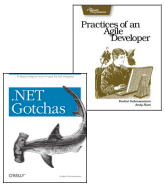
Domain Specific Languages

spkr.name = 'Venkat Subramaniam'
spkr.company = 'Agile Developer, Inc.'
spkr.credentials = %w{Programmer Trainer Author}
spkr.blog = 'agiledeveloper.com/blog'
spkr.email = 'venkats@agiledeveloper.com'

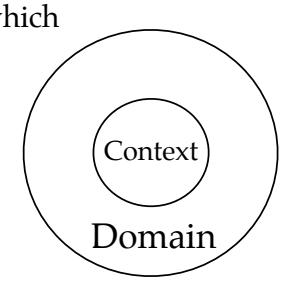


Abstract

- Domain Specific Languages or DSLs are languages that target a specific kind of problem or domain. We've had various degree of success with DSLs, over the past several years, in narrow areas. However, DSLs are not widely used in general purpose application partly because the popular widely used languages today do not make it easy.
- In this presentation we will introduce DSLs, delve into their benefits. We will look at a number of examples as to how we can utilize them for common application tasks. We will take a look at what language developments, tools, and techniques are in the horizon that could bring this to common place.

Domain and Context

- What's Domain?
  - It is an area of sphere of knowledge, influence, or activity
- What's Context?
  - A logical framework within which we evolve models
- What's Bounded Context?
  - Defines a logical boundary of relevance within the context



Domain Specific Language

- What's DSL?
  - A language targeted at a particular type of problem
  - Syntax focuses on intended domain/problem
  - Unlike a general purpose language, you can't use it for all kinds of stuff (or don't want to)
  - Limited in scope and capability
  - Hence, they're small and simple

External vs Internal

- External DSL
  - You define a new language
  - You then parse the commands to take actions
  - Can be lot of fun ;)
  - One of my first jobs was to maintain lex and Yacc...
- Internal DSL
  - Constructs of a language are construed to form a DSL
  - You don't need to write an separate parser
  - Commands tactfully map over to constructs like methods, etc.
  - Also known an Embedded DSL

External or Internal

- It depends on your application
- Certain languages are better suited for internal DSLs
  - Smalltalk, Lisp, Ruby, Groovy, ...
- Other languages are better suited for external DSL and offer good parsing libraries or capabilities
  - C++, Java, C#, ...

Characteristics of a DSL

- What are some of the qualities of a DSL?
  - Context Driven
    - Extremely context sensitive
  - Fluent Interfaces
    - Provides for a humane, easy to understand interface

Context Driven

- DSL is extremely context sensitive or context driven
- A lot of things are knows from the context
- Without context, we may be confused or misled
- With context, there is less clutter, terse, but readable

Context Driven ...

- I heard my friend Neal Ford yell out

  Venti Latte with 2 Extra Shots
- What's he talking about?
- He's using the Starbucks DSL
- Nowhere did he mention the word coffee, but he sure got a good one at a high price ;)
- That's context driven

Context Driven ...

- I heard my friend Scott Davis mutter

  place a $ in a GString
- What's he talking about?
- You don't want to know
- He sure was talking about embedding variables into the Groovy's mutable string class
- That's also context driven :)

```
today = new Date()
str = "Today is ${today}"
println str
println str.class.superclass
```

```
Today is Wed Sep 05 04:35:28 MDT 2007
class groovy.lang.GString
```

Fluent Interface

- I was going to call this hygienic interface
- I like the name Eric Evans and Martin Fowler promote–Fluent Interface/Humane Interface
- It's designed to be readable and to flow naturally
- Not easy to design–you put in more effort to create it, so it is easier for users to use

A Little Compromise

- Command Query Separation was promoted by Bertrand Meyer
  - Command methods—mutators or modifiers—affect object state and don't return anything
  - Query methods—don't affect object state and return useful information
- You compromise a little to be fluent (kind of like words bent to rhyme in a poem)
- Command methods return self or a promoter for a good flow

Fluent Loops

```
// Java Sytle
for(int i = 0; i < 10; i++)
{
  System.out.println(i);
}

// Groovy Styles
for(i in 0..9)
{
  println i
}

0.upto(9) { println it }

0.step(10, 2) { println it }
```


Fluent Interface in EasyMock

```
expect(alarm.raise()).andReturn(true);
expect(alarm.raise()).andThrow(new InvalidStateException());
```

Fluent Interface in JSONObject Library

```
JSONObject json = new JSONObject()
  .accumulate("key1", "value1")
  .accumulate("key2", "value2")
  .accumulate("key3", "value3");
```


Fluency in Rails/Active Records

```
class State < ActiveRecord::Base
  validates_uniqueness_of :code
  validates_numericality_of :year_into_statehood
  validates_inclusion_of :year_into_statehood, :in => 1776..1959

  validates_presence_of :name
  validates_presence_of :code
  validates_presence_of :year_into_statehood
end
```

Fluency in Grails/GORM

```
class State
{
    String twoLetterCode

    static constraints = {
          twoLetterCode(size: 2..2)
        }
}
```


