Domänenspezifische Sprachen und Microsoft Oslo

Markus Voelter, Independent/itemis, voelter@acm.org
Lars Corneliussen, itemis, lars.corneliussen@itemis.de

Auf der PDC letzten Herbst hat Microsoft eine neue Plattform vorgestellt: Oslo. Dabei handelt es sich um Frameworks und Tools zur Arbeit mit Modellen und der Erstellung von domänenspezifischen Sprachen. In diesem Artikel möchten wir einen Überblick über Oslo geben, und es in das Umfeld der modellgetriebenen Softwareentwicklung und der DSLs

Was sind Domänenspezifische Sprachen

Statt zu versuchen, eine geschlossene Definition des Begriffs "Domänenspezifische Sprache" zu liefern, möchten wir DSLs mit einer Reihe von anderen Technologien und Ansätzen vergleichen, und die Unterschiede aufzeigen. Dadurch wird klar, was eine DSL ist.

DSLs vs. Programmiersprache. Mit einer Programmiersprache lassen sich beliebige Sachverhalte beschreiben und Probleme lösen, sofern es dafür Algorithmen gibt. Programmiersprachen sind daher sehr mächtig und allgemein anwendbar. Domänenspezifische Sprachen sind für die Beschreibung von Sachverhalten und Algorithmik einer bestimmten Fachdomäne optimiert. Daher sind DSLs in aller Regel erheblich einfacher und kleiner. Sehr oft sind sie auch nicht Turing-vollständig, sondern dienen nur dazu, domänenspezifische Konzepte präzise und Tool-verarbeitbar zu beschreiben. Durch Codegeneratoren oder Interpreter werden solche Modelle[1] dann ausführbar.

DSLs vs. XML. Man könnte argumentieren, dass sich mit XML prinzipiell auch beliebige Domänenspezifische Strukturen und Verhaltensweisen beschreiben lassen. In gewisser Weise handelt es sich bei einem XML Schema ja um eine Sprachdefinition. Der Haken bei der Sache ist allerdings, dass man die Notation (also die konkrete Syntax) nicht anpassen kann. Alles sind geschachtelte und sich gegenseitig referenzierende spitze Klammern. Dies hat zur Folge, dass man mit XML nicht wirklich für den Menschen gedachte Modelle/Programme beschreiben kann (auch wenn dies ursprünglich einmal die Idee von XML war). DSLs unterscheiden sich also vor allem dadurch von XML, dass sich eine beliebige (grafische, textuelle oder anderweitige) Notation definieren lässt.

[1] Im Zusammenhang mit DSLs verwenden wir die Begriffe „Programm“ und „Modell“ synonym. Es handelt sich einfach um einen „Satz“ der mit der DSL beschrieben ist. Anneke Kleppe hat dazu den Begriff des „Mogram“ definiert ☺

Auch domänenspezifische Validierungsregeln lassen sich besser integrieren als bei XML, wo man im großen und ganzen auf XML Schema beschränkt ist.

DSLs vs UML. Auch UML ist eine Modellierungssprache, und seit Version zwei auch so formal definiert, dass man daraus prinzipiell ausführbare Systeme generieren kann. Allerdings ist die UML nicht wirklich domänenspezifisch. Man könnte sagen, die Domäne der UML ist "Softwareentwicklung". Wie mit Programmiersprachen lassen sich damit prinzipiell beliebige Systeme beschreiben. Allerdings erreicht man nicht die Vorteile der Domänenorientierung (siehe nächster Absatz): Man kann die UML mittels Profilen zwar beliebig anpassen, in real existierenden Tools sind diese Anpassungsmöglichkeiten aber sehr beschränkt. Durch Profile lassen sich Konzepte hinzufügen und die Verwendung bestimmter Elemente verbieten. Der gesamte Sprachumfang wird jedoch dabei nicht verringert, sondern erweitert.

Vorteile?

Die Vorteile von Domänenspezifischen Sprachen liegen in ihrer Domänenorientierung. Mittels einer an den zu beschreibenden Sachverhalt angepassten Sprache lässt sich dieser Sachverhalt erheblich präziser, knapper, und semantisch reicher beschreiben. Dadurch wird eine automatische Verarbeitung der entsprechenden Modelle durch Interpretation oder Codegenerierung ermöglicht. Alles in allem lassen sich dadurch erhebliche Effizienzsteigerungen in der Softwareentwicklung erzielen.

