch von diesen mindestens gelesen und wenn möglich sogar geschrieben werden kann.

Die Diskussionsgrundlage bei DSLs ist nicht frei interpretierbarer Prosatext, sondern ausführbarer Code in der Sprache der Domänenexperten.

Es bleibt die Aufgabe eines Softwarespezialisten, bei der Formalisierung der Gedanken zu helfen und die Strukturen dafür zu schaffen.

Die Angst, Entwickler würden durch die Nutzung von DSLs durch Domänenexperten überflüssig, ist unbegründet, da die DSLs und die nachgelagerten Verarbeitungswerkzeuge durch Entwickler zu implementieren sind.

Bestandteile einer DSL

Wenden wir uns dem Erstellungsprozess für eine neue DSL zu. Je nach Belieben beginnt man mit einem Entwurf der konkreten oder der abstrakten Syntax. Im weiteren Verlauf des Sprachdesigns wird meist iterativ vorgegangen, das heißt die Definition der abstrakten Syntax geht Hand in Hand mit der Erstellung der konkreten DSL-Syntax und der verarbeitenden Werkzeuge.

Konkrete Syntax

Die konkrete Syntax ist das Herzstück einer externen textuellen DSL. In den allermeisten Fällen wird die Syntax mittels einer Grammatik formalisiert. Grammatiken zerlegen eine DSL in Sätze, Wörter und Zeichen. Dabei werden Steuerzeichen und freie Eingaben zu einer Struktur zusammengefügt. Die Backus-Naur-Form (BNF) ist die bekannteste Sprache für abstrakte Grammatikbeschreibungen. Die BNF kann man ebenfalls als eine DSL verstehen. Die Domäne ist die Grammatikbeschreibung für formale Sprachen.

Die erweiterte BNF, kurz EBNF, unter vielen BNF-Varianten die meistverwendete, ist in ISO-14977 standardisiert. Sie ist ausdrucksstärker als BNF, das heißt sie fügt zwar keine wirkliche Funktionalität hinzu, ist jedoch kürzer.

So lässt sich beispielsweise die konkrete Syntax einer Division formalisieren:

```
Division =  Zahl "/" Zahl;
Zahl = {Ziffer};
Ziffer = "0" | "1" | "2" | "3" |
"4" | "5" | "6"
       | "7" | "8" | "9";
```

Diese Regeln bedeuten:

- Eine Division besteht aus zwei Zahlen, die durch einen Divisor „/“ verbunden sind.
- Eine Zahl ist eine beliebige Anzahl von Ziffern.
- Eine Ziffer ist eine 0, oder eine 1, oder eine 2 usw.

Der Einfachheit halber sind hier alle natürlichen Zahlen mit beliebig vielen führenden Nullen erlaubt.

Syntaktisch gültige Eingaben wären „3/4“ oder auch „2/0“.

Semantik

Der Sinn (die Semantik) des Divisionszeichens „/“ ist für Menschen klar ersichtlich, er geht aus der Grammatikdefinition jedoch nicht formal hervor. Es gibt Ansätze, auch die Semantik einer Sprache zu formalisieren. Allerdings werden diese in der Praxis selten verwendet. Stattdessen liegt die Semantik in der Übertragung auf ausführbaren Code durch einen Interpreter oder Codegenerierung. Im Beispiel also dort, wo die Software tatsächlich eine Division der linken durch die rechte Zahl vornimmt.

Parser

Der Parsevorgang ist die Transformation von menschenschreibbaren zu maschinenlesbaren Strukturen. Parser haben eine lange Historie und sind seit je her ein wichtiger Teil der Softwareentwicklung. Die großen Sprachen wie C#, Java oder Visual Basic .NET verwenden handgeschriebene, für die Sprache optimierte Parser. Für textuelle DSLs ist es jedoch üblich, auf Parsergeneratoren wie ANTLR oder lex/yacc zurückzugreifen. Diese erfordern deutlich weniger Programmierarbeit, haben aber einige für den Anfang unwesentliche Einschränkungen.