Let's Order Pizza

- Voice on line (VOL): Thanks for calling Joe's
- You: Ya, A Large Thin Crust
- VOL: Toppings?
- You: Olives, Onions, Bell Pepper
- VOL: Address
- You: 101 Main St.,...
- VOL: Card?
- You: Visa 1234-1234-1234-1234
- VOL: 20 minutes–Thank you

Java Pizza Order

```
PizzaShop joes_pizza = new PizzaShop();
joes_pizza.setSize(Size.LARGE);
joes_pizza.setCrust(Crust.THIN);
joes_pizza.setTopping("Olives");
joes_pizza.setTopping("Onions");
joes_pizza.setTopping("Bell Pepper");
joes_pizza.setAddress("101 Main St., ...");
int time = joes_pizza.setCard(CardType.VISA, "1234-1234-1234-1234");
System.out.printf("Pizza will arrive in %d minutes\n", time);
```

- Not fluent
- Very noisy—how many times do you have to say joes_pizza?
- Can we make it fluent?

with

- Some languages have a *with* keyword
- within the scope, methods are invoked on an implicit object
- Here is an example from JavaScript

```
// JavaScript accessing Java API in Java 6

lst = new java.util.ArrayList();

with(lst)
{
  add(1);
  add(2);
  println(lst.size());
}
```

with in Groovy?

- Groovy has no *with* keyword
- It has identity (Thanks! to Guillaume LaForge for this pointer ☺)

```
lst = []

lst.identity {
  add(1)
  add(2)
  println size
}
```


Let's use JS to Order Pizza

```
importPackage(com.agiledeveloper);

joes_pizza = new PizzaShop();

with(joes_pizza)
{
  setSize(Size.LARGE)
  setCrust(Crust.THIN)
  setTopping('Olives')
  setTopping('Onions')
  setTopping('Bell Pepper')
  setAddress('101 Main St.,...')
  time = setCard(CardType.VISA, "1234-1234-1234-1234")
  println("Pizza will arrive in " + time + " minutes")
}
```

How about Fluent Java?

```
int time = new PizzaShop()
          .setSize(Size.LARGE)
          .setCrust(Crust.THIN)
          .setTopping("Olives")
          .setTopping("Onions")
          .setTopping("Bell Pepper")
          .setAddress("101 Main St., ...")
          .setCard(CardType.VISA, "1234-1234-1234-1234");
System.out.printf("Pizza will arrive in %d minutes\n", time);
```

- Less clutter
- There is a context in each call
- Each set method (except setCard()) returns *this* (PizzaShop)
- Can we make this a bit more readable?

A bit more Fluent...

```
int time = PizzaShop.orderPizza()
          .ofSize(Size.LARGE)
          .withCrust(Crust.THIN)
          .withTopping("Olives")
          .withTopping("Onions")
          .withTopping("Bell Pepper")
          .deliverToAddress("101 Main St., ...")
          .chargingCard(CardType.VISA, "1234-1234-1234-1234");
System.out.printf("Pizza will arrive in %d minutes\n", time);
```

- Method names make sense in the flow, but may not make sense alone
- What's a catch?

Ordered vs. Unordered

- Some DSLs may require a certain order in the flow
  - You can force ordering by tactfully defining return types that will allow method chaining in only certain order
    - ~Not easy
- Others may not require ordering
- It depends on the particular problem
- Ideally, not requiring an order is great, but may not always be possible

How to Enforce Sequence?

```
int time = PizzaShop.orderPizza(
          new PizzaShop()
          .ofSize(Size.LARGE)
          .withCrust(Crust.THIN)
          .withTopping("Olives")
          .withTopping("Onions")
          .withTopping("Bell Pepper")
          .deliverToAddress("101 Main St., ...")
          .chargingCard(CardType.VISA, "1234-1234-1234-1234")
          );
System.out.printf("Pizza will arrive in %d minutes\n", time);
```

Closures Make It Easier

```
time = PizzaShop.orderPizza { pizzaShop ->
   pizzaShop.ofSize(Size.LARGE)
   .withCrust(Crust.THIN)
   .withTopping("Olives")
   .withTopping("Onions")
   .withTopping("Bell Pepper")
   .deliverToAddress("101 Main St., ...")
   .chargingCard(CardType.VISA, "1234-1234-1234-1234")
}
```

```
class PizzaShop
{
  private PizzaShop() {}

  public static int orderPizza(closure)
  {
    def shop = new PizzaShop()

    // Allow user to work with PizzaShop
    closure(shop)

    // process order after user is done
```


Use A Builder

- Do you have to burden your class with fluent interface?
- Nope
- You may create a builder or facade that will provide a fluent interface
- Users can use the builder to interact instead of the underlying class

Creating DSLs

- You may use tools like Antlr for working with DSLs
- JetBrains is working on Meta Programming System (MPS)
- Languages like Ruby and Groovy may it easier to create (internal) DSLs
- What makes Groovy attractive?
  - Classes are always open
  - Closures and Builders
  - Categories and ExpandoMetaClass
  - To some extent operator overloading
  - Parenthesis are almost optional (a bit lacking)

