Design and Implementation of Domain-specific Languages

Jaroslav Porubän
Jaroslav.Poruban@tuke.sk
JKU Linz, April 2009
1. Domain-specific languages
2. Language anatomy – concrete and abstract syntax, semantics
3. Concrete syntax - textual and graphical form
4. Embedded DSLs
5. Standalone DSLs
6. Language Environments
7. Language transformations, artifact generation
8. Case Study: DSL development
The marks for this course will be based on the project evaluation

Choose any domain (other than state machine charts)

Create a simple DSL for the selected domain

Minimal Requirements
- 1 domain class
- 1 domain relationship (reference)
- 1 artifact generator
- Simple model validation

If you have any problems with DSL project send an email

Send the DSL project to Jaroslav.Poruban@tuke.sk

Deadline: June 10, 2010
Literature


Language

Language - the words, their pronunciation, and the methods of combining them used and understood by a community

“Merriam-Webster Dictionary”

Natural Language
- human-human interaction

Computer Language
- human-computer interaction
- computer-computer interaction

Basic tool for communication
Effective communication is crucial
Computer Language

- **Artificial languages** used in computers created by humans
  - general purpose programming languages
  - domain-specific languages
  - modeling and meta-modeling languages
  - data models
  - ontologies

- **Formal Language** – suitable for machine processing
Computer Language Engineering

- **Computer language engineering** is the application of a systematic, disciplined, quantifiable approach to the development, use, and maintenance of computer languages.

- Concerned with all phases of the lifecycle design:
  - implementation
  - documentation
  - testing
  - deployment
  - evolution
  - recovery
The Role of Computer Languages

Tool, environments, platforms

Computer Languages

Paradigm
- functional
- modular
- object-oriented
- aspect-oriented
- domain-specific

Form
- textual
- graphical
- mixed

Principles
Approaches
Experience
Methods

System

Security
Usability

Functionality
Performance
Flexibility
The Effective Communication

- Human-computer interaction
  - Effective to be **comprehend**
  - Effective to be **processed**

- natural language
- GPL
- XML
- SGML
- YAML
- binary format
GPL - General-purpose Programming Language

- Useful for expressing algorithms in general
- Design without specific domain in mind
- **Programming paradigm** - fundamental style of computer programming
- General-purpose programming language are Turing-complete
- Examples:
  - Java
  - Pascal
  - Haskell
  - C#
General vs. Specific Tools

**General Purpose Tool**
- wide usage - frequently used
- takes longer to comprehend
- once learned - **do anything**

**Domain Specific Tool**
- domain specific usage
- simpler to comprehend
- higher effectiveness
A Domain-Specific Language is a language that **targets a problem domain**, which it describes and validates in terms native to the domain.

**Examples:**
- HTML - HyperText Markup Language
- CSS - Cascading Style Sheets
- SQL - Structured Query Language
- SVG - Scalable Vector Graphics
- VHDL - VHSIC hardware description language
- BNF - Backus–Naur Form
DSL - Domain Specific Language

- DSLs trade generality for expressiveness in a limited domain
- DSLs are also called
  - application-oriented
  - special purpose
  - specialized
  - task-specific
  - little languages
Bridging the Worlds

From a machine to a domain

Implementation Domain

Domain-specific languages

Problem Domain

From a domain modeling to machine execution
Advantages

- Ability to work in **terms of the problem domain**, with less scope for making the errors that come from representing it in a GPL
- Can make the models **more accessible** to people not familiar with the implementation technology (business people)
- Models expressed using DSL can be **validated** at the level of abstraction of the problem domain
- Business **knowledge is captured in a model**
- **Standardization** of domain vocabulary and notation
Advantages

- **Easier to migrate** a solution from one technology to another, or between versions of the same technology – technological independence
- Models can be used to configure an implementation consisting of multiple technologies of different types - reduce the skill and effort required to implement a solution using these technologies
- Models can also be used to generate other artifacts - configure other systems, networks, and products
Disadvantages