Werkzeugketten diverser Hersteller bieten eigene, teils BNF-ähnliche Grammatikdefinitions-Sprachen zur Generierung von Parsern an.

Abstract Syntax (Metamodell)

Um die Verarbeitung von DSL-Programmen zu vereinfachen, wird ein Schema angefertigt, oder aus der

Grammatikdefinition abgeleitet. Dieses Schema, auch Metamodell genannt, formalisiert die abstrakte Syntax einer DSL. Das allgemein am meisten verbreitete Format für Metamodelle ist die XML Schema Definition (XSD).

In der Praxis wird XML in der Softwareentwicklung maßgeblich für Konfigurationsdateien und zum Datenaustauschformat verwendet. Für Modelle und Metamodelle haben die verschiedenen Toolhersteller meist eigene Formate. Der Versuch der OMG, mit MOF (Meta Object Facility) ein plattformübergreifendes Format für die Metamodellierung zu etablieren, ist nicht gelungen.

Trotzdem sind XML und XSD gut geeignet, das Konzept von Metamodell und Model zu verdeutlichen. Ein XSD-Metamodell für die obige Division könnte so aussehen:

```
<xsd:element name=“division“
type=“Devision“/>
<xsd:complexType
name=“Devision“>
  <xsd:attribute name=“zahlA“
type=“xs:number“ use=”required”
/>
  <xsd:attribute name=“zahlB“
type=“xs:number“ use=”required”
/>
</xsd:complexType>
```

Das bedeutet, dass für ein valides Modell ein Element *division* mit den Attributen *zahlA* und *zahlB* erwartet wird.

Abstract Syntax Tree (Modell)

Das Ergebnis des Parsens ist ein Modell, welches die Struktur des DSL Programms wiedergibt. Dieses Modell ist die Instanz der abstrakten Syntax und wird deshalb Abstract Syntax Tree, kurz AST, genannt. Der AST ist die Grundlage für die Weiterverarbeitung durch weitere Software.

Da ein AST gewöhnlich in-memory verarbeitet wird, bedarf er keiner konkreten textuellen Syntax. Trotzdem definieren verschiedene Toolhersteller Datenformate für Modelle.

Für den Zweck dieses Artikels lässt sich der AST jedoch am einfachsten in Xml darstellen.

Für die Beispieleingaben „3/4“ und „2/0“ sähe der AST wie folgt aus:

```
<division zahlA=“3“ zahlB=“4“/>
```

Und:

```
<divistion zahlA=“2“ zahlB=“0“/>
```

Beide Eingaben ergeben durch das Parsen einen AST, der dem definierten Metamodell für die abstrakte Syntax genügt.

Modellvalidierung

Ein Metamodell kann jedoch nur die korrekte Struktur, nicht aber eine die semantische Schlüssigkeit eines Modells sicherstellen. Deshalb ist es mehr als hilfreich, dem Metamodell weitere, inhaltlich orientierte Beschränkungen beizulegen.

In der herkömmlichen Programmierung ist dies mit statischer Codeanalyse vergleichbar. Das Beispiel „2/0“ ist semantisch ungültig. In diesem Fall könnte der Fehler auch behoben werden, indem die Grammatik eine Null rechts vom Divisor nicht akzeptierte. Insgesamt würden solche Regeln grammatische Definitionen aber unnötig aufblähen.

Auch für die Modellvalidierung hat die OMG im Rahmen der UML-Spezfikationen einen Standard definiert, der auch auf Metamodellebene verwendet wird: die Object Constraint Language (OCL). Für OCL existiert in verschiedenen Tools unterschiedlich gute Toolunterstützung.

Die Sprache *Checks* (ein Teil des openArchitectureWare Projektes) ist eine pragmatische Umsetzung von OCL. Hier eine Constraint die die Division durch Null verbietet:

```
context Division ERROR "Division
durch Null ist nicht erlaubt.":
  nameB == 0;
```

Hierbei bezieht sich der Kontext *Division* auf den im Metamodell definierten Typen und wird für alle Instanzen von *Division* in dem zu prüfenden AST angewendet.

