Modellbildung in der Entwicklung mit Schwerpunkt Architekturen

Domain Specific Languages

Prof. Dr. Dr. h.c. Manfred Broy
Gemeinsam mit Dr. Bernhard Schätz
Fakultät für Informatik, TU München
SS 2007

Gliederung

Programmiersprachen

"Programming languages are a programmer's most basic tools " *Tony Hoare*

Klassifikations-Dimensionen:

- Paradigma (prozedural, funktional, objektorientiert,…)
- Textuell vs. Grafisch
- Imperativ vs. Deklarativ
- Zweck: Systemprogrammierung, Betriebliche Informationssysteme, …

Sprachen

Eine Sprache besteht aus:

- Syntax: Struktur der Worte einer Sprache
  - Abstrakte Syntax: Repräsentation der Wörter im Speicher
  - Konkrete Syntax: Repräsentation der Wörter für den Menschen (Mechanismen zur Manipulation)
- Semantik: Bedeutung der Worte einer Sprache
  - Operational: Ausführung auf einer Maschine
  - Translational: Übersetzung in eine andere Sprache (Mathematik/Modelle/andere Programmiersprache)
  - Axiomatisch:Beschreibung der Bedeutung durch (Umformungs-)Regeln
- Pragmatik: Methodik zur Verwendung der Sprache

→ Kein konzeptioneller Unterschied zwischen graphischen und textuellen Sprachen/Modellierungstechniken: **Programm = Modell**

Syntaxdefinition durch Grammatiken

Syntaxdefinition

- Sprachen sind eine Menge von Worten über einem Alphabet (Menge von Symbolen).
- Welche (syntaktische) Struktur ein Wort über dem Alphabet einhalten muss, um Element der Sprache zu sein, wird durch eine Grammatik beschrieben.

Verwendung

Grammatiken sind ein Werkzeug, um textuelle Sprachen zu definieren (meist in BNF)

→ dabei wird sowohl die konkrete als auch abstrakte Syntax festgelegt

Probleme von Grammatiken

Bestimmte Strukturen lassen sich nur sehr schwer durch Grammatiken ausdrücken oder nicht mehr effizient entscheiden (Chomsky-Hierarchie → Info 4).

TECHNISCHE UNIVERSITÄT MÜNCHEN

Von textuellen und graphischen Sprachen

- Eine textuelle Sprache (eine String-Sprache oder formale Sprache) über einem Alphabet A ist eine Teilmenge von A*
- Eine graphische Sprache ist eine Menge von Graphen über einer bestimmten Klasse von Graphen (knoten- und kantenmarkierte, gerichtete Graphen)

Beschreibungsmittel für graphische Sprachen:

- Meta-Modelle
- Graphgrammatiken (basierend auf Graphersetzungssystemen)

TECHNISCHE UNIVERSITÄT MÜNCHEN

Syntaxdefinition durch Metamodellierung

Syntaxdefinition

- Ein Metamodell beschreibt die Modellelemente und die Struktur, die gültige Modelle einhalten müssen.
- Ein Metamodell wird dabei meist anhand eines Datenmodells definiert.

Verwendung

Metamodelle werden meist dazu verwendet, graphische Sprachen zu spezifizieren.

→ dabei wird nur die konkrete Syntax festgelegt

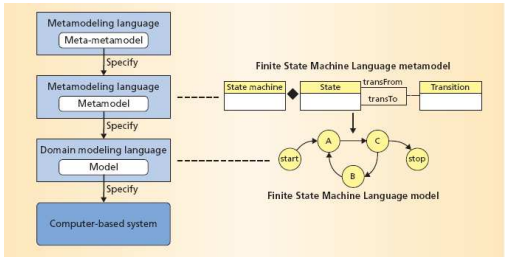
Probleme von Datenmodellen zur Syntaxbeschreibung

Nicht alle Strukturen sind über ein Datenmodell ausdrückbar, deshalb sind meist noch verschärfende logische Bedingungen (Constraints) notwendig.