- The necessity to build DSL tools
  - parsers/compilers
  - model validators
  - editors
  - code completion support tools
  - visualizers
  - document support tools
The Importance of Identifiers in GPL

```java
public void dit(BObject b1, BObject b2, double k) {
    if (k > 0 && b1.esm() > k) {
        b1.hbe(k);
        b2.ppo(k);
    } else {
        //...
    }
}

public void transfer(Account from, Account to, double ammount) {
    if (ammount > 0 && from.getDisposableBalance() > ammount) {
        from.withdraw(ammount);
        to.deposit(ammount);
    } else {
        //...
    }
}
```
Language Anatomy

Concrete syntax

Abstract syntax

Semantics
Informal definition of the SML abstract syntax (domain concepts)

- **State machine** has at least one state and at least one transition
- **State machine** has exactly one start state
- Every state has its unique name
- **Transition** is connection between two states
Abstract Syntax

- Characterizes in an abstract form
  - kinds of elements that make up the language
  - rules for how those elements may be combined
- There are
  - atomic elements
  - composite elements
- Useful for the definition of semantics
- Formally defined by
  - BNF
  - Metamodels
Abstract Syntax - SML

- **BNF**
  
  StateMachine ::= StartState State+ Transition+
  State ::= Identifier
  StartState ::= Identifier
  Transition ::= Identifier Identifier

- **Metamodel**

![Diagram showing the metamodel of a StateMachine, States, and Transitions]
Concrete Syntax

- Defines how the language elements appear in a concrete, human-usable form (computer-human interaction)
  - textual
  - graphical
- Another term for concrete syntax is notation
- A serialization syntax is used to persist and interchange language expressions in serialized form (computer-computer interaction)
Concrete Syntax - SML

- BNF
  
  StateMachine ::= StartState State+ Transition+
  StartState ::= 'start' Identifier
  State ::= 'state' Identifier
  Transition ::= 'transition' 'from' Identifier 'to' Identifier
  Identifier ::= ['a'-'z''A'-'Z']+

- Textual form – sentence

  start  Ready
  state  Ready
  state  Running
  state  Unsafe
  transition from  Ready to  Running
  transition from  Running to  Ready
  transition from  Running to  Unsafe
  transition from  Unsafe to  Running
Concrete Syntax - SML

- Graphical form - sentence

Diagram:

- Ready
- Running
- Unsafe
Semantics

- The **semantics** of a language define its meaning
- Translation semantics
  - translate expression of a language under study into another language that already has semantics
- Trace-Based Semantics
  - defines the meaning of a language in terms of execution traces
DSL Usage Scenario

- **Model (DSL concrete syntax)**
- **Metamodel**
- **Parser**
- **Generator**
- **Template**

The diagram outlines the process of a DSL usage scenario, starting with the model (DSL concrete syntax), passing through the parser, metamodel, and generator, leading to the generated artifacts (Language).
Parsing DSL

Jaroslav Porubän
Jaroslav.Poruban@tuke.sk
JKU Linz, April 2009
Approaches to DSL Implementation - Parsing

- Approaches (are explained on examples)
  - Standalone DSL (External DSL)
    - hand-crafted parser
    - parser generator
  - Embedded DSL (Internal DSL)
  - XML
  - YAML
  - Language Environments (aka Workbenches)
Standalone DSL

- Standalone textual DSL requires specific language processor – parser
- Operates on pure text input
- Hand-crafted Parser
  - Line (delimiter) parser
  - Recursive Descent Parser
- Generated Parser
  - Parser Generators
Parser Generator

- Parser generator generates parser from a language definition
  - YACC, Bison, JavaCC, ANTLR, Coco/R
- Parser is produced usually as source code in a GPL

Diagram:

- Language Definition
  - Grammar
  - Attribute Grammar
  - Reads
  - Parser Generator
  - Generates
  - Parser Source Code
  - Sentence
    - 1 + (2 - 7)
Standalone DSL – pros and cons

+ Concrete DSL syntax is fully under the control of language designer

- Parser generators are complex tools with their own DSLs

“Implementing a textual DSL by implementing its grammar can be a difficult and error-prone task, requiring significant expertise in language design and the use of a parser-generator” [1]

“Implementing a parser-generator is definitely an expert task, because a grammar might be ambiguous or inconsistent, or might require a long look-ahead to decide what to do.” [1]

“DSL development is hard, requiring both domain knowledge and language development expertise. Few people have both.” [6]
Embedded DSL