Ausführung

Die Ausführung einer DSL ist das „wie“. Hier findet die konkrete Umsetzung der Konzepte auf eine bestimmte technologische Plattform statt.

Es gibt zwei verschiedene Weisen DSLs auszuführen: Interpretation und Codegenerierung.

Interpretation

Bei der Interpretation wird ein Modell von einem Programm geladen und abgearbeitet. Dabei wird das Modell des DSL-Programms zur Laufzeit einem Interpreter übergeben, der die darin benutzen Konstrukte auf konkrete Aktionen abbildet.

Ein in C# geschriebener Interpreter für die Division könnte folgendermaßen aussehen:

```
public class DivisionInterpreter
{
  private XPathDocument model;
  public
DivisionInterpreter(XPathDocumen
t model)
  {
    this.model = model;
  }

  public double Execute()
  {
    var division =
model.CreateNavigator().SelectSi
ngleNode("division");
    double numberA =
division.SelectSingleNode("@numb
erA").ValueAsDouble;
    double numberB =
division.SelectSingleNode("@numb
erB").ValueAsDouble;

    // hier ist die Semantik
versteckt
    return numberA / numberB;
  }
}
```

Ein weiteres Beispiel für einen Interpreter im .NET Framework ist die Windows Presentation Foundation. Aus XAML-Dokumenten, die nur beschreiben wie etwas aussehen soll, wird ein Objektmodell erstellt. Der Interpreter sorgt dafür, die Plattform dazu zu bewegen, das gewünschte anzuzeigen.

Interpretation gilt als die „sauberere“ und flexiblere Variante ein Modell auszuführen, ist jedoch auch die aufwändigere. Bei etwas gehobenen Anforderungen wird Interpreter-Code hoch-komplex und fehleranfällig. Weiter gibt es kaum Toolsupport für die Erstellung von Interpretern.

Codegenerierung

Die Alternative ist Codegenerierung. Hierbei werden aus Modellen Artefakte für eine Zielplattform generiert. Hauptsächlich Quellcode in C# oder VB.NET, der zusammen mit manuell geschriebenem Code kompiliert wird, um dann ausgeführt zu werden. Häufig werden jedoch auch andere Artefakte wie Datenbankskripte oder Konfigurationsdateien generiert.

Beispiele im .NET-Umfeld in denen Codegenerierung zum Einsatz kommt sind Webservice-Clients, die

aus WSDL-Definitionen generiert werden und das Entity Framework das aus einer grafischen DSL .NET-Klassen für die jeweiligen Entitäten generiert. Sogar C# als Sprache generiert IL-Code für anonyme Delegate und das yield-Statement.

Leider genießt das Genre das Image eines Flickenteppichs. Das mag daran liegen, dass generierter Code viel zu häufig schlecht strukturiert und hässlich formatiert ist. Zum Beispiel werden wie bei Linq-to-SQL hunderte Klassen in eine Datei abgelegt, oder der Code ist mit voll qualifizierten Referenzen übersät.

Generierter Code ist Bestandteil der Gesamtsoftware und sollte den gleichen Qualitätsmerkmalen genügen wie handgeschriebene Quelltexte. Das ist zwar anfänglich etwas mehr Aufwand, erspart aber später viel Frustration.

Die Qualität des erzeugten Codes liegt in der Hand derer, die den Generator erstellen. Damit werden die typischen Vorurteile gegenüber generiertem Code (nicht verständlich, nicht lesbar, nicht kommentiert, nicht eingerückt, schlechte Benamung) entkräftet und die Verantwortung für die Code-Qualität wird zurück auf die Entwickler der Generatoren übertragen. Beste Ergebnisse werden erzielt, wenn der Generierung eine vorläufige, handgeschriebene Referenzimplementierung vorausgeht.

