DSL Tools

By Massimiliano Menarini

CSE294 – 2/28/2011
Domain Specific Languages are “small” languages with a limited scope

- Goal: elicit code from domain experts
- Different way to implement them
  - Internal vs. External Languages
  - Textual vs. Graphical Languages
- To be successfully used we need multiple tools
  - IDEs, Transformation Tools, Verification, Debugging
Outline

- Papers that inspired this presentation
- Overview of exiting tools
  - Case study the Eclipse Modeling Project
- Language-Oriented Programming
- Textual vs. Graphical DSL
- Internal vs. External DSL
- Language Workbenches
- Case study: Eclipse Modeling Tools
- Outlook
Developing DSLs
- Editors: textual or graphical editor
- Verification: consistency, debug

Leveraging DSLs
- Transformation: Compilers, Interpreters, Translators, refactoring tools

Learning
- Simple vs. complex concrete syntax
- Limited scope of the language, few primitives
Eclipse Modeling Project

- It is a good example of language workbench
- Suite of tools integrated in eclipse IDE
- Support both textual languages and graphical languages
- Multiple tools for transformation
- Different languages implemented with this tools (for example UML)
Language-Oriented Programming

- Key idea is to have multiple languages to program different aspects of a software system

- Benefits
  - Domain experts can contribute directly to the program
  - Optimizations can be applied to all programs by the compiler
  - Separation between application knowledge modeled in DSL by domain experts and programming knowledge embedded in DSL tools

- Drawbacks
  - Developing a DSL language requires both domain and programming knowledge
  - Good tools are difficult to develop
  - To avoid language cacophony problem each language must be very simple and close to what domain experts are familiar with

Adapted from M. Fowler [4]
Requirements for LOP

* Full IDE available for the DSL
  * Support for workgroups
  * Support for integration of multiple DSLs
* DSL must be simple (not a full GPL)
  * Must be able to leverage the abstractions in DSL
  * Concrete syntax dictated by domain experts
* Tools must simplify creation of professional tools
  * Tools dependability
  * High productivity
We have successful examples of both textual and graphical DSL.

**Textual**
- Backus–Naur Form for context free grammars
- LaTeX Document Description Language
- Mule ESB XML configurations for routing
- Apache Camel embedded DSL for routing

**Graphical**
- UML Activity Diagrams
- Petri nets
- Graphical UI Development tools
<choice>
  <when expression="payload=='foo'" evaluator="groovy">
    <append-string-transformer message="Hello foo" />
  </when>
  <when expression="payload=='bar'" evaluator="groovy">
    <append-string-transformer message="Hello bar" />
  </when>
  <otherwise>
    <append-string-transformer message="Hello ?" />
  </otherwise>
</choice>

From: http://www.mulesoft.org/documentation/display/MULE3USER/Routing+Message+Processors
RouteBuilder builder = new RouteBuilder()
{
    public void configure()
    {
        errorHandler(deadLetterChannel("mock:error"));
        from("seda:a")
            .choice()
                .when(header("foo").isEqualTo("bar"))
                    .to("seda:b")
                .when(header("foo").isEqualTo("cheese"))
                    .to("seda:c")
                .otherwise()
                    .to("seda:d");
    }
};

From: http://camel.apache.org/message-router.html
* <camelContext errorHandlerRef="errorHandler"
  xmlns="http://camel.apache.org/schema/spring">
  *  <route>
  *   <from uri="seda:a"/>
  *   <choice>
  *    <when>
  *     <xpath>$foo = 'bar'</xpath>
  *     <to uri="seda:b"/>
  *    </when>
  *  </choice>
  * </route>
  *
  ...

*  </camelContext>

From: http://camel.apache.org/message-router.html
Tradeoffs

Textual

* Easier tool implementation
* Can use any text editor
* Can be easily embedded in other language
* Can be less understandable for domain experts

Graphical

* Often more intuitive for domain experts
* Can be difficult to develop tools with the appropriate qualities
* Can be embedded in some graphical language (UML profiles for example)
Internal vs. External DSL

- **Internal DSL**
  - Implemented inside a general purpose host language
  - Characteristics vary depending on the feature of the base language: Macros, Monoids...
  - Example of internal friendly languages is LISP
  - Sometimes can be implemented as libraries

- **External DSL**
  - Developed from the ground up – give maximum syntax freedom
  - Need good language development support to be usable
  - External languages give potentially better results but are expensive to develop
Tradeoffs

External DSL

* Close to notation used by domain experts
* Possible domain specific analysis
* Possible good error reporting
* Harder to develop
* Easier to incur in incorrect designs
* Difficult to implement language extensions

Internal DSL

* Modest development effort
* Often more powerful language
* Host language infrastructure reuse
* Syntax is not optimal
* Overloading existing operators can be confusing
* Bad error reporting
* Hard to implement domain specific optimizations

Adapted from Mernik et al. [3]
Tools that support creation of external DSLs
- Provide ways of specifying the concrete and abstract syntax
- Provide support for generation of editors and other tools
- Provide frameworks for developing translators or interpreters

Language Workbenches switch the focus from a single concrete syntax to a single abstract syntax
- Support integration of multiple concrete syntaxes
- Difference between Abstract syntax serialization and concrete syntaxes used by programmers

Adapted from M. Fowler [4]
Language Workbenches

From: http://martinfowler.com/articles/languageWorkbench.html
Example: Eclipse Modeling Tools

- Various tools and technologies integrated with the Eclipse IDE
  - Professional IDE can be easily reused for your DSLs
Abstract Syntax

* Eclipse Modeling Framework (EMF)
  * Representation of abstract syntax using metamodel
    * Metamodel technology similar to UML MOF
    * Support serialization to XML
  * Graphical tools to model EMF
  * Graphical editor for editing models based on EMF metamodel integrated in eclipse
    * It as a generic editor for the storage representation of languages based on this framework
Concrete Syntax

* Tools to support both textual and graphical concrete syntaxes
* Graphical Modeling Project (GMP)
  * GMF Tooling and GMF Runtime support
* Textual Modeling Framework (TMF)
  * Xtext – support IDE generation with syntax highlighting, code completion, etc ...
Support multiple transformation technologies
  * Model to Model Transformation (M2M)
  * Atlas Transformation Language (ATL)
  * Model to Text Transformation (M2T)
  * Java Emitter Templates (JET)
  * Xpand
Best solution for language oriented programming is external languages

Tools to support creating and manipulating external languages have progressed greatly

Currently easy to create a textual language with great IDE support for editing and manipulating

Also graphical languages are supported and they have good tools for editing and transforming

My evaluation:

The current state of the art is mature enough for DSL to become a major player in software development
Questions?

Thank you!