Domain Specific Languages

Elmar Jürgens, Martin Feilkas

Perlen

6.12.2006

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, …

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."*

Metapher:

- Allzwecksprache als Werkbank mit unterschiedlichen Werkzeugen für verschiedene Aufgaben.
- DSL als Akkuschrauber: spezialisiertes Werkzeug für einen definierten Aufgabenbereich.

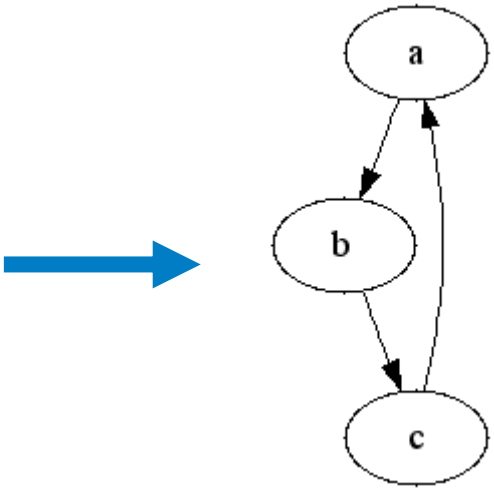
*Arie van Deursen, Paul Klint, Joost Visser

Beispiele (1): „Klassische DSLs“

Textuell

- Lex (RegExps), Yacc (BNF)
- SQL
- HTML, MathML, VRML, SGML, ...
- Teapot: Cache-Kohärenz Protokolle.
- Dot:

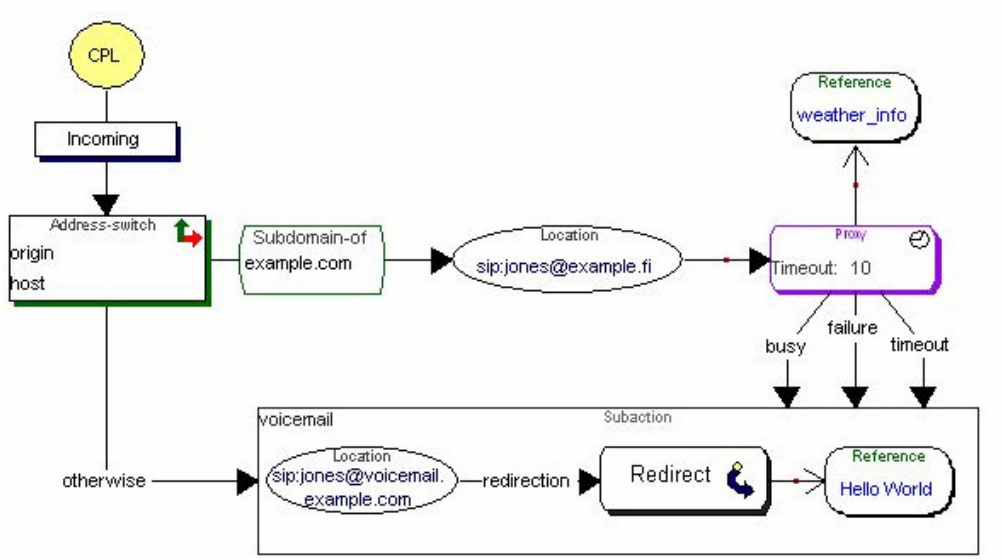
```
digraph Cycle {
  a -> b
  b -> c
  c -> a
}
```

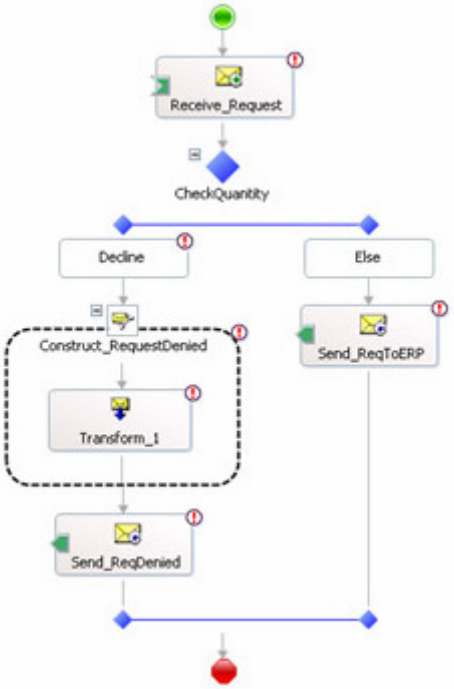


Beispiel (2): Unbekanntere Domänen

Grafisch

- GUI Builder, E\R Diagramme
- Biztalk Orchestration Designer
- CPL: Internet Telephony Services*





*Internet Engineering Task Force

TECHNISCHE UNIVERSITÄT MÜNCHEN

Alter Hut?

DSLs sind (fast) so alt wie Informatik selbst

- (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“

Vorteile

- Domänenspezifische Notationen

  => Selbst-dokumentierend

  => Werden von Domänenexperten verstanden

- Hoher Abstraktionslevel

  => Knappere, kleinere „Programme“

  => Weniger Fehler (da kleinere Programme)

  => Höhere Produktivität

Vorteile (2)

- Eingeschränkte Ausdrucksstärke

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

- Wiederverwendung von Domänenwissen

- Müssen nicht ausführbar sein

  => Man merkt Fehler nicht ;) „Berater- Bonus:“

  „Pictures, as opposed to programs, don‘t crash“*

*David Garlan

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)
- Viele grundlegende Fragen weitgehend unverstanden
- Potentiell weniger performant, als handgeschriebene (und –optimierte) Lösung

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

Reduktion von „Accidential Complexity“

Einteilung der Schwierigkeiten im Software- Entwurf:*

Essential: Datenstrukturen, Beziehungen zwischen Entitäten, Algorithmen, Funktionen

= Problem-inhärente Komplexität

Accidential: historische, künstliche, nicht problem-inhärente Komplexität.

= „Künstliche Komplexität“

Beispiel: GUI Builder, Parser Generator

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

Vermeidung von Redundanz

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

$\Rightarrow$ Updateanomalie: Eine Änderung am Konzept Kunde erzwingt mehrere Änderungen der Lösungsdomäne

$\Rightarrow$ DSL auf „richtigem“ Abstraktionsgrad hilft, diese Art der Redundanz zu vermeiden

Beispiele: CDL, „Application Generators“

Analysierbarkeit durch Einschränkung

Ansatz:

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

Beispiele:

- Protokollverifikation
- ConQAT: Load Time Type Checking
- Giotto

Isolation von Variabilität

Ansatz:

- Bündelung von zusammengehöriger Information mit hoher erwarteter Änderungsfrequenz

Beispiele:

- JBoss Rules: Business Logic in Form von Regeln. Das vermeidet zwar keine Redundanz, aber wenn sich BL ändern, hat man alles auf einem Fleck.
- CSS: Layout Informationen zentral an einem Ort.

Aktuelle Ansätze / Vertreter

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

- Model Driven Architecture
- Software Produktlinien

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, Meta-Modell, Schema, …)

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

Danke!

Fragen?

TECHNISCHE UNIVERSITÄT MÜNCHEN