Für die Division würde solch ein Prototyp wie folgt aussehen.

```
public class Division
{
  public static double
Dividiere()
 {
    double a = 10;
    double b = 5;
   return a/b;
 }
}
```

Ein DSL Programm mit dem Namen *MeineDivision* und dem Inhalt „3/4“ könnte dann in eine Klasse *MeineDivision* resultieren, in dem die Variable *a* den Wert 3 und Variable *b* den Wert 4 zugewiesen bekommt.

Das Beispiel soll nur den Unterschied zwischen Interpretation und Codegenerierung klar machen und ist keineswegs eine sinnvolle Anwendung für die Codegenerierung.

Übrigens: Ein Codegenerator kann ebenfalls als Interpreter betrachtet werden. Nur dass anstelle direkter Interpretation der Konzepte zur Laufzeit ein Umweg über eine universelle Sprache gemacht wird. Außerdem läuft die Interpretation in einem Codegenerator zur Entwurfszeit und ist damit leichter kontrollierbar als ein gewöhnlicher Interpreter, der zur Laufzeit ausgeführt wird.

Damit schlägt man nach aktuellem Technikstand zwei Fliegen mit einer Klappe. Die Werkzeugunterstützung für die Erstellung von Codegeneratoren ist gut, und die erstellten Artefakte können mit Unmengen bereits vorhandener Werkzeuge weiterverarbeitet werden.

Vergleich und Bewertung

Beide Ausführungsmuster haben ihre Daseinsberechtigung. Es ist jedoch wichtig zu wissen, wann welche Variante die bessere Wahl ist. Dabei sind sowohl die Anforderungen zur Laufzeit als auch die zur Entwurfszeit in Betracht zu ziehen.

Laufzeitanforderungen

Flexibilität zur Laufzeit: Ein Interpreter ist erheblich flexibler, da das Verhalten des Interpreters durch Umgebungsparameter beeinflusst werden kann. Wenn ein Modell erst zur Laufzeit erstellt wird, oder zur Entwurfszeit noch unbekannt ist, führt ebenfalls kein Weg an der Interpretation vorbei.

Performanz zur Laufzeit: Hier ist der generierte, statische Code deutlich stärker, da die Abläufe der einzelnen Modelle fest kompiliert sind. Je nach Anwendungsfall variiert der Unterschied von nennenswert bis vernachlässigbar. Die in modernen GPLs verwendet Just-in-time-Compilierung würde diesen Nachteil in Interpretern aufheben, ist aber für DSLs keine gängige Praxis.

Auch die **Startzeit** ist bei generiertem Code wesentlich kürzer. Der Ablauf ist vorbereitet und kompiliert. Ein Interpreter hingegen muss zuerst das Metamodell laden, das DSL-Skript parsen und in ein Modell laden und gegebenenfalls das Modell validieren. Diese Schritte können je nach Größe der Modelle und Metamodelle wesentliche Zeit in Anspruch nehmen.

Die **Beschaffenheit der Zielplattform** kann ein sofortiges Ausschlusskriterium für einen Interpreter sein. Es ist beispielsweise unpraktikabel einen DSL-Parser auf einem Mobiltelefon (Apple hat dies auf dem iPhone explizit verboten!) oder einem Steuergerät für Autos laufen zu lassen. Auch für verteilte Szenarien in denen Teile der Logik auf verschiedenen Plattformen, zum Beispiel Server und Client, laufen müssen, eignet sich die Codegenerierung besser.

Entwurfzeitanforderungen

Je einfacher und schneller der Erstellungsprozess und die Wartung vorhandenen Codes ist, desto produktiver kann ein Entwickler sein.

