DSL vs. Library API Shootout

Rich Unger
Salesforce.com

Jaroslav Tulach
Oracle
Agenda

- What do we mean by DSL?
- What do we mean by library?
- When is it good to use a DSL?
- When is it a bad idea?
- Evolution
- Versioning
- Tooling
- Q/A
What is a DSL?

A *programming* language or *specification* language dedicated to

- a particular *problem domain*,
- a particular *problem representation technique*, and/or
- a particular *solution technique*.

---[wikipedia](http://en.wikipedia.org/wiki/Domain_Specific_Language)
DSL Classification

- Processing style
  - Own parser (External)
  - XML based
  - Embedded in other programming language (Internal)
- Computational power
  - Declarative programming
  - Turing complete
- Not quite here: Tooling
  - Need to extend IDEs to support the DSL
  - Tooling standardized for all IDEs
DSL Examples

- LOGO (or Karel)
- SQL
- ZIL (Zork Implementation Language)
- Postscript
- TeX
- CSS
- BNF Grammars (YACC, Antlr, etc)
- Apex
- XML variants (Ant, VoiceXML, XSLT, Docbook, SVG)
- Embedded/Internal (in Haskell, Scala, Java6)
It's Okay to Use XML?

- Quick to develop
- Free lexing
- Lots of existing libraries to manipulate it
- Standard syntax for AST representation
- Poor performance
- Completely unreadable to humans

```
<one-of>
  <item>Michael</item>
  <item>Yuriko</item>
  <item>Mary</item>
  <item>Duke</item>
  <item>
    <ruleref uri="#otherNames"/>
  </item>
</one-of>

<one-of>
  <item weight="10">small</item>
  <item weight="2">medium</item>
  <item>large</item>
</one-of>
```
Libraries and Embedded DSLs

- Lexing automated
- Free interpretation
- Targeting wide audience of developers
- Bound to syntax of the language
  - Not a real problem for functional languages
  - People like Java
- Creates de-facto new language
  - Reading on paper?

```
expr ::= expr + term | term
term ::= term * factor | factor
factor ::= '(' expr ')' | digit+
digit ::= '0' | '1' | ... | '9'
```

```scala
object arithmeticParser extends StdTokenParsers {
  type Tokens = StdLexical ; val lexical = new StdLexical
  lexical.delimiters ++= List("(" , ")", "+", "*")

  lazy val expr =  term*("+" ^^^ {(x: int, y: int) => x + y} )
  lazy val term = factor*("*" ^^^ {(x: int, y: int) => x * y} )
  lazy val factor: Parser[int] = "(" ~> expr ~< "")" | numericLit ^ ^ (.toInt)
}
```
When is it good to use a DSL?
1. Targeting Domain Experts, Not Java Experts

- ZIL: lets novel authors program whole games
- TeX: used in academia across many disciplines

Would a Java API for outputting typography even make sense?
- Excel formulas: non-programmers do amazing things with excel

This is a sliding scale...
2. The Domain Lends Itself to an Idiom Expressed in a Simple Syntax

TO REDSQUARE
; draw the outline
REPEAT 4 [FORWARD 100 RIGHT 90]

; move into the square
PENUP
RIGHT 45
FORWARD 4

; fill the square with red
SETFLOODCOLOR 4
FILL

; move back
BACK 4
LEFT 45
PENDOWN
END

http://et.bgcbellevue.org/logo/
Example: Apex Triggers

Actual Trigger

trigger CashOnlyPlease on Account (before insert, before update) {
    for (Account a : Trigger.new) {
        if (a.name == 'Deadbeat Inc.')
            a.credit_terms = 'COD';
    }
}

Proposed Java Library Syntax

@DbTrigger("BEFORE_INSERT, BEFORE_UPDATE")
public class CashOnlyPlease implements Trigger<Account> {
    public void execute(List<Account> triggerOld, List<Account> triggerNew) {
        for (Account a : triggerNew) {
            if ("Deadbeat Inc.".equals(a.getName()))
                a.setCreditTerms("COD");
        }
    }
}
Example: Apex Triggers

Actual Trigger

trigger CashOnlyPlease on Account (before insert, before update) {
    for (Account a : Trigger.new) {
        if (a.name == 'Deadbeat Inc.')
            a.credit_terms = 'COD';
    }
}

Proposed Java Library Syntax

@DbTrigger({Before.INSERT, Before.UPDATE})
public static void execute(TriggerContext<Account> ctx) {
    for (Account a : ctx.getNew()) {
        if ("Deadbeat Inc.".equals(a.getName()))
            a.setCreditTerms("COD");
    }
}
3. You Can Eliminate Boilerplate or Do Validation Based on Domain Assumptions

*If it doesn't make sense for the domain, it shouldn't compile.*

Static type checking for domain objects

```java
Account[] accs =
[SELECT firstname, lastname FROM Contact] // compile error
```

Bring in a set of assumptions from the domain

```java
public class Foo with sharing { ... }
```
Example: Apex SOAP Endpoints

**Apex Syntax**

```apex
webservice String getSomething(integer someParam) { ... }
```

**Proposed Java Syntax**

```java
@webservice public String getSomething(integer someParam) { ... }
```

But the *intent* of `webservice` is to define a scope (the web). You wouldn't say:

```java
@public private String ...
```
Why not use DSL?