TECHNISCHE UNIVERSITÄT MÜNCHEN

Beispiel: Metamodellierung

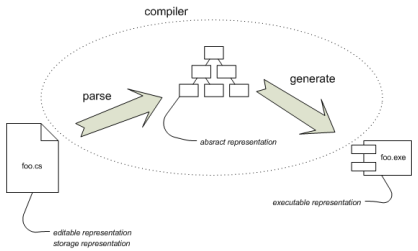
Das Metamodell eines einfachen Automaten:



TECHNISCHE UNIVERSITÄT MÜNCHEN

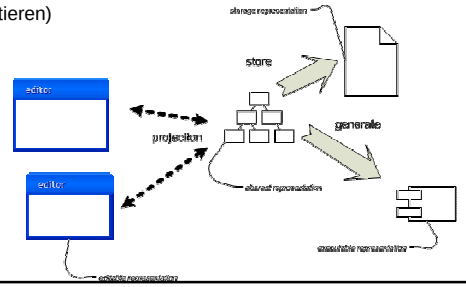
Kompilierung

- Errechnen des Abstrakten Syntax-Baums (AST) aus der textuellen Repräsentation des Programmcodes
- Erzeugung des Codes durch Traversierung des AST



Editierung graphischer Sprachen

- Editierung der Modelle anhand grafischer Repräsentation
- Der Abstrakte Syntax Graph wird bereits während des Editierens im Speicher gehalten
- Instanziierung von Modellelementen ist (meist) nur in syntaktisch korrektem Zusammenhang möglich (Syntax-getriebenes Editieren)



Interpretation vs. Compilation

- Semantik einer Sprache meist (leider) nur über den generierten Code oder einen Interpreter definiert.
- Interne und externe DSLs
- TODO <Bild: Kategorisierung anhand Backend>

Gliederung

1. Was sind eigentlich Sprachen?
   1. Definitionen
   2. Konzepte
2. Warum Domänenspezifische Sprachen?
   1. Prinzipien
   2. Aktuelle Ansätze
3. Diskussion
   1. Vor- und Nachteile
   2. Offene Fragen
   3. Literatur

Alter Hut?

DSLs sind nicht neu

- (erste sog. DSL: „Automatically Programmed Tools“ von 1959.)

Warum dann jetzt erhöhtes Interesse?

- Hoffnung, Erfolg von Allzwecksprachen zu wiederholen

  „Surely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high- level languages for programming. Most observers credit an increase in productivity with at least a factor of five.“ *
- Zunehmende Abstraktion zwingt irgendwann zu Einschränkung der Ausdrucksstärke. => DSL

*Fred Brooks, 1975/1995, „No Silver Bullet“

Definitionen

“A DSL is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.“*

Oft gehörte Metaphern:

- Allzwecksprache als Werkstatt mit vielen Allzweckwerkzeugen, z.B. Hammer, Zange, Nagel ...
- DSL als spezialisierte Fabrik, die nur Varianten eines Produkts herstellen kann, dafür aber effizienter.

*Arie van Deursen, Paul Klint, Joost Visser

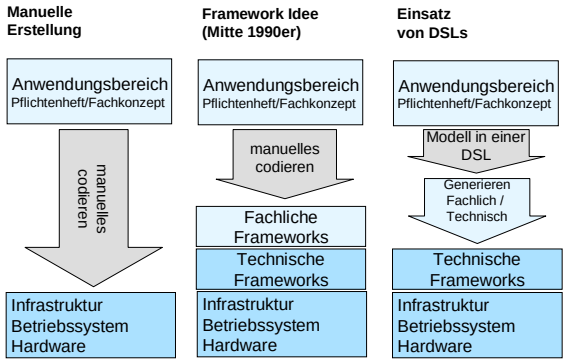
Paradigmen

- General Purpose Sprachen folgen einem bestimmten Paradigma, sind aber grundsätzlich geeignet, jedes beliebige System zu implementieren (sind Turing vollständig)

  → Aber: Auch bei Allzwecksprachen eignen sich für bestimmte Domänen unterschiedlich gut (z.B. Java und Ada für Echtzeitanwendungen)
- Domänenspezifische Sprachen sind speziell auch eine technische oder fachliche Domäne zugeschnitten.

  → Eher auf dem Abstraktionsniveau einer Bibliothek oder eines Frameworks

Abstraktionslücken und ihre Überwindung



Frameworks vs. Generierung

	Generierung	Frameworks
Wiederverwendung	Wiederverwendung von Abbildungsvorschriften zum Generierungszeitpunkt	Wiederverwendung von ausführbarem Programmcode zur Laufzeit
Parametrierung	Parametrierung erfolgt zum Generierungszeitpunkt und ist zur Laufzeit statisch	Parametrierung erfolgt dynamisch zur Laufzeit
Generizität	Abbildungsvorschrift ist generisch. Der generierte Code kann auf einen Anwendungsfall zugeschnitten sein.	Schnittstelle und Funktionalität ist generisch gehalten, um mehrere Anwendungsfälle abzudecken
Portierbarkeit	Erstellen neuer Abbildungsvorschriften; Generierung für neue Plattform	Wiederverwendung von ausführbarem Programmcode zur Laufzeit
Programmgröße	Produziert ggf. zu redundanten Programmcode	Redundanter Parametrierung; Große Bibliotheken, die u.U. nur in Teilen verwendet werden
Performance	Performance tendenziell besser generierter Code kann optimiert werden	Performance schlechter; Frameworks verlangen zusätzliche Interaktionen und besitzen dynamisches Verhalten