Komplexität: Generierter Code ist relativ leicht auf Korrektheit überprüfbar, da er immer nur den konkreten, im Modell beschriebenen Fall repräsentiert. Ein Interpreter hingegen muss so geschrieben sein, dass er mit der gesamten Ausdrucksmächtigkeit der Sprache umgehen kann. Zu beachten ist allerdings, dass im Falle der Codegenerierung diese Gesamtkomplexität im Codegenerator steckt. Wie bereits erwähnt: Codegeneratoren sind spezielle Interpreter.

Typkonformität: Soll eine DSL von anderen Komponenten aufrufbar sein, bietet die Codegenerierung mehr Komfort, da Typkonformität gewährleistet werden kann. Eine generierte Klasse kann direkt so benutzt werden, als wäre sie handgeschrieben. Bei einem Interpreter ist der Zugriff nur über Reflection-ähnliche Mechanismen möglich. Auch die typisierte Übergabe von im Modell definierten Parametern ist nur mit der Codegenerierung machbar.

Debugging: Komplexe Szenarien ohne einen Debugger zu erstellen ist sehr nervenaufreibend. Ein Interpreter zu debuggen ist auch nervenaufreibend, weil die gleichen Codeteile mit vielen unterschiedlichen Parametrisierungen durchlaufen werden. In generiertertem Code kann man einfach an der Stelle einen

Breakpoint setzen, die man unter die Lupe nehmen will.

Das Beste ist natürlich ein Debugger für die eigene DSL. Aber der Aufwand rechnet sich nur selten, da es bislang wenig Toolsupport dafür gibt.

Verifizierbarkeit und Lesbarkeit: Es ist zur Entwurfszeit nicht festzustellen, ob ein Interpreter sich mit den Modellen zur Laufzeit richtig verhalten wird. Es bleibt einem nichts anderes übrig, als ihn anhand von Beispielen zu testen und sich dann darauf zu verlassen, dass er produktiv fehlerfrei agiert. Generierter Code kann hingegen statisch analysiert werden. Dabei helfen Tools wie FxCop, NDepend oder auch ein klassisches Review durch Entwickler.

Persistenz

Gewöhnlicherweise werden DSL-Programme als Textdateien wie anderer Quelltext abgelegt. Damit sind sie einfach in Versionsverwaltungssysteme zu integrieren und als Team zu nutzen. Man kann sie mit jedem beliebigen Texteditor einsehen und bearbeiten.

Alternativ lassen sich die Texte auch als Modell (AST) abspeichern. Entweder in Dateien zum Beispiel als XML oder XMI, oder auch in relationalen Datenbanken. Die Herausforderung hierbei ist, die Modelle wieder auf die ursprüngliche konkrete Syntax zurückzuführen.

Tooling

Eine DSL zu erstellen und zu verwenden bedingt den Einsatz diverser Tools. Die konkrete Syntax erfordert einen Parser und die abstrakte Syntax einen Interpreter oder Codegenerator. DSLs sollen zumeist auch mit herkömmlicher Software integrieren.

Texteditor

Visual Studio + Plugins haben in den letzen Jahren die Produktivität beim Entwickeln mit GPLs extrem gesteigert. Zu den unabdingbaren Features gehören Syntaxhervorhebung, Code-Completion, Code-Navigation und Refaktorisierungshilfen.

Manche Tools generieren aus Grammatikdefinitionen zusätzlich zu Parserprogrammen auch Texteditoren, die einige der gewohnten Features auch für DSLs verfügbar machen.

Language Workbench

Sogenannte Language Workbenches zur Erstellung und Integration von DSLs sind in den Technologiewelten Java und eingebettete Software wesentlich weiter verbreitet als im .NET-Umfeld. Microsoft hat den Trend schon vor einigen Jahren erkannt. Die seit Visual Studio 2005 erhältlichen DSL Tools sind eine Lösung spezifisch für grafische DSLs, die per Generierung auf Code abgebildet werden sollen. Die Software Factories-Initiative hat es ebenfalls nicht geschafft, die breite Masse zu erreichen.

In der Modellierungsinitiative „Oslo“ sieht Microsoft die nächste Generation der Softwareentwicklung. „Oslo“ soll in Zukunft die modellbasierte Methodik in das gesamte Produktportfolio bringen, und so für höhere Produktivität und bessere Software sorgen.

