Domain-Specific Languages

A Financial Engineering Case Study

Arie van Deursen

CWI, Amsterdam

February 18, 1998
Interest Rate Products

- Interest rate swap, loan, financial future, deposit, fixed rate agreement, ...

- Banks frequently introduce new products.

Consequences for bank automation:

- Financial administration management information risk analysis;

- High time-to-market for new products.

⇒ High-level product descriptions needed!
The Risla code generator

- From product description, generate:
  - data structures
  - screens
  - procedures
  - calls to legacy systems
  - ...
The Risla Language

- **Risla** – Rente Informatie Systeem *Language*.

- *Describe* interest rate product: *Generate* COBOL.

- Product characterized by *cash flows*.

- Product’s access functions: information / registration.

- Based on COBOL library accessible as *functions*.

- Time-to-market: 3 months → 3 weeks.
The Risla Project

• 1990: ORFIS / Mees & Hope: start.

• 1992 (with CWI): Language design tool prototyping.

• 1993: Risla compiler in Lex/Yacc/C. Approx. 40 products in daily use.

• 1995: (with CWI): Language extensions: Type checking, modularization, component library.

• 1996: Extension to options.


• ...: ... !!

February 18, 1998
Domain Specific Languages

Use a DSL to separate domain-specific concerns from implementation details.

Issues:

- Domain selection
- Language design considerations
- Maintainability.
- Link with object-oriented frameworks
- Methodology
Anticipated Positive Effects

- Better portable, more flexible, reliable, understandable;
- “Business rules” explicitly available, verifiable and re-usable;
- Easier to predict impact of changes;
- Lower LOC figures.
  
  (With $M = D \times ACT$ lower maintenance)

- Reduce costs:
  Private library of domain-specific programs, share costs of DSL compiler;

February 18, 1998
Domain Definition

- Domain as the real world.
  - Adopted in OO / AI community;

- Domain as a set of systems.
  - Systematic reuse community.

- Domain criteria:
  Mature, stable, and economically viable.

- Use legacy systems (specs, design, code, ...) to predict required variability.

- Scope domain by set of boundary decisions.
Design Considerations (I)

• Who in the organization will write DSL programs? Background required?

• Who will do the maintenance?

• Anticipated number of DSL programs? Average length?

• Which compile-time analyses are desirable?

• User-definable syntactic freedom?
Design Considerations (II)

How are the data types and operations implemented?

- By mapping them to a target language and associated library;
  - New operations or data types: adapt compiler.

- Within the DSL itself:
  - Language can define new types;
  - Language is Turing complete;
  + Introducing new operations or data types is easy.
Maintainability

• Maintenance of
  – DSL programs and libraries;
  – DSL compiler (new expertise?)

• (Understanding of) domain not stable.

• Language needs minimal number of users in order to survive.

• Interfacing with other programming languages, systems, DSLs, ...
## Maintainability Factors

<table>
<thead>
<tr>
<th>Maintainability Factor</th>
<th>DSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of expressing anticipated modif.</td>
<td>++</td>
</tr>
<tr>
<td>Small development costs per application</td>
<td>++</td>
</tr>
<tr>
<td>Small code size (low LOC)</td>
<td>++</td>
</tr>
<tr>
<td>Low annual change traffic (ACT)</td>
<td>0</td>
</tr>
<tr>
<td>Code readability</td>
<td>++</td>
</tr>
<tr>
<td>System modularity</td>
<td>++</td>
</tr>
<tr>
<td>Locality of changes</td>
<td>++</td>
</tr>
<tr>
<td>Testability</td>
<td>+</td>
</tr>
<tr>
<td>Code portability</td>
<td>+</td>
</tr>
<tr>
<td>Maintenance process followed</td>
<td>+</td>
</tr>
<tr>
<td>Maintainability as an objective</td>
<td>+</td>
</tr>
<tr>
<td>Quality of configuration management</td>
<td>0</td>
</tr>
<tr>
<td>Repository for modification requests</td>
<td>0</td>
</tr>
<tr>
<td>Small number of languages used</td>
<td>--</td>
</tr>
</tbody>
</table>
Language Technology

- Grammar → parser generation
- Abstract syntax tree construction
- AST traversal
- Code generation (correctness)
- Rapid prototyping
- [ Compiler construction ]
**Parse**: source text to term (AST);

**Rewrite**: apply function to source term

**Unparse**: present resulting term;

**ASF+SDF**: literate algebraic specifications.
Object-Oriented Framework

- Framework: “semi-complete” application, that can be specialized to produce custom applications.

- Set of cooperating classes.

- **White box**: adjust by *inheritance*

- **Black box**: adjust by *composition*.

- **Hot spots**: Places of *variability*.

- Don’t call; get called.
Three Examples

White Box Framework
Black Box Framework

Component Library

Hot Spots
Pluggable Objects
Fine-grained Objects

Visual Builder
DSL

Time
DSL versus OO Framework

● Developing a DSL:
  – use framework in implementation.

● Developing a framework:
  Extending with DSL helps to
  – Guide / focus framework design. Can concept be expressed naturally?
  – Encourage black-box rather than white-box frameworks
  – Abstract access to framework.

● Interface to/with legacy systems.
Domain Engineering

• Optimize software development for
  – set of multiple applications
  – in same business / problem domain

• Use legacy artifacts:
  – scope domain definitions
  – as source of domain knowledge
  – as resources for reengineering

• Result:
  – Domain-specific components
  – Domain-specific language

February 18, 1998
ODM Steps

1. Plan domain
   • Stakeholder dossier
   • Project objectives
   • Boundaries of selected domain

2. Model domain
   • Domain info from legacy artifacts
   • Commonality / variability analysis
   • Lexicon, concepts, features

3. Engineer Asset Base
   • Correlate features / customers
   • Implement asset base.

February 18, 1998
Discussion

● Other domain-specific languages?

● New application areas?

● *Embedded* domain-specific languages?

● *Visual* languages?

● DSL description as *component*?

● Anything!