TECHNISCHE UNIVERSITÄT MÜNCHEN

Generische CASE-Tools

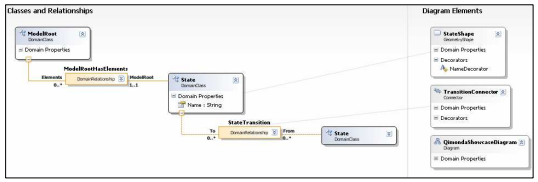
- Ein CASE-Tool bietet Funktionalitäten zur Erstellung von Modellen in einer bestimmten im Tool fest verdrahteten Modellierungstechnik in Form eines Metamodells (
  - z.B. AutoFocus, Matlab, Ascet, …
- Domänenspezifische Sprachen (meist graphisch) können mit generischen CASE-Tools implementiert werden, die es erlauben, selbst ein Metamodell zu definieren und daraus alle nötigen Editoren zur Erstellung der Modelle zu generieren.
  - Über eine Generatorsprache können schließlich textuelle Artefakte (Code, Dokumentation, …) generiert werden
  - Typische Vertreter: MetaEdit+, MS DSL-Tools, GME, …
  - Viele dieser Werkzeuge sind noch im Beta-Stadium
  - Oftmals als „Language Workbench“ bezeichnet (nach M. Fowler)

TECHNISCHE UNIVERSITÄT MÜNCHEN

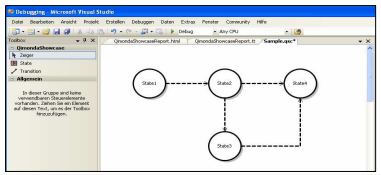
18

Beispiel: Microsoft DSL-Tools im Visual Studio

Schritt 1: Erstellung des Metamodells und eines Mappings der grafischen Repräsentation der Modellelemente in den DSL-Tools:



Schritt 2: Generierung von Editoren, welche die Erstellung von Modellen zu diesem Metamodell erlauben, als Visual Studio Plugin:



TECHNISCHE UNIVERSITÄT MÜNCHEN

19

Aktuelle Ansätze / Vertreter

- Generative Programming (Czarnecki, Eisenecker)
- Domain Specific Modeling (MetaCase)
- Model Driven Software Development (Völter, oAW)
- Model Integrated Computing (Vanderbuilt, GME)
- Language Oriented Programming (JetBrains, MPS)
- Software Factories (Microsoft, DSL Tools)

Nicht zu verwechseln mit:

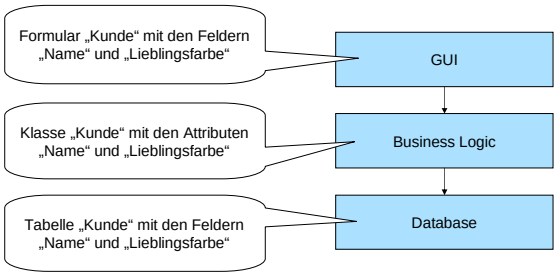
- Model Driven Architecture (OMG)
- Software Produktlinien (z.B. FODA)

TECHNISCHE UNIVERSITÄT MÜNCHEN

20

Beispiel: Redundanz in einer 3-schicht Architektur

Die Information, dass ein „Kunde“ einen „Namen“ und eine „Lieblingsfarbe“ hat, ist im System mehrfach enthalten:



Vermeidung von Redundanz

In vielen Systemen ist ein Anwendungsdomänenkonzept (z.B. Kunde) an verschiedenen Stellen der Lösungsdomäne implementiert.

⇒ Updateanomalie: Eine Änderung am Konzept „Kunde“ erzwingt mehrere Änderungen der Lösungsdomäne

⇒ DSL auf „richtigem“ Abstraktionsgrad hilft, diese Art der Redundanz zu vermeiden

Analysierbarkeit durch Einschränkung

Ansatz:

- Einschränkung der Konstrukte und Ausdrucksstärke einer Sprache, um bessere Analysierbarkeit zu erreichen.