Die Modellierungsplattform Microsoft „Oslo“

Oslo besteht im Wesentlichen aus drei Komponenten. Einem Repository, einem grafischen Modellierungswerkzeug Quadrant und einer Sprachfamilie M. Für die Erstellung von DSLs ist vor allem M interessant. Es besteht aus derzeit zwei Modellierungssprachen MSchema und MGrammar, sowie aus einem Datenformat MGraph. Weiter ist das Tool intellipad erwähnenswert. Es ist ein erweiterbarer Texteditor, der das Entwickeln von Grammatiken und Schemas sehr effizient unterstützt. Eine Einbindung in Visual Studio ist ebenfalls implementiert.

MSchema beschreibt Datenstrukturen, Funktionen und Constraints. MGrammar definiert Grammatiken, parst DSL-Skripte und integriert mit intellipad und Visual Studio zwecks Fehleranalyse und Syntaxhervorhebung.

Microsoft bietet auf dem Oslo Developer Center alles was man braucht, um mit dem Designen einer DSL zu beginnen. Der Oslo Januar 2009 CTP steht dort zum Download bereit und eine Menge Videos, Hands-on-labs, Präsentationen, und die öffentliche Spezifikation der Sprache M erleichtern den Einstieg in die Redmonder Modellierungswelt.

Microsoft scheint jedoch darauf zu setzen, Modelle nur zu interpretieren. Bei der Betrachtung der Oslo Videos oder der Lektüre der Artikel zu Oslo scheint es so, als würde Oslo nur im Kontext des Repositories und des geplanten Applikationsservers „Dublin“ Sinn ergeben. Dies ist jedoch ein Trugschluss.

Dotnetpro zeigt Ihnen in dieser Artikelreihe, wie Sie eine DSL mit Oslo entwerfen und verwenden, auch ohne sich von Repository, Quadrant & Co. abhängig zu machen.

Open Source Alternative

Doch muss man auf „Oslo“ warten um DSLs einsetzen zu können? Die Eclipse Foundation hat eine lange Tradition in der Modellierungswelt. Projekte wie EMF, Xtext und openArchitectureWare sind ausgereift, kostenfrei und in realen Projekten erprobt. Für .NET-Entwickler eine wunderbare Gelegenheit, über den Tellerrand hinaus zu schauen.

EMF beschreibt Metamodelle und Daten. Diese dienen als Grundlage sowohl für Interpreter, als auch für Codegeneratoren. Xtext ist eine Grammatikdefinitions-Sprache, die perfekt in Eclipse integriert ist. Fehleranalysen, Syntaxvervollständigung, Syntaxhervorhebung und sogar die Navigation durch die eigenen DSL Skripte über Gliederungsansichten sind möglich. Außerdem ist der Editor einfach zu erweitern, so dass das Entwickeln mit DSL Skripten schiere Freude aufkommen lässt.

oAW ist ein hervorragendes Framework zur Generierung von Quellcode. Beide Produkte sind ausgereift und in vielen Branchen getestet. Die Zielplattformen variieren von Mobile über Embedded zu Enterprise Java Applications und eben .NET-Code.

Dotnetpro zeigt Ihnen im nächsten Artikel dieser Reihe wie Sie mit Eclipse Xtext und oAW in Ihren Projekten Budget und Zeit sparen können.

Referenzen

Wikipedia-Artikel

- http://de.wikipedia.org/wiki/Modellgetriebene_Softwareentwicklung
- http://de.wikipedia.org/wiki/Erweiterte_Backus-Naur-Form
- http://de.wikipedia.org/wiki/Dom%C3%A4nenspezifische_Sprache
- http://en.wikipedia.org/wiki/Turing_complete

Werkzeuge zur Erstellung von DSLs

- http://msdn.microsoft.com/oslo
- http://www.xtext.org