- Industry is conservative
- Developers love Java
- Libraries naturally extend the language
- Good library increases adoption
- People don't know better options than DSL
  - Annotation Processors
  - Natural like syntax
- Can Java be DSL meta language?
Demo
"Java on Rails"
Compile time live access to Data Base
Evolution

Requirements change over time. How do you evolve a DSL to **sunset** old features and **compatibly introduce** new ones?
Goodbye @Deprecated

New versions can completely change syntax/semantics
- Complete Control over Parser
- Allows you to keep the **intent** clear in the syntax

Mechanism: *versioning*
Version is in the file itself

```xml
<vxml version="2.0">
```

Can use transforms, intermediate representations, or just multiple parsers
Example: Apex

- Classes/Triggers stored in the DB
  - Column for version
  - User editable
- One parser internally
  - Checks version when behavior differs

<table>
<thead>
<tr>
<th>Action</th>
<th>Name</th>
<th>Namespace Prefix</th>
<th>Api Version</th>
<th>Valid</th>
<th>Status</th>
<th>Size Without Comments</th>
<th>Last Modified By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit</td>
<td>anExt</td>
<td>rungerdev</td>
<td>20.0</td>
<td>✓</td>
<td>Active</td>
<td>481</td>
<td>Test User, 7/15/2010 3:43 PM</td>
</tr>
<tr>
<td>Edit</td>
<td>DescTest</td>
<td>rungerdev</td>
<td>19.0</td>
<td>✓</td>
<td>Active</td>
<td>595</td>
<td>Test User, 3/18/2010 6:50 PM</td>
</tr>
<tr>
<td>Edit</td>
<td>FooBatch</td>
<td>rungerdev</td>
<td>19.0</td>
<td>✓</td>
<td>Active</td>
<td>418</td>
<td>Test User, 2/10/2010 11:25 AM</td>
</tr>
<tr>
<td>Edit</td>
<td>Ret</td>
<td>rungerdev</td>
<td>20.0</td>
<td>✓</td>
<td>Active</td>
<td>172</td>
<td>Test User, 5/14/2010 4:16 PM</td>
</tr>
<tr>
<td>Edit</td>
<td>startHereController</td>
<td>rungerdev</td>
<td>14.0</td>
<td>✓</td>
<td>Active</td>
<td>3,223</td>
<td>Test User, 1/23/2010 12:42 PM</td>
</tr>
<tr>
<td>Edit</td>
<td>TestClass</td>
<td>rungerdev</td>
<td>19.0</td>
<td>✓</td>
<td>Active</td>
<td>116</td>
<td>Test User, 1/23/2010 12:48 PM</td>
</tr>
<tr>
<td>Edit</td>
<td>TestController</td>
<td>rungerdev</td>
<td>19.0</td>
<td>✓</td>
<td>Active</td>
<td>194</td>
<td>Test User, 2/23/2010 9:41 AM</td>
</tr>
<tr>
<td>Edit</td>
<td>XMLDom</td>
<td>rungerdev</td>
<td>14.0</td>
<td>✓</td>
<td>Active</td>
<td>6,713</td>
<td>Test User, 1/23/2010 12:42 PM</td>
</tr>
</tbody>
</table>
Floating point literals:

- In version 16.0, this literal is a double
  
  12.4

- If you change the class to 17.0, it's a BigDecimal. To get a double, you need
  
  12.4d
Example: Apex

Implementation:

Object value;
if (currentVersion > 16.0)
    value = new BigDecimal(floatingPointToken);
else
    value = Double.valueOf(floatingPointToken);
Evolution of Libraries

Requirements change over time. How do you evolve a library to **sunset** old features and **compatibly introduce** new ones?
Versioning of Libraries

- Library identification
  - code name
  - version
- Dependencies on other libraries
  - no classpath
  - specify code name and version
- Runtime Container
  - NetBeans, OSGi
- Backward Compatibility Rules
  - Bytecode is a "DSL" for compatibility
Deprecations in Libraries

- `@Deprecated`
- `@Transformation`
  - http://lang.dev.java.net
  - Support in all good IDEs
- `@PatchByteCode`
  - non public for compilation
  - public for execution
- Moving to separate library
  - dependency transformations
Versioning of Annotation Processors

- Compile time
- Complete control on generated code
- Annotations support default values
- Adding new annotations

```
// version 1.0
@ActionRegistration
class MyAction {
}
```

```
// version 1.1
@ActionRegistration(asynchronous=true)
class MyAction {
}
```

```
// alternative 1.1
@ActionRegistration
@ActionAsynchronous
class MyAction {
}
```
Tooling for Free

- IDEs support Java
  - code completion
  - javadoc
  - navigation
  - overrides, usages, refactoring
- Good IDEs support Java6 - e.g. annotation processors
  - to generate classes
  - to provide code completion
  - compilers yield errors
  - no changes to build process (javac is enough)
- Write once, edit, compile, publish anywhere!
Most IDEs provide easy tooling for creating support in that IDE for DSLs

Language Workbenches (JetBrains MPS)

Simple syntax? Restricted domain? Perhaps you don't need an IDE!
References

apidesign.org

antlr.org

developer.force.com