Beispiele:

- Protokollverifikation
- Model Checking

Isolation von Variabilität

Ansatz:

- Bündelung von zusammengehöriger Information mit hoher erwarteter Änderungsfrequenz.

Beispiele:

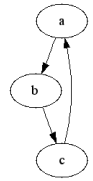
- Annahme: Business Logic ändert sich oft (z.B. wann ein Preisnachlass gewährt wird.)
  - → Rules Engine: DSL zur Beschreibung von Business Logic in Form von Regeln.
- Annahme: Layout einer Webseite wird oft geändert. Trennung der Rollen Entwickler und Designer.
  - → Cascading Style Sheets: Layout Informationen zentral an einem Ort.

Beispiele (1): „Klassische DSLs“

Textuell

- Lex (RegExps), Yacc (BNF)
- SQL
- HTML, MathML, VRML, SGML, ...
- Dot: Sprache zur Beschreibung von Graphen, automatische Berechnung eines optimalen Graphlayouts

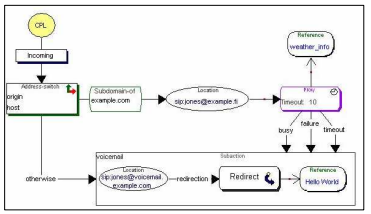
```
digraph Cycle {
  a -> b
  b -> c
  c -> a
}
```

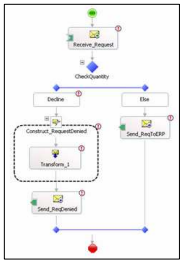


Beispiel (2): Heutige DSLs

Grafisch

- GUI Builder
- Biztalk Orchestration Designer
- CPL: Internet Telephony Services*





*Internet Engineering Task Force

Gliederung

1. Was sind eigentlich Sprachen?
   1. Definitionen
   2. Konzepte
2. Warum Domänenspezifische Sprachen?
   1. Prinzipien
   2. Aktuelle Ansätze
3. Diskussion
   1. Vor- und Nachteile
   2. Offene Fragen
   3. Literatur

Vorteile

- Hoher Abstraktionslevel
  => Knappere, kleinere „Programme“
  => Weniger Fehler (da kleinere Programme)
  => Höhere Produktivität

- Domänenspezifische Notationen
  => Selbst-dokumentierend
  => Werden von Domänenexperten verstanden

Vorteile (2)

- Eingeschränkte Ausdrucksstärke

  => Bessere Möglichkeiten für Analyse, Verifikation, Optimierung, …

- Wiederverwendung von Domänenwissen

Nachteile (1): Verwendung

- Für viele Domänen
  - Keine DSL vorhanden
  - Keine / kaum Erfahrung mit DSL
- User müssen neue Sprache lernen, aber:
  - Domänenwissen müssen sie sowieso lernen
  - Frameworks/Bibliotheken müssen auch verstanden werden
- Viele grundlegende Fragen weitgehend unverstanden
- Oftmals als weniger performant angesehen, als handgeschriebene (und –optimierte) Lösungen

Nachteile (2): Eigenbau

- Kosten für Entwurf, Implementierung und Wartung einer DSL
- Sowohl Domänenwissen, als auch Compilerbau Kenntnisse notwendig
- Schwierigkeit, den „richtigen Scope“ zu finden
- Wartung von DSL weitgehend unverstanden

Offene Fragen

- Wann lohnt sich der Einsatz von DSL, wann ist GPPL billiger / effizienter?
- Wie gehe ich bei der Entwicklung von DSLs vor?
- Was sind Qualitätseigenschaften „guter“ DSLs?
- Wie kann man DSLs effizient warten? (Gekoppelte Evolution von Spezifikation, Instanzen und Werkzeugen?)
- Welcher „Technical Space“ eignet sich in welchem Szenario? (Grammatik, Metamodell, Schema, …)

In eigener Sache

- Bei Interesse an Arbeiten in diesem Themenfeld, wenden Sie sich bitte an:

Martin Feilkas

Tel.: 089-289 17876

feilkas@in.tum.de

- Z.B. Masterarbeit: „**Erweiterung des Eclipse Modeling Framework um ein Modulkonzept zur Realisierung von Architektursichten** “

Literatur

- „Domain Specific Languages: An Annotated Bibliography“: A.van Deursen, P. Klint, J. Visser, 2000
- „When and How to Develop Domain-Specific Languages“: M. Mernik, J. Heering, A. Sloane, 2003
- Overview of Generative Software Development, C. Czarnecki, 2005
- „Language Workbenches – The Killer-App for Domain Specific Languages?“, Martin Fowler, 2005