Die Historie der Softwareentwicklung (und der Programmiersprachen insbesondere) ist durch immer weiter ansteigende Abstraktion von der Rechnerhardware charakterisiert. Beginnend mit Sprachen wie Assembler und C wurde die Implementierung von Algorithmik erheblich vereinfacht. Objektorientierte Sprachen hatten das Ziel, durch die Abstraktionen *Klasse,* und *Objekt, Vererbung* und *Assozation* die reale Welt abzubilden. Konzepte wie Polymorphismus und einige Designpatterns erlauben die Implementierung von Verhaltensabstraktion. Allerdings sind diese Sprachen immer noch allgemeingültig. Domänenspezifische Abstraktionen lassen sich nur mithilfe der Features in diesen Programmiersprachen implementieren (also als Bibliotheken und Frameworks). Die Sprache selber ist nicht an die Domäne anpassbar, das bedeutet in der Praxis, dass insbesondere die Notation und statische Prüfungen nicht veränderbar sind.

Im Laufe der Zeit sind weitere Formalismen zur Beschreibung insbesondere von Verhalten entstanden. Das bekannteste Beispiel sind Zustandsmaschinen. Diese formalisieren die Beschreibung von Verhalten auf eine ganz bestimmte Art und Weise, und falls das zu beschreibende Verhalten tatsächlich zustandsbasiert ist, lässt es sich mittels Zustandsmaschinen erheblich präziser und übersichtlicher beschreiben als einer allgemeinen Programmiersprache.

Diese Erkenntnis machen sich DSLs zu Nutze und richten die Sprache direkt an den Konzepten einer konkreten Zieldomäne aus. Hier einige Beispiele:

- Architektur-DSLs eignen sich zur Beschreibung von Softwarearchitekturen bestimmter Systeme oder Plattformen. Die

Konzepte der Sprache entsprechen den Architekturabstraktionen der Zielplattform. Typische Sprachelemente sind *Komponente*, *Interface*, *Nachricht*, *Datenstruktur*, *primitiver Typ*, oder *Namespace*.

- Geschäftsregel-DSLs werden verwendet um regelorientierte Geschäftslogik zu beschreiben, idealerweise in einer Form, zu der die Geschäftsexperten direkt beitragen können.
- Grafische, an Blockschaltbilder erinnernde Sprachen werden verwendet, um beispielsweise Regelalgorithmen in der Automobilindustrie zu beschreiben.
- Es gibt Versicherungen, die die Beschreibung der Berechnungsregeln der Versicherungsverträge mittels Domänenspezifischen Sprachen durchführen. Diese enthalten dann beispielsweise temporale Datentypen da sich sehr viele Größen in Versicherungsverträgen über die Zeit ändern (beispielsweise das Gehalt der versicherten Person). Neben diesen temporalen Datentypen bieten solche Sprachen dann auch Unterstützung für Arithmetik basierend auf diesen Datentypen.

Hier noch eine Liste von konkreten Vorteilen von DSLs. Wie oben schon erwähnt, lässt sich die Implementierung eines Systems durch Interpreter und Codegeneratoren zu weiten Teilen, manchmal auch komplett, automatisieren. Dies steigert offensichtlich die Effizienz der Softwareentwicklung. Ein weiterer Vorteil besteht darin, dass die Beschreibung von domänenspezifischen Verhalten komplett von der Implementierungstechnologie abgekoppelt werden kann. Dies führt zu besserer Wartbarkeit und leichterer Migration falls sich die Technologie ändert. Bei geeigneter konkreter Syntax ist es auch durchaus realistisch, dass Domänenexperten (also Nicht-Programmierer) direkt die Modelle/Programme beschreiben und damit eine erheblich konkretere Rolle in der Softwareentwicklung spielen.

Bestandteile von DSLs

Wie bei Programmiersprachen auch, muss man sich bei der Definition einer DSL mindestens um die folgenden Aspekte kümmern.

Abstrakte Syntax. Für das verarbeitende Werkzeug (beispielsweise Code Generatoren) ist die abstrakte Syntax relevant. Die abstrakte Syntax ist eine Datenstruktur die den semantisch relevanten Inhalt eines mit der DSL beschriebenen Programms repräsentiert. Interpreter oder Codegeneratoren traversieren Instanzen dieser Datenstruktur (das Modell, oder der Abstrakte Syntaxbaum (AST)) und führen der Semantik der Datenstruktur entsprechende Aktionen aus (beim Codegenerator ist dies die Ausgabe von semantisch äquivalentem Code in der Zielsprache).