- Implemented in a **host language** – usually GPL (e.g. Java, C#, Ruby)
- It is not required to create language parser
- DSL source code is developed using a standard programming environment (and also with support from the environment)
- Embedded DSL design patterns are explained in [3], [6]
public abstract class StateMachineBuilder {
    public StateMachine create() { define(); }

    protected abstract void define();

    protected void start(String name) { ... }

    protected void state(String name) { ... }

    protected void transition(String fromName, String toName) { ... }
}

public class TestMachine extends StateMachineBuilder {
    protected void define() {
        start("Ready");

        state("Ready");
        state("Running");
        state("Unsafe");

        transition("Ready", "Running");
        transition("Running", "Ready");
        transition("Running", "Unsafe");
        transition("Unsafe", "Running");
    }
}
Embedded DSL – pros and cons

- It is not required to write a DSL parser
- Standard programming environment can help with DSL source code creation
- Concrete syntax is influenced by host language syntax
- Standard programming environment is not enough restrictive when DSL source code is created
XML - Extensible Markup Language

- XML is a technology concerned with the description and structuring of data
- XML document is platform (machine) independent form of storing data
- XML is standard for creating languages that meet the XML criteria

Example

```xml
<person>
  <name nickname="007">
    <first>James</first>
    <last>Bond</last>
  </name>
</person>
```
<machine start="Ready">
    <state name="Ready" />
    <state name="Running" />
    <state name="Unsafe" />

    <transition from="Ready" to="Running" />
    <transition from="Running" to="Ready" />
    <transition from="Running" to="Unsafe" />
    <transition from="Unsafe" to="Running" />
</machine>
XML – pros and cons

- Good tool support
- Industrial XML language processor
- Syntax defined through standard formats (DTD, XSD, Relax NG)
- Too verbose for the human readers/writers
- DSL’s syntax must follow XML spec

```
1 + 2 * 7
```

```
<add>
  <number>1</number>
  <mul>
    <number>2</number>
    <number>7</number>
  </mul>
</add>
```
YAML - YAML Ain't a Markup Language

- YAML is a human-readable and human-writable data serialization
- Notation
  - list
  - hash
- Style
  - indented
  - inline
- Example
  - {name: John Black, age: 29}
  - name: Joan Red
    - age: 25

- JSON (JavaScript Object Notation) is subset of YAML
{ 
    start: Ready
    states: [Ready, Running, Unsafe]
    transitions: 
        - {from=Ready, to=Running}
        - {from=Running, to=Ready}
        - {from=Running, to=Unsafe}
        - {from=Unsafe, to=Running}
}
YAML – pros and cons

- + Industrial YAML processor
- + Human friendlier than XML
- - Lack of schema formats
- - DSL’s syntax must follow YAML spec
Language Environments

- Tools that support creating DSLs in the style of modern IDEs
- Generates language tools from the language definition (DSL)
  - code completion
  - graphical editors
  - refactoring tools
  - other support tools
Microsoft Visual Studio DSL Tools

Classes and Relationships

StateMachine
- DomainClass
  - Domain Properties

StateMachineHasStates
- Elements
  - DomainRelationship 0..*
- StateMachine 1..1

State
- DomainClass
  - Domain Properties

Transition
- Targets
  - DomainRelationship 0..*
- Sources
  - DomainRelationship 0..*

IssueState
- DomainClass
  - Domain Properties
  - Name : String

StartState
- DomainClass
  - Domain Properties
Microsoft Visual Studio DSL Tools
Language Environment – pros and cons

- + Simplifies the development of DSL
- + Rapid language development with support tools
- - Only starts to widespread, LE are in early stages
Generating Artifacts

Jaroslav Porubän
Jaroslav.Poruban@tuke.sk
JKU Linz, April 2009
Transforming Artifacts

- DSL₁ Artifacts
- DSL₂ Artifacts
- DSL₃ Artifacts
- GPL Artifacts

**Vertical Transformation**

**Horizontal Transformation**
Approaches to Artifacts Generation

- Approaches (are explained on examples)
  - Transformed Generation
  - Templated Generation
  - XSLT Transformation
Conclusions

- DSLs vs. GPLs
- Programming vs. Modeling
- Concrete vs. Abstract Language Syntax
- Standalone vs. Embedded DSLs vs. Language Environments
- Transformed Generation vs. Templated Generation