External DSLs Example

```
<project name="whatever" default="package">

    <target name="clean" description="Clean the build directory.">
        <delete dir="${proj.dist}"/>
    </target>
...
```

- Ant is a good example of an external DSL
  - The domain here is build
- Ant parses your build commands
- Can't exploit full capability of Java, however
- So, somewhat limiting or hard to extend

Internal DSLs Example

```ruby
require 'rake'
...

require File.join(File.dirname(__FILE__), 'lib/rails', 'version')

...

task :test do
  Dir['test/**/*_test.rb'].all? do |file|
    system("ruby #{file}")
  end or raise "Failures"
end

Rake::TestTask.new("regular_test") do |t|
  t.libs << 'test'
  t.pattern = 'test/**/*_test.rb'
  t.warning = true
  t.verbose = true
end
```

- Pure Ruby syntax

Gant similarly is pure Groovy

- Parsed by Ruby—no external parsing
- You can do as much Ruby in it as you like

Groovy Builder: XML

```groovy
import groovy.xml.MarkupBuilder

map = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'C#' : 'Hejlsberg']

markupBldr = new MarkupBuilder()

xml = markupBldr.languages {
  for(entry in map)
  {
    language(name : entry.key) {
      author (entry.value)
    }
  }
}
```

Groovy Builder: Swing

```
setLabel = { lbl -> lbl.setText(new Date().toString()) }

swingBldr = new groovy.swing.SwingBuilder()


frame = swingBldr.frame(
  title : "Example",
  size:[200, 100],
  layout: new java.awt.FlowLayout(),
  defaultCloseOperation:javax.swing.WindowConstants.EXIT_ON_CLOSE) {

  myLabel = label(text: "Hello")
  button(text : 'Click', actionPerformed: {  setLabel(myLabel) })
}

frame.show()
```


Creating a DSL in Groovy

```
ordering = false
orderInfo = [toppings : []]

def invokeMethod(String name, args)
{
  if (ordering)
  {
    if (name == 'toppings')
    {
      orderInfo[name] << args[0]
    }
    else
    {
      orderInfo[name] = args[0]
    }
  }
  else
  {
    // silenty ignore or throw exception...
  }
}
```

```
orderPizza {
size 'large'
toppings 'Olives'
toppings 'Onions'
toppings 'Bell Pepper'
deliverTo '101 Main St.,...'
chargeCard '1234-1234-1234-1234'
}
```

```
Here is your order:
deliverTo : 101 Main St.,...
chargeCard : 1234-1234-1234-1234
size : large
toppings : ["Olives", "Onions", "Bell Pepper"]
Pizza will arrive in 25 minutes
```

Using Categories

```
use(DatesUtil.class)
{
  println 2.days.ago.at(4.30)

  println 20.days.ago.at(16.30)
}
```

Type conversions

---

2 => int
.days => int
.ago => Calendar
.at => Date

```
class DatesUtil
{
  public static int getDays(Integer self)
  {
    self
  }

  public static Calendar getAgo(Integer self)
  {
    def today = Calendar.instance
    today.add(Calendar.DAY_OF_MONTH, -self)

    today
  }

  public static Date at(Calendar self, Double time)
  {
    def timeDbl = time.doubleValue()
    def hours = (int)timeDbl
    def minutes = (int)((timeDbl - hours) * 100)

    self.set(Calendar.HOUR_OF_DAY, hours)
    self.set(Calendar.MINUTE, minutes)
    self.time
  }
}
```


Using Expando

```
Integer.metaClass.getDays = {->
    delegate
  }

Integer.metaClass.getAgo = {->
  def today = Calendar.instance
  today.add(Calendar.DAY_OF_MONTH, -delegate)

  today
  }

GregorianCalendar.metaClass.at = {Double time ->
  def timeDbl = time.doubleValue()
  def hours = (int)timeDbl
  def minutes = (int)((timeDbl - hours) * 100)

  delegate.set(Calendar.HOUR_OF_DAY, hours)
  delegate.set(Calendar.MINUTE, minutes)
  delegate.time
}

println 2.days.ago.at(4.30)

println 20.days.ago.at(16.30)
```

Using Expando...

```
// Allows you to enhance hierarchy instead of a specific class
ExpandoMetaClass.enableGlobally()


//GregorianCalendar.metaClass.at = {Double time ->

Calendar.metaClass.at = {Double time ->
  ...
```


Categories vs. ExpandoMetaClass

- Categories allows you to tactically enhance a class
- ExpandoMetaClass is far reaching, global in nature
- You may not want to affect a class globally
- Categories provide controlled flexibility

Quiz Time

References

- A number of references from the following URL
- http://martinfowler.com/bliki/DomainSpecificLanguage.html
- http://docs.codehaus.org/display/GROOVY/Writing+Domain-Specific+Languages
- http://docs.codehaus.org/display/GROOVY/Gant
- http://groovy.codehaus.org/ExpandoMetaClass
- http://www.onboard.jetbrains.com/is1/articles/04/10/lop/
- http://www.jetbrains.com/mps

Thank You!

http://www.agiledeveloper.com — download