Konkrete Syntax. Die konkrete Syntax ist die grafische oder textuelle Darstellung der abstrakten Syntax. Anwender die die Sprache nutzen interagieren mit der konkreten Syntax. Es ist daher essenziell dass diese für die entsprechende Zielgruppe geeignet ist. Oft werden Sprachen mit grafischer Syntax als Modellierungssprachen bezeichnet.

Bei Sprachen mit textueller Syntax wird ein Parser verwendet, um die konkrete Syntax auf die abstrakte Syntax abzubilden. Dazu kommt eine Grammatik zum Einsatz. Diese ist letztendlich eine formale Beschreibung, der Abbildungsregeln von der konkreten auf die abstrakte Syntax.

Semantik. Der dritte Aspekt einer Sprachdefinition ist deren Semantik, also die Bedeutung dessen was man mittels der konkreten Syntax hinschreibt, und mittels der abstrakten Syntax dem Tool zur Verarbeitung zur Verfügung stellt. Es gibt verschiedene Arten der formalen Semantikdefinition, in der Praxis werden diese allerdings kaum verwendet. Semantik wird entweder durch Dokumentation der Sprache in Prosa definiert, oder mittels Interpretern und Codegeneratoren. Wenn ein Codegenerator eine DSL auf eine bekannte Sprache mit bekannter Semantik abbildet (beispielsweise C#), so lässt sich daraus im Umkehrschluss die Semantik der DSL ableiten (das funktioniert natürlich nicht, wenn man die gleiche DSL auf zwei verschiedene Zielsprachen abbildet. Man bräuchte dann eigentlich eine Semantikdefinition gegen die man die beiden Generatoren prüfen kann). Um die Korrektheit eines Generators und die Semantik einer DSL in der Praxis festzuzurren, pflegt man ein möglichst vollständiges Beispielmodell mit, generiert davon Code, und schreibt dann Unittests die die Semantik verifizieren.

In der Praxis werden noch zwei weitere Aspekte benötigt. Dazu gehören Constraints, Interpreter und Codegeneratoren, sowie benutzerfreundliche Editoren für die konkrete Syntax.

Constraints. Bei den meisten DSLs gibt es Korrektheitsregeln, die sich mit gängigen Formalismen zur Beschreibung von abstrakter Syntax nicht definieren lassen. Beispielsweise müssen die Attribute einer Datenstruktur eindeutige Namen haben. Gleiches gilt sinngemäß auch für die Zustände in einer Zustandsmaschine. Constraints dienen dazu, genau solche Regeln sicherzustellen. Constraints sind boolesche Ausdrücke. Damit ein Modell als korrekt betrachtet werden kann, müssen alle Constraints zu *true* evaluieren.

Modellverarbeitung. Im hier betrachteten Zusammenhang dienen DSLs der Entwicklung von Software. In aller Regel bedeutet dies, dass die per DSL erstellten Programme/Modelle in existierende Laufzeitinfrastrukturen integriert werden müssen. Dafür gibt es prinzipiell zwei Vorgehensweisen: Interpreter und Codegeneratoren (es existieren auch Mischformen).

Interpreter. Ein Interpreter ist ein Programm welches den abstrakten Syntaxbaum (AST) eines Modells traversiert und der Semantik der Sprache entsprechende Aktionen ausführt. Ein einfacher Interpreter für einen Taschenrechner würde für einen AST der die Addition zweier Zahlen repräsentiert tatsächlich diese beiden Zahlen addieren.

Codegeneratoren. Codegeneratoren sind spezielle Interpreter, die als Ergebnis der Modelltraversierung semantisch äquivalenten Quellcode in einer existierenden Programmiersprache ausgeben (oder beliebige andere textuelle Artefakte, wie zum Beispiel Konfigurationsfiles). Dieser Code wird dann nachfolgend interpretiert oder seinerseits kompiliert.

Editoren. Eine weitere wichtige Klasse von Werkzeugen, die wir noch nicht erwähnt haben, sind Editoren. Deren Aufgabe ist es, den Umgang mit der

konkreten Syntax der DSL für den Anwender möglichst effizient zu gestalten. In aller Regel baut der Editor (mehr oder weniger direkt) einen AST auf. Bei Sprachen mit grafische Syntax geschieht dies in aller Regel direkt: wenn man beispielsweise einen Zustand in ein Zustandsdiagramm einfügt, so wird im entsprechenden Syntaxbaum auch direkt das entsprechende Element angelegt. Bei Sprachen mit textueller Syntax kommt im Rahmen des Editors der oben bereits erwähnte Parser zum Einsatz.

Werkzeuge zum Bau von DSLs

Schon vor längerer Zeit hat Martin Fowler den Begriff der Language Workbench definiert. Der Begriff lässt sich schlecht ins Deutsche übersetzen, weswegen wir ihn einfach so weiter verwenden werden. Die einstmals sehr strikte Definition hat sich in der Zwischenzeit etwas aufgelockert, und lässt sich folgendermaßen zusammenfassen: eine Language Workbench ist eine Wergzeugsammlung mit der sich DSLs und die dafür zur Benutzung benötigten Werkzeuge wie z.B. Editoren oder Generatoren effizient erstellen lassen. Wichtig ist, dass die erstellten DSLs potenziell interoperabel sind, und so verschiedene Aspekte eines großen Ganzen repräsentieren können.

In den letzten Jahren sind verschiedene Umsetzungen dieses Konzeptes entstanden. Hier einige Beispiele:

- Im Rahmen des Eclipse Modeling Projektes gibt es Werkzeuge wie EMF, GMF, Xtext oder openArchitectureWare. Die Werkzeugkette unterstützt grafische und textuelle Modellierung, sowie Modelltransformationen und Codegenerierung.
- Microsoft hat vor ein paar Jahren im Rahmen der Software Factories Initiative die DSL Tools veröffentlicht, mithilfe derer sich abstrakte Syntax, grafische Notationen, und Codegeneratoren definieren lassen.
- Die finnische Firma Metacase entwickelt und vertreibt seit über 10 Jahren ein Werkzeug namens MetaEdit+, mithilfe dessen sich sehr effizient grafische DSLs und Codegeneratoren erstellen lassen.
- JetBrains, Hersteller von Eerkzeugen wie IntelliJ IDEA oder Resharper sind gerade in der Betaphase eines Werkzeuges namens MPS (Meta Programming System). Dabei handelt es sich um ein System zur Arbeit mit textuellen Sprachen (zumindest derzeit). Allerdings kommt das System ohne Parser aus, die textuelle Syntax ist lediglich eine Projektion des abstrakten Syntaxbaumes. Damit wird die Sprachmodularisierung und -erweiterung recht einfach.
- Intentional Software arbeitet seit geraumer Zeit an ihrer so genannten Domain Workbench, einem sehr anspruchsvollen Projekt, welches die vollkommene Integration verschiedener Sprachen beliebiger Syntax erwarten lässt. Auch dieses System basiert wie MPS auf projizierenden Editoren.

Microsoft Oslo lässt sich in diese Aufzählung einreihen. Oslo ist Microsofts aktuelles Bestreben, eine Language Workbench in der Microsoft-Landschaft zu etablieren.

Microsoft „Oslo“

Microsoft Oslo ist eine Language Workbench bestehend aus verschiedenen Komponenten und Sprachen: das Repository, Quadrant, und M.

M ist eine Sprachfamilie zur Definition von Datenstrukturen und Sprachen innerhalb von Oslo:

- MSchema dient zur Definition von Datenstrukturen, ganz ähnlich wie XML Schema, verwendet aber eine gefälligere Syntax. Anknüpfend an obige Erläuterungen dient MSchema also zur Definition der abstrakten Syntax von DSLs.
- MGrammar ist eine Sprache zur Grammatikdefinition. Diese dient also letztendlich zur Festlegung der konkreten Syntax für textuelle DSLs und deren Abbildung auf MSchema.
- MGraph dient zur Beschreibung der tatsächlichen Nutzdaten von Modellen, und entspricht damit mehr oder weniger XML. MGraph ist letztendlich auch die API, mit der Anwendungsentwickler auf Oslo-Daten zugreifen, beispielsweise aus einem in C# geschriebenen Interpreter.

Ein kleiner Hinweis am Rande: die Unterscheidung in diese drei Teilsprachen wird vermutlich nicht so klar bleiben wie es hier scheint. In einem Interview mit Software Engineering Radio hat Doug Purdy erläutert, dass diese drei Sprachen gerade mehr oder weniger in eine Sprache überführt werden.

Wie oben erläutert, ist natürlich auch einiges an Tooling notwendig um mit all diesen Sprachen effizient umgehen zu können. Das primäre Werkzeug zur Interaktion mit Modellen nennt sich Quadrant. Damit sollen sich in Zukunft sowohl grafische als auch textuelle Modelle anzeigen und bearbeiten lassen. Die sich im Umlauf befindliche Version arbeitet derzeit nur mit grafischen Modellen. Zur Definition und zum Testen von abstrakter Syntax und Grammatiken existiert außerdem ein Werkzeug namens intellipad. Durch eine direkte Anzeige von konkreter und abstrakter Syntax, können damit sehr einfach und interaktiv Grammatiken definiert werden. Schließlich bringt Oslo auch ein Repository mit, das potenziell große und viele Modelle in einer auf MS SQL Server basierenden Datenbank ablegt. Diverse Compiler und Kommandozeilentools runden das Paket ab.

In obiger Diskussion der Tools fällt vermutlich auf, dass nicht von Codegenerierung die Rede ist. Das liegt daran, dass Microsoft derzeit vor allem Interpreter im Hinterkopf hat. Dafür gibt es aus unserer Sicht zwei Gründe:

Zum einen hat Microsoft in der Vergangenheit gute Erfahrungen mit Plattformen gemacht, die XML Konfigurationsdateien interpretieren. Die WCF (Windows Communication Foundation) und die WF (Windows Workflow Foundation) sind Beispiele dafür. Das Problem bei all diesen Plattformen ist

vor allem die schlechte Syntax von XML, nicht der interpretative Ansatz. Microsoft will vermutlich in erster Linie Plattformen wie jene auf eine solide Basis stellen, indem eine anpassbare Syntax und ein vernünftiges Repository zur Verfügung gestellt werden.

Zum anderen existieren ja immer noch die DSL Tools. Diese sind für grafische DSLs vorgesehen, und kommen mit Werkzeugen zur Codegenerierung. Es zwingt sich der Eindruck auf, dass sich Microsoft nicht selbst Konkurrenz machen will. In oben erwähntem SE Radio Interview sagte Doug auch, dass es mittelfristig eine Integration der beiden Welten geben wird, so dass man beispielsweise für mit MSchema definierte abstrakte Syntax graphische Editoren definieren kann, oder mittels der Generatorwerkzeuge der DSL Tools aus MGraph Code generieren kann.

Unsere persönliche Interpretation dieser Aussage ist, dass die DSL Tools mittelfristig in die Oslo-Initiative aufgehen werden.

Aktueller Stand

Oslo wurde auf der Professional Developer Conference im Oktober 2008 als Community Technology Preview veröffentlicht. Es gibt kein Releasedatum, kein definiertes Produkt, und das Werkzeug ist auch noch nicht in der Betaphase. Im Laufe der nächsten Monate oder Jahre will das Oslo Team in Abständen von jeweils einigen Monaten Drops der Werkzeugkette veröffentlichen, insbesondere zu Konferenzen oder anderen relevanten Events. Die aktuelle und insgesamt zweite Veröffentlichung ist der Januar 2009 CTP.

Der Download des CTP und eine beachtliche Menge an Hand-on-labs und Videos sind im Oslo Dev Center unter http://msdn.microsoft.com/oslo verfügbar.

Oslo ist also noch nicht zur ernsthaften Verwendung geeignet (auch wenn es bereits Schulungen zu dem Thema zu buchen gibt). Auf der anderen Seite stellt Oslo aus unserer Sicht eine extrem interessante Technologie dar, bei der es sich allemal lohnt am Ball zu bleiben und erste Erfahrungen zu sammeln.

Daher werden wir im Rahmen dieser Kolumne in den folgenden Ausgaben des DotNetMagazins weitere Artikel zu Oslo veröffentlichen. Diese werden erheblich konkreter und beispielorientierter als dieser Einführungsartikel sein.

Über die Autoren

Markus Völter arbeitet als freiberuflicher Berater und Coach für die itemis AG in Stuttgart. Seine Schwerpunkte liegen dabei auf Architektur, Modellgetriebener Softwareentwicklung und domänenspezifischen Sprachen sowie Produktlinenengineering. Er hält regelmäßig Vorträge auf den entsprechenden Konferenzen und ist (Mit-) Autor verschiedener Bücher, Patterns und Artikel. Markus ist zu erreichen unter www.voelter.de sowie via voelter@acm.org

Lars Corneliussen ist Entwickler und Berater bei der itemis AG in Lünen. Nach einer Ausbildung zum informationstechnischen Assistenten spezialisierte er sich in acht Jahren intensiver Projektarbeit auf die Konzeption und Implementierung von Enterprise-Corporate-Portalen und E-Commerce-Systemen. Seit den Anfängen der Entwicklungsplattform Microsoft .NET verfolgt Lars Corneliussen die rege Bewegung der Community und bloggt über seine Herausforderungen. Seine aktuellen Steckenpferde sind Clean Code, Software-Architekturen und die modellbasierte Softwareentwicklung speziell im Bereich .NET.